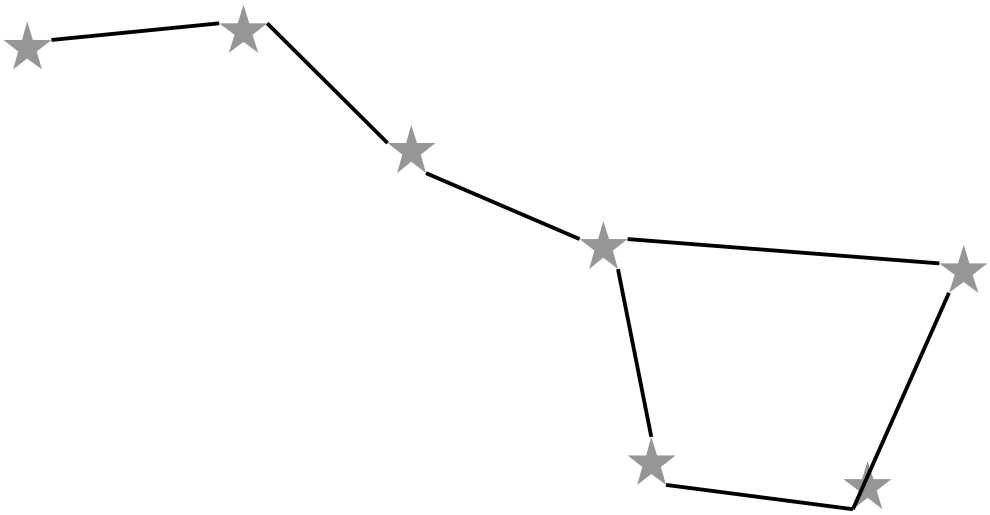
ГрафоМания



Редакция 02.02

от 2024-05-13

г. Воронеж, Россия

http://oleg.derevenets.com

E-mail: oleg.derevenets@gmail.com

Аннотация

Рассмотрены алгоритмы на графах и множествах. Неформальное изложение алгоритмов сопровождает работающий код с контрольными примерами, доведенными до числа. Код воплощен на объектно-ориентированном языке программирования **Delphi**. Подробно описана техника программирования задач, а листинги детально прокомментированы. Книга может служить дополнением к учебникам по дискретной математике, а также справочником по алгоритмам. Будет полезна студентам, аспирантам и программистам, решающим практические задачи на графах.

Условия использования

Оглавление

Вторая редакция от 13 мая 2024 г.

Вторая редакция книги (02.01) дополнена двумя главами:

Глава 29 — поиск паросочетания в произвольном взвешенном графе быстрым алгоритмом Эдмондса (Edmonds).

Глава 35 — решение замкнутой и разомкнутой задач коммивояжёра на графах и орграфах.

Нумерация глав, начиная с 29-й сдвинута. Исправлены мелкие технические и стилистические ошибки. Пояснительная часть некоторых глав существенно переработана.

В редакции 02.02 исправлены некоторые термины, опечатки и технические погрешности.

Предисловие

Графомания — мучительная хворь, симптом которой выражен в неодолимой тяге к маранию бумаги. Замечено, что поражает она организмы, ослабленные математикой. Иначе чем ещё объяснить обилие книг под общим названием «Теория графов»? Все они сочинены *математиками* для *математиков*, и лишь немногие доступны простым смертным: инженерам, студентам, и любознательным пенсионерам. И поскольку эти книги от *математиков*, там не обходится без их любимых теорем и доказательств. Впрочем, отбросим иронию, и воздадим хвалу всем талантам, коим мы обязаны этим украшением дискретной математики, тесно примыкающим к информатике и программированию.

Однако, в конечном счёте, любую теорию применяют на практике. И тогда то, что родилось в голове *математика*, попадает в голову *программиста*, становясь (если повезёт) работающей программой. На этом пути программист решает массу *своих* проблем: ищет оптимальные структуры данных, строит иерархию объектов, процедур и функций. При всём уважении к математике, он вынужден направлять усилия не на теоремы и доказательства, а на эти технические детали. Разумеется, что программисту надо ясно представлять воплощаемый в программу алгоритм, понимать идеи, лежащие в его основе, и без математической литературы ему не обойтись. И всё же книга, задуманная *программистом* для *программистов*, ему не повредит.

Итак, цель этой книги — ознакомить программиста с *идеями* решений задач на графах, а также с *техникой* кодирования этих задач. Идеи излагаются *неформально*, без теорем и доказательств, взамен чего даются ссылки на литературу. Потому книгу можно трактовать и как *популярную* теорию графов. И всё же программирование, как и математика, — дисциплина строгая и точная. Потому словесные пояснения и рисунки здесь дополнены работающим, практически «боевым» кодом. Этот подробно прокомментированный код подкрепляет пояснения: так, детали, не вполне понятные в словесном изложении, могут проясниться при изучении листингов. По этой причине при написании кода автор стремился сделать его в первую очередь *понятным* и *наглядным*, отдавая эффективности второе место (но не последнее).

Код написан на языке **Pascal**, точнее, на том его диалекте, что нынче зовётся **Delphi**. И тому есть, по меньшей мере, две причины. Во-первых, язык **Pascal** распространён в образовании, стало быть, многим знаком. К тому же он так прост, что листинг поймёт и тот, кто пишет на других языках. Во-вторых, **Delphi**-версия языка обладает мощным *объектно-ориентированным* аппаратом, как нельзя лучше подходящим для воплощения излагаемых идей. Уверен, что при должном понимании материала книги, читатель без особого труда переведёт код на любой другой нужный ему язык. Замечу, кстати, что задачи на графах — те ещё *загогулины*, — дают прекрасный полигон для обретения опыта объектно-ориентированного программирования (ООП). Почему бы не воспользоваться случаем?

В тщетных попытках объять необъятное, автор смог представить здесь лишь часть наиболее популярных алгоритмов на графах; перечень этих задач и сопутствующего материала дан в таблице:

Содержание материала	Главы
Знакомство с объектами, отношениями и множествами	2
Представление объектов в языке Delphi	3, 4, 5
Представление множеств, операции с множествами	6, 7
Понятие о сложности (трудоёмкости) алгоритмов	8
Задачи на множествах: • разбиение множества на подмножества; • задача о наименьшем разбиении (ЗНР); • задача о наименьшем покрытии (ЗНП).	9, 10, 11
Представление отношений графами	12
Программная реализация графов, ввод и вывод графов	13
Группа задач на достижимость: • взаимная достижимость вершин; • кратчайшие пути между вершинами; • выделение сильно связанных компонент.	14, 15, 16
Группа задач на размещение: • независимые вершины и клики; • доминирующие множества; • раскраски; • центры; • p-центры; • p-медианы.	17 – 22
Остовные деревья	23
Группа задач о потоках: • максимальный поток в сети; • поток, ограниченный сверху и снизу; • минимальная стоимость потока.	24 — 26
Паросочетания: • паросочетание в двудольном графе; • паросочетание в произвольном графе.	27 — 29
Цикл Эйлера и задача почтальона: • на неориентированном графе; • на орграфе.	30, 31
Задачи Гамильтона и коммивояжёра: • разомкнутая задача Гамильтона; • замкнутая задача Гамильтона (цикл); • комбинирование методов для задач Гамильтона; • замкнутая и разомкнутая задачи коммивояжёра.	32 — 35

Буду благодарен за полезные замечания и предложения, обращайтесь с ними по адресу:

oleg.derevenets@gmail.com

Свежую редакцию книги ищите на странице: http://oleg.derevenets.com

Глава 1 Вступление или «Когда я стану программистом?»

«Когда я стану программистом? — втайне терзается новичок, — сколько языков мне освоить, и каких? Что востребовано теперь на рынке? Вот выучу *этот* язык, затем *тот*, а уж потом...». И что *потом* ?

1.1. «Всё проходит»

Всё проходит — сокрушался мудрый царь Давид. Будь он программистом, то поправил бы: *всё проходит очень быстро*. Ещё бы! Всего лишь три десятка лет тому пара языков программирования вкупе с небольшой практикой вполне законно давали вам титул программиста. Ныне же соискателей звания терзают, по меньшей мере, два вопроса, второй из которых таков: как долго я останусь программистом? Ведь новые языки всё плодятся и плодятся, а модные технологии и направления стремительно меняют ландшафт нашей профессии. Только вчера ты освоил новинку и был «на коне», а сегодня ловишь на себе сочувствующие взгляды юных конкурентов. Коллеги не дадут соврать: удел программиста — грести против течения: чуть сбавил усилия, бросил вёсла, и уже влечёт тебя поток к опасным порогам. Где якорь, что остановит гибельное движение? Где опора о твёрдое дно бурлящего потока? Когда новичок станет программистом и как долго останется им?

Предлагаю вам, проявив каплю терпения, и одолев десяток страниц, самим ответить на эти важные для многих вопросы. Со стороны сфера программирования видится пышным раскидистым деревом, в котором некоторые ветви заметно увяли, зато другие бурным ростом сулят заманчивые перспективы. И всё же это только ветви… Мы же докопаемся до корней, и начнём издалека.

1.2. История трёх революций

Прежде, чем двинуться к цели, выясним исходную позицию. Где место программиста в современных технологиях? Как ни странно, мы найдём его через знакомство с тремя кибернетическими революциями.

1.2.1. Революция первая — биологическая

Она случилась миллионы лет тому с появлением первых примитивных одноклеточных организмов. Своё пропитание — энергию, они брали, главным образом, от солнца. А надо заметить, что уже тогда, задолго до рождения Дарвина, работали его законы: мутация и естественный отбор. Мутации — это случайные изменения организма под действием внешних факторов. Мутации могут быть и полезными, и вредными — вердикт о полезности выносит естественный отбор.

И вот, вследствие мутаций и отбора, часть микробов приноровилась брать дополнительную энергию из останков своих соплеменников, что усилило их конкурентное преимущество, — так появились наши предки — животные, и был

сделан первый шаг в последующей истории. Известно, однако, что под лежачий камень вода не течёт: мудрено прокормиться, сидя на месте, и питаясь лишь ближайшими соседями. Второй шаг эволюции привёл к случайному появлению примитивных органов движения — ресничек. И естественный отбор утвердил новинку: ведь даже хаотическое перемещение в среде обитания существенно обогащало рацион обладателей ресничек.

На третьем этапе эволюция сотворила примитивные органы чувств — своего рода датчики, которые воздействовали прямо на органы движения. Датчики могли реагировать на освещённость, температуру среды обитания и её химический состав. Движения организмов обрели некую степень разумности: примитивное животное двигалось не куда попало, а в направлении лучших условий обитания.

По мере развития появились промежуточные звенья, передающие сигналы от органов чувств к органам движения — прообраз нервной системы. Этот посредник помаленьку занялся логической обработкой сигналов, что сделало животных «умнее». Нервная система усложнялась и совершенствовалась миллионы лет, что и породило современных животных и человека. В нервной системе (мозгу) животного отражены и среда существования, и его роль в этой среде. Если отражение окружающей действительности назвать *моделью*, то мозг и нервная система, безусловно, являются *биологической моделью* окружающей среды и той роли, которую играет в ней данный организм.

Отметим две особенности биологической модели. Во-первых, она передаётся только по наследству от родителей к потомкам: животные не могут передать генетический опыт иначе. Конечно, высшие животные иногда обучают потомство, организуют коллективную охоту и оборону, но это не меняет врождённую модель. Отсюда следует другая её особенность: биологическая модель крайне консервативна, она не успевает приспособиться к резким изменениям условий обитания.

1.2.2. Революция вторая: символические модели

Старт ей дали примитивные изображения животных и людей, вырубленные на камне грубой рукой древнего художника. Восхищение от этих каракулей не уступало тому, что возбуждает сейчас заезжая рок-группа у обитателей провинции. Сегодня нам трудно их понять, но ведь тогда человек впервые нутром почуял возможность *самому* создать *модель* какой-то части окружающей его природы. Эта модель, в отличие от модели биологической, оказалась доступной всем зрячим соплеменникам художника: и живущим, и будущим поколениям. Модель можно было дополнять, копировать, развивать, и для этого не требовались миллионы лет эволюции или участие автора прототипа.

Первые письмена (египетские, например) были рядом картинок, восточные иероглифы, вероятно, также порождены рисунками. Позже возникли современные алфавиты, не связанные с изображениями чего-либо, а также цифры, математические знаки, нотная запись и т.д. Появились литература, науки, художества и музыка. Всё это — современная культура, которая, подобно древним

рисункам, являет совокупность *моделей* окружающего мира, но, в сравнении с наскальным рисунком, куда более совершенных. Эти модели объединяет то, что все они используют символы, и потому их называют *символическими*. Модели хранят в себе накопленные знания и служат, как и биологические модели, всё той же цели: обретению преимуществ в борьбе за выживание. Но теперь мы говорим о выживании не только отдельных людей или групп, — речь о государствах и человечестве в целом.

Отметим, что в масштабах истории Земли культура развилась с быстротой лавины: с момента её зарождения прошли не миллионы, а тысячи лет, особенно бурным выдалось предыдущее столетие. Однако накопившийся к середине 20-го века громадный объём знаний породил новую проблему.

1.2.3. Революция третья — «кибермозг»

Символические модели — науки, искусства — создаются людьми и оживляются ими же: даже самая удачная модель — научная теория — мертва без человека. Модель оживает только в головах тех, кто постиг её ценой обучения. Так, работа моделей, связанных с созданием машин, возможна лишь через хорошо образованных людей: конструкторов, инженеров, рабочих.

Известно, что в деле инженера, наряду с творческой, немало и черновой работы, а именно рутинных вычислений по известным формулам (а формулы — это тоже символические модели). Вычисления утомляют экономистов, бухгалтеров, и других работников умственного труда. К середине 20-го века с усложнением технологий эта рутина буквально задавила специалистов, не оставляя им времени на творчество. Разверзлась пропасть между количеством и качеством научных знаний с одной стороны, и возможностью их применения с другой. По-прежнему рождалось много полезных, но не осуществимых по причине их сложности моделей, — и научно-технический прогресс затормозился.

И тогда — не чудо ли? — на смену счётам явились электронные вычислительные машины (ЭВМ). Первые ЭВМ служили, главным образом, для выполнения громоздких вычислений по формулам. Не зря название одного из ранних языков программирования — ФОРТРАН — так и расшифровывается: ФОРмульный ТРАНслятор. Постепенно компьютеры и языки программирования стали приспосабливать к переводу в электронную форму всё более сложных символических моделей. И теперь компьютеры понимают речь, различают образы, процессоры спрятаны в телефонах, микроволновках и другой технике — всего не перечислить. Куда двинулся прогресс? — в сторону перевода *символических* моделей в *электронную* форму. И программист оказался в самой гуще этой третьей стремительной кибер-революции. Какова ж его роль?

1.3. Модели, модели, модели…

Прежде, чем уяснить роль программиста, бегло ознакомимся со свойствами и назначением того, что мы называем *моделями*.

1.3.1. Много ль в корыте корысти?

Любая модель (биологическая, символическая) служит для *предсказаний*. Мозг хищника *предсказывает* поведение жертвы и строит на этом предсказании план охоты. А жертва пытается *предсказать* поведение хищника и планирует спасение. Мозг тренированного игрока *прогнозирует* полёт мяча. Такова роль биологических моделей.

Тому же служат и символические модели. Инженерные расчёты *предсказывают* поведение будущей машины или прочность постройки. Социально-экономические модели, пусть и не вполне точные, с некоторой вероятностью *предрекают* грядущее. Даже такая расплывчатая модель, как художественное произведение, даёт, наряду с эстетическим наслаждением, представление о сторонах жизни, с которыми мы не сталкиваемся повседневно. Входя в положение героев романа или фильма, мы невольно *прогнозируем* своё поведение в сходных обстоятельствах. Итак, назначение любой модели — *предсказание*.

1.3.2. Неформализованные модели: искусство и философия

Поскольку разного рода искусства обладают некоторой *предсказательной* силой, мы причисляем их к моделям. Художественные произведения обычно фиксируют на бумаге или на других носителях посредством символов: буквами, нотами; стало быть, эти модели — символические. Но воздействуют они на нас не символами, а *образами* или *звуками*, порождаемыми символами, — именно они вызывают эмоции.

К искусствам примыкает философия, которая тоже выражается символами (словами) и оперирует образами. Однако если объектами искусств являются *конкретные* образы, взятые из природы (люди, пейзажи, звуки и т.д.), то в голове философа рождаются *абстрактные* образы — идеи, отражающие, по его мнению, общие законы природы.

Искусства и философию объединяет их *неформализованность*. Понятно, что машину не увлечёшь ни фильмом, ни музыкой, её «мозг» не породит философских мыслей. Да и зачем ей это? — оставим приятное себе.

1.3.3. Формализованные модели: описательные науки и математика

Обратимся теперь к точным наукам, к тому, что можно хотя бы отчасти *формализовать* и переложить на компьютерные плечи. Науки делят на описательные и теоретические, хотя чёткой грани тут нет. География, биология, химия — что это? Скорее описательные науки. Но чем больше в науке математики, тем сильнее она смещается в сторону теоретическую, — такова физика, например. Крайней гранью является сама математика, объекты которой в природе не встречаются, хотя и навеяны ею. Подобно идеям философов, они являют плод

размышлений математиков. Описательные науки соотносятся с теоретическими так же, как искусства соотносятся с философией: описательные исследуют конкретные объекты природы, а теоретические — абстрактные, воображаемые объекты (табл. 1-1).

Табл. 1-1 — Классификация наук
в зависимости от языка и изучаемых объектов

Используемый язык	Характер объектов	
	Конкретные объекты	Абстрактные объекты
Неформализованный	*Искусство*	*Философия*
Формализованный	*Описательные науки*	*Теоретические науки*

1.3.4. Царица наук

Царица наук — это математика, конечно. Не будучи прямо привязана к реальным объектам природы, она, тем не менее, проникает во все естественные науки, включая описательные. Все теории, претендующие на сколь-нибудь точные предсказания, не ограничиваются словами, а выражаются строгим языком математики.

1.3.5. Универсальность и обратимость математических моделей

Рассмотрим следующую формулу:

A = B • C

Угадайте, какую природную закономерность она выражает? Подсказки ищите в таблице, где та же формула слегка перелицована и снабжена пояснениями:

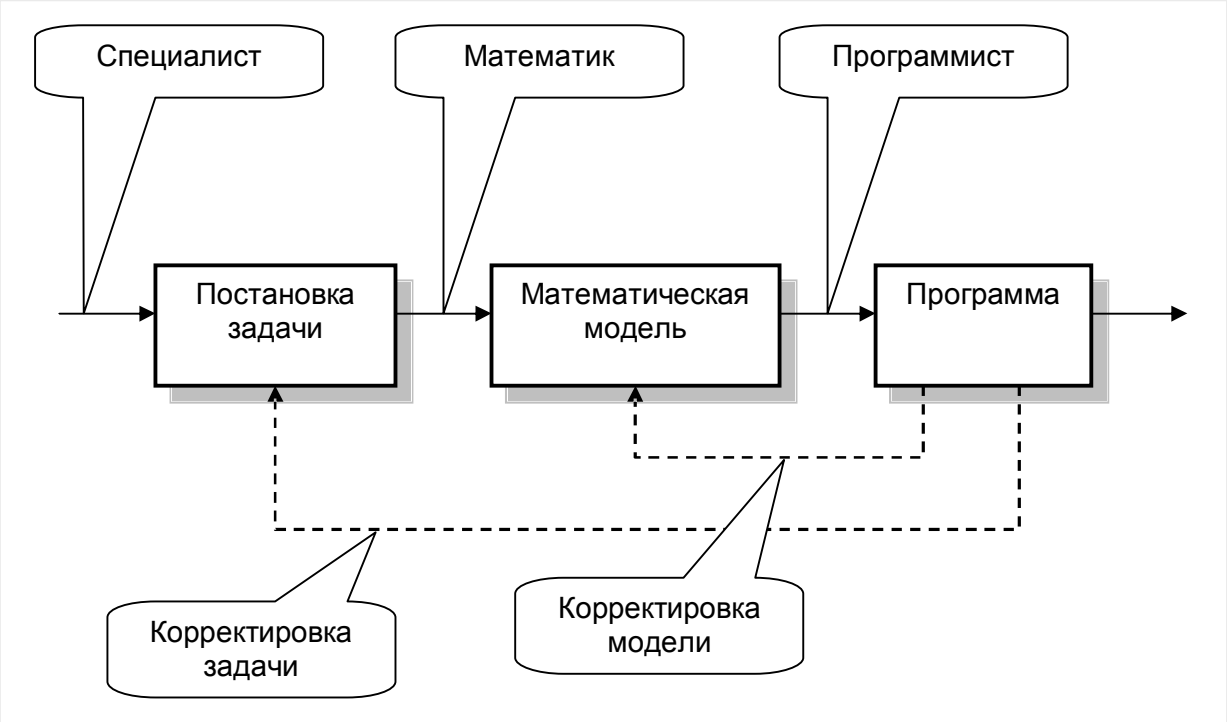
Формула	Обозначения
S = H • L	S — площадь прямоугольника H — его ширина L — его длина
L = V • T	L — преодолённый путь V — скорость T — время
S = C • N	S — стоимость партии товара C — цена единицы товара N — количество товара

Список примеров легко продлить, но и так ясно: простейшая модель, выраженная формулой умножения, весьма востребована. Эта *универсальность* характерна почти для всей математики. Существуют, разумеется, и очень специфические модели (например, математическая модель ядерного реактора), но и они строятся на базе тех же простых универсальных моделей.

Отметим, также, *обратимость* математических моделей. Так, из формулы, связывающей путь, скорость и время, можно через простейшие преобразования определить любой из параметров по двум остальным.

1.4. На троих

Теперь мы близки к тому, чтобы указать место программиста в новой, третьей кибернетической революции. Рассмотрим упрощённую схему разработки какой-либо компьютеризированной системы. Ею может быть и сложная система управления самолётом, и сравнительно простая бухгалтерская программа.



Разработку любой такой системы начинают с *постановки задачи*, — за это отвечают специалисты в данной области. Они должны чётко и однозначно описать исходные данные и все требования к будущей системе. Затем эстафету принимают математики, которые ищут методы решения поставленных задач, подбирают готовые алгоритмы, или же строят новые, и доказывают их правильность. Одним словом, строят математическую *модель*. Программистам же остаётся перенести эту модель с бумаги в электронные «мозги» компьютера.

На первый взгляд всё просто и ясно: здесь каждый делает своё дело. Но жизнь богаче и сложнее схем. Хорошо, если постановщик задачи может создать и математическую модель. В других ситуациях роль математика берёт на себя программист. Когда же роли разделяются (как на схеме), на стыках возможны проблемы: специалист затрудняется сформулировать задачу, математик — не всегда понимает специалиста, а программист — математика (тут сказывается уровень его математической подготовки). Как бы то ни было, конечным продуктом является система. Если она работает не так, как надо, следует «разбор полётов». В

ходе разбирательства могут исправляться и *программа*, и *математическая модель*, и даже *постановка задачи*.

Итак, тесная связь программиста с математиком очевидна. Программисту надо, по меньшей мере, владеть языком математики и понимать выражаемые им идеи. Именно это понимание, а не владение модным языком программирования, будет фундаментом и будущих ваших успехов, и профессионального долголетия.

1.5. Математика и математики

Раз уж мы коснулись математики, припомним свою школу. Основу математики составляют три сущности: аксиомы, теоремы и доказательства. *Аксиома* — это базовое утверждение, принимаемое без доказательств. *Доказательство* — это цепочка рассуждений, подчинённых нескольким простым правилам логики. Наконец, *теорема* — это утверждение, истинность которого вытекает из аксиом и доказана цепочкой рассуждений. В этом вся суть математики.

Приступая к очередной задаче, математик, прежде всего, включает интуицию и угадывает конечное или промежуточное решение. Как угадывает? — загадка. Но заподозрив решение, он должен показать его истинность, доказав одну или несколько теорем. Пока решение не доказано, это лишь догадка, *гипотеза*, а не теорема. Порой гипотеза бывает интуитивно понятной, но трудно доказуемой, некоторые доказательства даются лишь потомкам спустя десятки, сотни лет. Но, обретя доказательство, математик «умывает руки» — его работа сделана.

А что же программист? Обязан ли он доказывать теоремы? Если он изобрёл что-то новое, то придётся. Но чаще программист берёт готовые плоды математических усилий, и вправе освободить себя от перепроверки доказательств, — в конце концов, каждый грызёт свою морковку. Программисту важнее сосредоточиться на том, суть чего выражается в следующих вопросах:

- Какую реальность описывает данная математическая модель?
- Что за идеи лежат в основе решения?
- Как эффективно организовать данные и скомбинировать процедуры, чтобы превратить эту математическую модель в работающую программу?

К счастью, в отличие от изучающих математику инженеров других специальностей, у программиста есть возможность превращать изучаемый материал в работающие программы. Создание программ, оживляющих сухую математику, — прекрасный способ и освоения математики, и обретения опыта программирования, — *это наш путь*.

1.6. Дискретная математика

Математика включает много ветвей, крупных и мелких ответвлений. То, что применяют в инженерном деле и других науках, называют *прикладной* математикой. В ней, в свою очередь, выделяется ветвь, вплотную примыкающая к

компьютерной науке — информатике. Речь о *дискретной* математике, грань между которой и информатикой трудно различима. Но и дискретная математика весьма обширна, и потому в этой книге коснёмся лишь двух её разделов, на взгляд автора, наиболее интересных и полезных для программиста: *теории множеств* и *графов*.

У математики особый язык, и чтение математических книг, рекомендованных мною в списке литературы, предполагает владение оным. Но здесь я не буду «грузить» читателя чистой математикой, ведь эта книга для программистов. Не ищите здесь милых математикам крючков и загогулинок, — обойдёмся без них. Всё, что надо, я объясню «на пальцах». Мы не будем доказывать теорем, зато напишем море программ. Так будут убиты оба зайца, которых мне не жаль: вы освоите много интересных и важных алгоритмов, а заодно научитесь создавать весьма сложные программы. В путь!

В конце каждой главы подводятся краткие итоги, а также даётся список рекомендуемой литературы, в котором особо отмечаются книги по теме текущей главы.

1.7. Итоги

- В ходе эволюции живых организмов природа создала биологические кибернетические системы. Главное назначение этих систем — предсказание будущего с целью повышения конкурентных возможностей индивидуума.
- Развитие человека и общества породило второй уровень кибернетических систем — символические модели, знакомые нам в виде наук и искусств. Эти системы тоже обладают предсказательной силой, и служат тем же целям: росту конкурентных преимуществ отдельных людей, обществ, стран, и человечества в целом.
- Третий этап развития кибернетических систем связан с внедрением компьютеров: символические модели переводятся в электронную форму, и в этой работе не обойтись без программиста.
- Главное конкурентное преимущество хорошего программиста состоит в способности воспринимать, анализировать, и превращать в работающие программы сложные символические модели, описывающие явления окружающего мира. Поскольку эти модели изложены языком математики, степень владения этой наукой определяет и квалификацию, и профессиональное долголетие программиста.

1.8. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
√	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 2
Объекты, множества и отношения

Дискретная математика манипулирует с неделимыми, то есть, *дискретными* объектами, — обычно это символы или числа, хотя ими наука не ограничена. Здесь мы бегло ознакомимся с понятиями, которые являются предметом этой книги. Наш обзор не претендует на полноту и строгость, — то и другое вы найдёте в предлагаемой литературе.

2.1. Понятия и объекты

Не легко формулировать простые, интуитивно понятные вещи. Что такое *неделимый объект* — предмет манипуляций дискретной математики? Ответ кроется где-то внутри нас: поставьте перед собой фотографию или картину. Если это не художество абстракциониста, вы найдёте там нечто знакомое: людей, животных деревья, цветы. Хотя перед вами лишь клочок крашеной бумаги, никаких предметов там нет! Более того: всё, что мы видим, — это лишь матрица «пикселей» на сетчатке глаза. Однако наш мозг без труда выделяет в этом море пикселей знакомые предметы, запросто отличая кошку от собаки (кстати, невероятно трудная для компьютера задача).

Эта чудесная способность выработана годами эволюции, — в мозгу сформировались *понятия*, полезные для выживания организма. Сложились *понятия* не только о предметах в целом, но и о частях предметов: «плавник», «усы», «хвост». Более того, возникли и нематериальные *понятия*. Таких предметов, как «покой», «шутка», «родство», «мысль», в природе не существует, но мы представляем себе, что это такое, имеем *понятия* о них. А что выражают прилагательные и глаголы? Ими тоже передаются *понятия*, и вообще, всё, что мы ощущаем как *нечто, отличное от другого*, является *понятием*. Однако понятия, выражаемые существительными, нам особо дороги, мы назовём их *объектами*.

2.2. Дискретные объекты

Возьмём три предмета: вода, воздух, супчик, — это *вещества*, таким объектам свойственно отсутствие определённой формы, а также способность оставаться тем же самым при разделении на части. Вода остаётся водой хоть в реке, хоть в кружке, хоть в пригоршне.

Существуют предметы иного рода, форма которых относительно неизменна, и возможность деления отсутствует, поскольку разделённый на части объект прекращает существование. Так, мяч после серии ударов остаётся всё тем же узнаваемым мячом. Но нельзя, разрезав, сделать из него два или три мяча. То же будет и с кошкой. Такие объекты называют *дискретными*.

Неделимы, *дискретны* и многие нематериальные объекты: не бывает кусочков теоремы или ломтиков настроения. Далее мы будем оперировать лишь

такими неделимыми *дискретными объектами*, как материальными, так и нематериальными.

2.3. Различимость и свойства

Любые два объекта *различимы*, *уникальны*, иначе это был бы один и тот же объект. Часто они различимы ввиду отличия их естественных *свойств*, как то: цвета, размера, веса и т.п. Абсолютно одинаковых объектов нет, но даже предметы, которые кажутся одинаковыми (шарики подшипника), различимы по своему месту в пространстве. Иногда такие предметы мы снабжаем дополнительными, искусственно введёнными свойствами. Например, сходящие с конвейера изделия снабжают уникальными заводскими номерами.

2.4. Простое и сложное

Дискретные объекты по определению неделимы, элементарны, однако при этом могут обладать сложной внутренней структурой. Здесь нет противоречия, поскольку при разрушении структуры объект перестаёт быть самим собой. Не зря говорят: из песни слова не выкинешь. Потому один и тот же объект может трактоваться и как *элементарный*, неделимый, и как сложная составная *система*. Например, участникам дорожного движения удобно считать автомобили элементарными объектами, но в ремонтной мастерской преобладает другая точка зрения. Или возьмём город. Даже небольшой посёлок обладает весьма сложной структурой, включающей дома, коммуникации, дороги. Но на глобусе всё это обозначают лишь кружком — элементарным объектом.

Эта двойственность объектов никогда не затрудняла математиков, наделённых правом всё обозначать буквами, но озадачивала программистов, оперирующих со сложными структурами данных. К счастью, современные объектные технологии сняли остроту этой проблемы.

2.5. Абстрактное и конкретное

Сколько самых разных понятий роится в нашей голове? Не будь они как-то упорядочены, разложены «по полочкам», мозг утонул бы в этой информационной каше. И над этим тоже потрудилась эволюция, породив в мозгу *абстрактные* понятия о *конкретных* объектах. Конкретные объекты мы наблюдаем наяву: вот урчит у ног наша серая *кошка*, а там крадётся соседский рыжий *кот*.

В борьбе за выживание побеждали те, кто быстрее и точнее реагировал на события, и потому мозг стал выделять в объектах или ситуациях наиболее *существенные* для принятия решения признаки. И теперь всех на свете *конкретных* кошек мозг представляет моделью *абстрактной* кошки вообще. Эта абстрактная кошка обожает молоко, гоняет птиц, а порой и мышей ловит, — всё это тотчас представляется нам при слове «кошка». Чья она, какой породы и масти? — этими менее существенными деталями мы интересуемся во вторую очередь.

На абстрактных понятиях первого уровня эволюция не успокоилась. Из кошек, собак, лошадей и прочей живности составилось ещё более абстрактное понятие «животные». Последующее абстрагирование — порождение из частных понятий более общих — ведёт нас к вершине этой иерархии понятий — *абстрактному объекту*, — для нас это нечто, отличное от другого, с этого мы начали главу. Стало быть, *абстракция* — отбрасывание несущественных признаков — это природный процесс, который усовершенствовал наш мозг.

2.6. Классификация и множества

Итак, своему биологическому выживанию мы обязаны естественной, природной абстракции. Но с тех пор, как люди занялись наукой и стали строить символические модели, пошёл и обратный процесс — *конкретизация*. Иначе говоря, перенося «на бумагу» то, что представлено в наших головах, мы стали раскладывать это по полочкам, *классифицировать*. Так возникло понятие *множества*, которое немецкий математик Кантор определил как *объединение в одно целое объектов, хорошо различимых нашей интуицией и нашей мыслью*.

2.6.1. Универсум

Определение Кантора позволяет «бросить в один мешок» всё, что мы можем себе представить, все конкретные и абстрактные объекты. Множество, содержащее все мыслимые объекты, называется *универсумом*. На первый взгляд такое множество кажется абсолютно бесполезным. Однако универсум является тем резервуаром, из которого черпаются объекты для всех прочих множеств. Другими словами, все сколь-нибудь полезные множества являются подмножествами универсума.

Универсум единственен, но иногда это понятие намерено сужают. Например, при классификации всего живого, биологи не рассматривают неживые объекты, и для них универсум — это все живые организмы.

2.6.2. Предикаты

Объекты, входящие в полезные множества, выделяют из универсума путём применения *предикатов* — утверждений об объектах, которые могут оказаться либо истинными, либо ложными. Предикату можно сопоставить булеву функцию в языке **Pascal**. Так, например, для выделения из универсума всех живых объектов можно вообразить такую функцию:

```
function ОбменВеществ(объект): boolean;
```

Функция возвращает истину, если объекту-аргументу свойственен обмен веществ. Тогда сформировать множество всего живого можно перебором всех объектов универсума с применением условного оператора:

```
If ОбменВеществ(объект)
  then МножествоЖивого:= МножествоЖивого + объект
```

Действие предикатов обычно основано на свойствах объектов.

2.6.3. Эквивалентность

Об объектах одного множества говорят, что они *эквивалентны* в некотором смысле. Так, например, множество предметов белого цвета составляют белые объекты, — они эквивалентны в смысле одинаковой окраски. Множество чётных чисел содержит те, что при делении на два дают в остатке ноль — они эквивалентны в этом смысле. Другими словами, любому *множеству* соответствует некий предикат, и все его объекты *эквивалентны* в смысле истинности этого *предиката*.

2.6.4. Выделение подмножеств

Классификация состоит в разбиении множества на менее крупные *подмножества*. Для этого нужны дополнительные предикаты: каждый из них разбивает исходное множество ровно на два непересекающихся подмножества: в первое попадают объекты, для которых предикат истинен, а во второе — для которых он ложен.

Разобьём множество всех автомобилей по цвету: белые, чёрные и красные (рис. 2-1). Здесь нужны три булевы функции, подобные той, что представлена выше (нужны три предиката). Применение ко всем автомобилям предиката **Белый** разобьет множество на «белые» и «не белые». Второй предикат разобьёт «не белые» на «чёрные» и «не чёрные». Предикат **Красный** разбивает подмножество «не чёрные» на «красные» и «не красные». Последнее подмножество можно подвергнуть дальнейшему разбиению.

Одно и то же множество может быть разбито разными предикатами на несколько разных подмножеств, как пересекающихся, так и не пересекающихся. Так, например, все автомобили могут быть классифицированы как по своему назначению (легковые, грузовые, автобусы и т.д.), так и по цвету. И тогда подмножество белых автомобилей наверняка пересечётся с подмножеством легковых.

Рис. 2-1 — Разбиение множества автомобилей тремя предикатами

2.6.5. Множества как объекты

Напомню, что согласно Кантору элементами множеств могут быть любые объекты. Но ведь само *множество* — это тоже *объект*, стало быть, множества могут входить в другие множества в качестве элементарных объектов. А множества множеств, в свою очередь, тоже можно трактовать как элементарные объекты и включать в другие множества, и т.д. Скоро и нам пригодятся эти трюки.

2.6.6. Операции с множествами

Если бы множества славились только своею универсальностью, «всеядностью», теория множеств не царствовала бы в математике. Силу ей даёт возможность *оперировать* с множествами. Важнейшими из этих операций являются *объединение* и *вычитание* множеств. На них строится ряд других операций, а также море полезных алгоритмов.

2.7. Отношения и графы

Итак, мы выяснили, что множества — один из тех инструментов, посредством которых человек строит символические модели, исследуя и познавая мир. Отметим, что классификация объектов, применяемая при построении множеств, основана лишь на *свойствах* объектов. Но с расширением круга решаемых задач всё трудней моделировать природу, опираясь лишь на свойства объектов. Многие явления объясняются не столько *свойствами*, сколько *отношениями* между объектами, их взаимными *связями*. Так, к примеру, поведение людей часто определяется не сколько их личными качествами, сколько отношениями в обществе (начальник-подчинённый, учитель-ученик). Понимание важности отношений породило, в конце концов, *теорию графов*.

2.7.1. Бинарные отношения

Бинарное отношение отражает некую связь между *двумя* объектами. Связи могут быть как зримыми, осязаемыми (дороги, провода), так и мыслимыми, воображаемыми (отношения подчинённости или родства). Отношения возможны только в паре, но иногда роль «напарника» исполняет тот же самый объект, — такое отношение называют рефлексивным. К примеру, любого человека можно считать родственником самому себе.

Пара объектов может быть связана сразу несколькими отношениями разного рода. Например, два человека могут быть одновременно: а) родственниками, б) сослуживцами и в) соседями по даче. В теории графов рассматривают обычно лишь однородные отношения, когда между двумя объектами существует не более одной связи, и соответствующие графы называют *простыми*. Простые графы могут обладать самой причудливой «геометрией»: объекты могут быть связаны с одним или несколькими другими объектами, либо сами с собой, а то и вообще не связаны с другими, изолированы.

2.7.2. Свойства отношений

Отношения характеризуют тремя свойствами: *рефлексивностью*, *симметричностью* и *транзитивностью*. Отношение может либо обладать, либо не обладать любым из этих свойств, что порождает несколько типов отношений.

Рефлексивность подразумевает возможность связи объекта с самим собой. Так, например, отношение «родственник» рефлексивно (каждый сам себе родня), а отношение «родитель» — нет (никто не рождает сам себя).

Симметричность отношения предполагает его взаимность. Так, отношение «одноклассники» симметрично, а отношение «родитель» — нет (сын не может быть родителем отца).

Транзитивным называют отношение, которое можно перенести со второго объекта на третий, четвёртый и последующие объекты. Так, отношение «родитель» не транзитивно: если некто родил сына, а этот сын — внука, то это не значит, что дед родил внука. Однако отношение «предок» транзитивно: и отец, и деды, и все прадеды являются предками человека. То же можно сказать об отношении «потомок».

Рефлексивные и симметричные отношения отмечают на графах линиями и стрелками так, как показано на рис. 2-2.

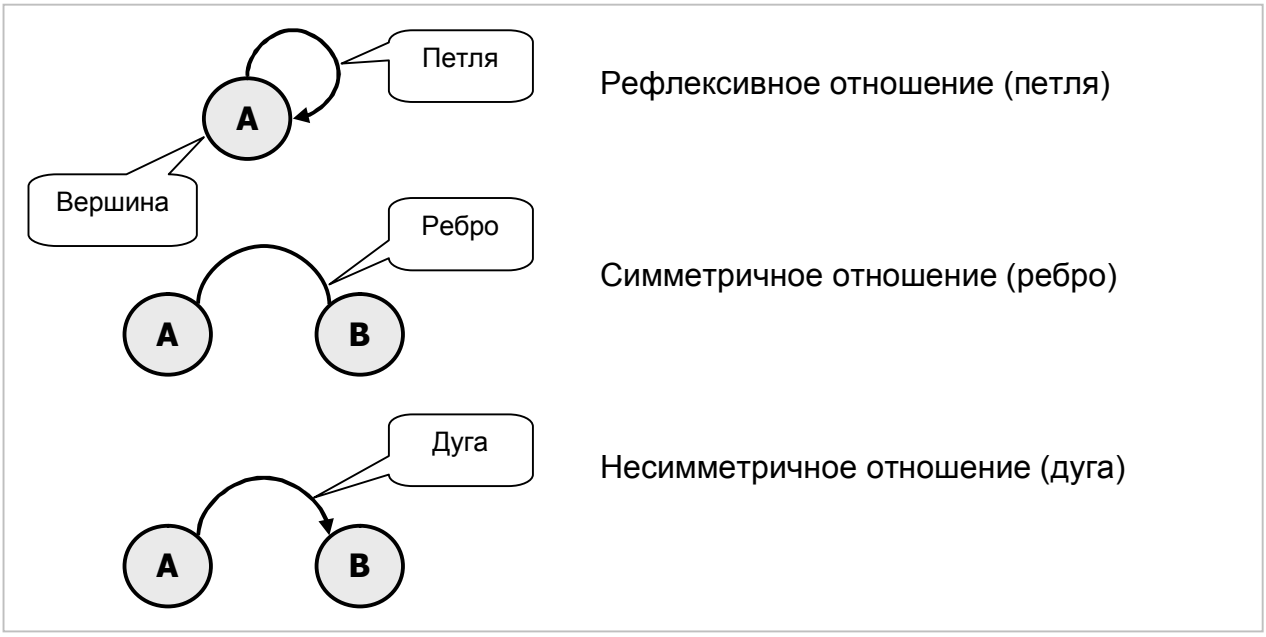


Рис. 2-2 — Графические обозначения отношений

2.7.3. Связи — это тоже объекты

Итак, на рис. 2-2 показаны примеры графов. Объекты здесь изображены кружками, их называют *вершинами* или *узлами* графа. Вершины содержат в себе некую информацию об объектах, например, название города, его населённость и т.п. Связи между вершинами обозначают *дугами* (со стрелкой) или *рёбрами*, они тоже могут содержать информацию, например, о расстоянии между городами.

Отметим, что объектами являются не только вершины графа, но и связи между ними (дуги, рёбра). Стало быть, и вершины, и связи можно сгруппировать в два множества: множество вершин X, и множество связей A. Тогда граф G в целом будет множеством, состоящим из двух элементов-множеств, что обычно записывают так: $G = \langle X, A \rangle$.

2.8. Объекты, объекты, объекты...

В начале этой главы мы дали понятие элементарного дискретного объекта. Затем узнали, что в ходе построения математических моделей объекты объединяются в множества, а множества сами по себе тоже можно трактовать как дискретные объекты. Далее уяснили, что многие модели могут быть построены лишь с учётом связей между объектами, причём эти связи тоже являются объектами. В конце концов, граф в целом состоит из двух множеств: множества вершин и множества связей, и тоже является объектом. Всё это наводит на мысль сконструировать *иерархию объектов*, способную вместить в себя всё богатство этого понятия — *объект*. Эту задачу начнём решать, начиная со следующей главы.

2.9. Итоги

- В ходе эволюции в мозгу человека сформировались *понятия*, — они ощущаются как нечто, отличное от другого.
- Существует две категории понятий: *вещества* и неделимые *дискретные* объекты. К последним относятся и реальные предметы, и части предметов, и абстрактные понятия, такие, как «мысль», «настроение», «теорема».
- Дискретный объект нельзя разделить, не разрушив. В то же время он может обладать внутренней структурой и состоять из других дискретных объектов.
- Дискретные объекты обладают рядом свойств, на основе которых могут группироваться в *множества* и *подмножества* (посредством предикатов). Операции с множествами составляют основу современной математики.
- При построении моделей, отражающих реальность, учитывают не только свойства объектов, но также их *отношения*. Отношения в математике моделируют посредством *графов*.

2.10. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
√	11	Турчин В.Ф.	Феномен Науки	
√	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 3
Объекты в языке Delphi

Задача программиста — перенос моделей из дискретной математики в «мозг» компьютера. То, насколько легко мы справимся с этой задачей, отчасти определяется выбранным языком. Многим дискретным объектам присуща двойственность: с одной стороны они трактуются как элементарные, а с другой — представляют собой сложные структуры данных (массивы, множества или даже графы). Были времена, когда эта проблема решалась сложными окольными путями, но с появлением объектно-ориентированных языков всё изменилось: объекты, словно нарочно, придуманы именно для этой цели.

Мы воспользуемся языком **Delphi**, известным по одноимённой среде программирования. **Delphi** — это развитие языка **Object Pascal**, а тот, в свою очередь, порождён от **Borland Pascal**, хорошо знакомого многим студентам и школьникам. Надеюсь, это облегчит усвоение представленного здесь материала. Отметим, что во многих книгах при объяснении алгоритмов тоже предпочитают паскале-подобный синтаксис ввиду его простоты и выразительности.

Хотя язык **Delphi** в своём развитии ушёл далеко от базового Паскаля, для восприятия книги не требуется досконально владеть его тонкостями, — хватит основ объектно-ориентированного программирования (ООП), приобретённых в рамках **Borland Pascal**. Наиболее существенные новшества и отличия от **Borland Pascal** будут пояснены в этой главе. Тем, кто не владеет основами ООП, рекомендую сначала обратиться к предлагаемой литературе, а если вы хорошо знакомы с **Delphi**, можете без ущерба пропустить эту главу.

3.1. Классы и объекты

Начнём с синтаксического новшества касаемо объявления объектов: если в **Borland Pascal** для этого применялось ключевое слово **Object**, то в **Delphi** для этой цели предусмотрено ключевое слово **Class**, например:

```
type   T1 = class(TObject)   end;
```

Здесь объявлен объектный тип данных **T1**, произведённый от предопределённого в языке типа данных **TObject** (в данном случае он совпадает с предком). Таким образом, в языке чётко разделены два понятия: *класс* — это *тип* данных, *объект* — это *переменная* объектного типа.

3.2. Объекты – это динамические переменные

Следующая отличие **Delphi** состоит в том, что переменные объектного типа всегда являются *указателями* на динамические переменные, — статических объектов в **Delphi** не предусмотрено. Таким образом, для работы с объектами

недостаточно одного лишь объявления: надо ещё создать объект конструктором, а впоследствии удалить из динамической памяти, например:

```
var X: TObject;  // объявление
begin
    X:= TObject.Create;   // создание объекта
    { работа с объектом... }
    X.Free;               // освобождение объекта
end.
```

3.2.1. Разыменование по умолчанию

Поскольку компилятору языка **Delphi** известно о динамической природе объектов, для доступа к полям и методам не требуется разыменования таких переменных стрелочкой «^». Это существенно разгружает тексты программ и облегчает их восприятие.

3.2.2. Конструкторы

Как видно из выше приведенного примера, конструктор объекта — это *функция*, возвращающая указатель на объект. Вызов конструктора предваряется префиксом в виде имени класса, к которому принадлежит создаваемый объект. То есть, общий синтаксис вызова конструктора таков:

```
Объект:= ИмяКласса.ИмяКонструктора
```

В корневом классе **TObject** предусмотрен конструктор по имени **Create** (создать), он не требует параметров. Однако классы-наследники могут снабжаться разными конструкторами, имеющими различные имена и параметры, и даже несколькими конструкторами сразу, например:

```
type  TMyClass = class(TObject)
        m1: integer;
        constructor Create(aVal: integer);
        constructor InitOne;
        constructor InitTwo;
      end;

constructor TMyClass.Create(aVal: integer);
begin
  inherited Create;  // обязательный вызов унаследованного конструктора
  m1:= aVal;
end;

constructor TMyClass.InitOne;
begin
  inherited Create;  // обязательный вызов унаследованного конструктора
  m1:= 1;
end;

constructor TMyClass.InitTwo;
begin
  inherited Create;  // обязательный вызов унаследованного конструктора
  m1:= 2;
```

```
end;

var X0, X1, X2 : TMyClass;

begin
  X0:= TMyClass.Create(10);
  X1:= TMyClass.InitOne;
  X2:= TMyClass.InitTwo;
  ...
end.
```

Как показывает этот пример, внутри нового конструктора вначале обязательно вызывают унаследованный конструктор. Хотя конструктор является функцией, её тип в объявлении не указывают (тип очевиден), и возвращаемое значение в теле конструктора не определяют.

3.2.3. Инициализация полей

Назначение конструктора — создать объект и заполнить его поля (инициализировать). Корневой конструктор **Create** только очищает все поля объекта, — иногда это всё, что требуется. После очистки, поля объекта, в зависимости от их типа, принимают следующие значения:

Тип поля	Значение
Число	0
Булево	false
Символ	#0
Строка	Пустая
Указатель	nil

Таким образом, унаследованным классам не всегда нужен новый конструктор, он необходим лишь для инициализации полей, содержащих непустые значения.

3.2.4. Деструкторы

Назначение деструктора — освободить память, занимаемую объектом. Деструктор корневого класса **TObject** объявлен как виртуальный:

```
    destructor Destroy; virtual;
```

Его реализация пуста, сам по себе он ничего не делает, зато позволяет переопределять себя в наследниках. Переопределение требуется, когда поля наследника тоже являются объектами, и тогда деструктор прежде уничтожает эти поля, а уже затем вызывает унаследованный деструктор. Запомните: унаследованный конструктор всегда вызывают *первым*, а унаследованный деструктор — *последним*, например:

```
type  TMyClass = class(TObject)
        mObj: TObject;
        constructor Create;
        destructor Destroy; virtual;
```

```
      end;

constructor TMyClass.Create(aVal: integer);
begin
  inherited Create;        // унаследованный конструктор
  mObj:= TObject.Create;   // инициализация поля
end;

destructor TMyClass.Destroy;
begin
  mObj.Free;          // освобождение поля
  inherited Destroy; // унаследованный деструктор
end;

var X: TMyClass;

begin
  X:= TMyClass.Create;
  { работа с объектом }
  X.Free;  // неявный вызов деструктора Destroy
end.
```

Но почему в этом примере вызван не деструктор **Destroy**, а процедура **Free**? Посмотрим на реализацию метода **Free**:

```
procedure TObject.Free;
begin
  if Self <> nil then Destroy;
end;
```

Перед вызовом деструктора метод проверяет, не вызван ли он для несуществующего объекта (здесь **Self** — это указатель на данный объект).

3.2.5. Очистка переменных-классов

Хотя при удалении объекта деструктор и освобождает занимаемую им память, он не трогает сам указатель на объект (переменную), например:

```
var X: TObject;

begin
    Writeln('1= ', Assigned(X));   // false
    X:= TObject.Create;
    Writeln('2= ', Assigned(X));   // true
    X.Free;
    Writeln('3= ', Assigned(X));   // true
    X:=nil;
    Writeln('4= ', Assigned(X));   // false
    Readln;
end.
```

Когда надо отметить факт уничтожения объекта, освободившейся переменной принудительно присваивают пустое значение **NIL**.

3.3. Методы класса

Методы класса — ещё одно ценное средство **Delphi**. Они могут вызываться двояко: 1) как обычно — с указанием префикса-переменной, 2) указанием в качестве префикса имени типа (как при вызове конструктора). Иначе говоря, для вызова методов класса наличие объектов не обязательно. Методы, встроенные в корневой класс **TObject**, дают ряд полезных возможностей по отслеживанию типов переменных, а также их размеров на этапе исполнения программы. Рассмотрим некоторые из них, а именно:

```
TObject = class
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    ...
  end;
```

3.3.1. Определение имени класса

Метод **ClassName** возвращает строку с именем класса, совпадающую с именем в тексте программы с точностью до регистра букв. Для сравнения имени класса со строкой, или с именем другого класса лучше подходит функция **ClassNameIs**, игнорирующая регистр букв, например:

```
var T1, T2 : TObject;

begin
  Writeln(TObject.ClassName);              // 'TObject' - как в исходном коде
  Writeln(TObject.ClassName='TOBJECT');    // false (несовпадение регистра)
  Writeln(TObject.ClassNameIs('TOBJECT')); // true (регистр игнорируется)
  Writeln(TObject.ClassNameIs('tobject')); // true (регистр игнорируется)
  Writeln(TObject.ClassNameIs('obj'));     // false
  T1:= TObject.Create;
  T2:= TObject.Create;
  Writeln(T1.ClassNameIs(T2.ClassName));   // true
  Readln;
end.
```

3.3.2. Сравнение типов

Сравнивать типы объектов можно функциями **ClassType** и **InheritsFrom**. Первая из них возвращает тип **TClass**, который, по сути, является *указателем* на специальную структуру данных, связанную с классом: для переменных одного класса эти указатели совпадают.

Метод **InheritsFrom** действует тоньше, возвращая **true**, если данный класс, либо совпадает с классом-аргументом, либо унаследован от него, например:

```
type

  T1 = class(TObject)  // унаследован от корневого
    m1: integer;
  end;

  T2 = class(T1)  // унаследован от T1
    m2: integer;
  end;

var V0 : TObject;
    V1 : T1;
    V2 : T2;

begin
    V0:= TObject.Create;
    V1:= T1.Create;
    V2:= T2.Create;

    Writeln(V0.ClassType = TObject);  // true
    Writeln(V1.ClassType = TObject);  // false
    Writeln(V2.ClassType = TObject);  // false

    Writeln(V0.InheritsFrom(TObject));  // true
    Writeln(V1.InheritsFrom(TObject));  // true
    Writeln(V2.InheritsFrom(TObject));  // true

    Writeln(V0.InheritsFrom(T1));  // false
    Writeln(V1.InheritsFrom(T1));  // true
    Writeln(V2.InheritsFrom(T1));  // true

    Writeln(V0.InheritsFrom(T2));  // false
    Writeln(V1.InheritsFrom(T2));  // false
    Writeln(V2.InheritsFrom(T2));  // true
    Readln;
end.
```

3.3.3. Сравнение типов операцией is

На практике сравнивать типы переменных удобней основанной на функции **`InheritsFrom`** операцией **`is`**, например:

```
type

  T1 = class(TObject)
    m1: integer;
  end;

  T2 = class(T1)
    m2: integer;
  end;

var V0 : TObject;
    V1 : T1;
    V2 : T2;

begin
    V0:= TObject.Create;
    V1:= T1.Create;
```

```
    V2:= T2.Create;

    Writeln(V0 is TObject);  // true
    Writeln(V1 is TObject);  // true
    Writeln(V2 is TObject);  // true

    Writeln(V0 is T1);  // false
    Writeln(V1 is T1);  // true
    Writeln(V2 is T1);  // true

    Writeln(V0 is T2);  // false
    Writeln(V1 is T2);  // false
    Writeln(V2 is T2);  // true

    Writeln(V1 is V0.ClassType);  // true
    Writeln(V1 is V2.ClassType);  // false

    Readln;
end.
```

3.4. Копирование объектов

Оператор присваивания копирует не объекты, а указатели на них. При этом компилятор требует, чтобы справа от знака присваивания располагался либо объект того же класса, что и слева, либо его наследник, например:

```
  // определения см. в предыдущем примере
  v0:= v1;  // верно
  v1:= v0;  // ошибка
```

Это общее правило основано на полиморфизме в ООП, оно гарантирует, что в случае необходимости будет вызван метод того объекта, на который указывает переменная в текущий момент, будь то объект того же типа, что и переменная, либо любой его наследник. Это работает, когда вызываемый метод определён в родительском классе. Если же надо обратиться к новым, не существующим в предке полям и методам, прибегают к приведению типов.

3.5. Приведение типов операцией as

Приведение типа не влечёт преобразований данных. Оно лишь заставляет компилятор трактовать некую переменную не так, как она объявлена, а так, как надо программисту в данном операторе. При этом ответственность за возможную ошибку переходит от компилятора к программисту.

К примеру, некоторое число **N** можно трактовать как код символа: **Char(N)**. Точно так же можно было бы приводить и объектные переменные, трактуя предка, как потомка. Но если в первом случае мы ничем не рискуем, то в случае с объектами наша невнимательность может повлечь крах или необъяснимое поведение программы. Поэтому для объектных типов в **Delphi** предусмотрена особая конструкция приведения типов ключевым словом **as**. Рассмотрим следующий пример:

```
type

  T1 = class(TObject)   m1: integer;  end;
  T2 = class(T1)        m2: integer;  end;
  T3 = class(T2)        m3: integer;  end;

var V1 : T1;

begin
    V1:= T1.Create;   // "Родной" тип данных
    Writeln(V1.m1);   // Всё верно
    V1.Free;

    V1:= T2.Create;   // Тип потомка
    // Корректное приведение типа:
    Writeln(T2(V1).m2);          // Жёсткое приведение типа
    Writeln((V1 as T2).m2);      // Мягкое приведение типа
    // Некорректное приведение типа:
    Writeln(T3(V1).m3);          // Здесь ошибка приведения не обнаружена
    Writeln((V1 as T3).m3);      // Обнаружена ошибка EInvalidCast
end.
```

Здесь имеется иерархия из трёх классов: **T1 -> T2 -> T3**. Переменная **V1** имеет тип родителя, что даёт нам право присваивать ей указатели на потомков без приведения типов. Однако для обращения к новым полям потомка мы должны привести переменную к типу этого потомка. Традиционное «жёсткое» приведение типа бесконтрольно и может повлечь труднообъяснимые ошибки. «Мягкое» приведение операцией **as** порождает дополнительный проверяющий код, который генерирует исключение **EInvalidCast** в случае несоответствия типов при исполнении программы. Оно же побуждает проверить соответствие типов в ходе компиляции.

Альтернативой может быть следующий код:

```
    if V1 is T3
      then {безопасно приводим V1 к типу T3}
      else {сообщаем об ошибке};
```

3.6. Размер объекта

Иногда требуется знать объём памяти, занимаемый объектом. Однако традиционная псевдо-функция **SizeOf**, применённая к объектной переменной, вернёт размер указателя, а это не то, что нужно. Задачу решает функция класса **InstanceSize**, которая возвращает реальный объём памяти, занимаемый полями объекта. Для корневого «пустого» объекта **TObject** функция возвращает размер указателя, поскольку такой указатель на специальную структуру данных, описывающую класс, скрыт где-то в недрах корневого объекта. Для всех унаследованных классов функция возвращает сумму размеров полей плюс размер указателя. Для классов, определённых в предыдущем примере, будут выданы следующие результаты:

```
    // Выводится размер указателя
    Writeln(SizeOf(T1));  // 4
    Writeln(SizeOf(T2));  // 4
    Writeln(SizeOf(T3));  // 4

    // Выводится размер объекта вместе со скрытым полем
    Writeln(TObject.InstanceSize);  // 4
    Writeln(T1.InstanceSize);       // 8
    Writeln(T2.InstanceSize);       // 12
    Writeln(T3.InstanceSize);       // 16
```

3.7. Области видимости

Области видимости полей и методов частично скрывают внутренности объекта, что в целом повышает надёжность программ. Чтобы определить полю или методу область видимости, его объявление помещают в одну из четырёх секций класса, мы рассмотрим только три из них:

```
Type
  T = class(TObject)
    private
      // Приватная секция
      m1: integer;
      ...
    protected
      // Защищённая секция
      m2: integer;
      ...
    public
      // Общедоступная секция
      m3: integer;
      ...
  end;
```

Краткое описание областей видимости дано в табл. 3-1.

Табл. 3-1 — Описание областей видимости

Наименование секции	Ключевое слово	Описание
Приватная	private	Поля и методы доступны только в модуле, где описан объект.
Защищённая	protected	Поля и методы доступны за пределами модуля, но только из методов данного объекта и его наследников.
Общедоступная	public	Поля и методы доступны в любом месте программы.

Секции могут повторяться и чередоваться в любом порядке.

3.8. Свойства

Ещё одним ценным средством повышения надёжности программ являются *свойства* — это механизм доступа к полям через специально созданные для этого методы. Хотя реализация этих методов обычно несложна, дополнительный код может порой затруднить понимание программы. Это единственная причина того,

что в данной книге механизм свойств не используется, и обращение к полям объектов выполняется непосредственно.

3.9. Статические, динамические, виртуальные и абстрактные

Этими титулами награждаются *методы* объектов. Соответствующие ключевые слова и особенности методов представлены в табл. 3-2.

Табл. 3-2 — Разновидности методов

Метод	Ключевое слово	Описание метода
Статический	Нет	Реализуют действия, выполняемые одинаково во всех потомках данного класса. Никаких ключевых слов в объявлении метода не используется.
Виртуальный	virtual	Реализует разное исполнение однотипных действий в разных потомках класса. Объявление метода в родителе снабжается ключевым словом virtual: **procedure ABC; virtual;** Объявление метода в потомках снабжается ключевым словом override: **procedure ABC; virtual; override;**
Динамический	dynamic	То же, что и виртуальный метод, но таблица динамических методов построена компактней, что экономит память.
Абстрактный	abstract	Методы, объявленные в родителе как абстрактные, не нуждаются в реализации, но и вызваны быть не могут. Эти методы обязательно переопределяются в потомках. Пример объявления в родителе: **procedure ABC; virtual; abstract;** Пример объявления в потомках: **procedure ABC; virtual; override;**

Больше информации на эту тему найдёте в рекомендуемой литературе.

3.10. Прочие замечания

Теперь обратим ваше внимание на некоторые часто используемые в книге средства языка.

3.10.1. Локальные процедуры и функции

Сложные процедуры и функции обычно разбивают на ряд простых, что и предусмотрено в современных языках программирования. В Паскале и **`Delphi`**, в отличие, скажем, от языка Си, допускается вставлять вспомогательные процедуры и функции внутрь других процедур и функций (подобно матрёшкам), что делает программы более надёжными и обозримыми. Видимость локальных процедур и функций ограничена той процедурой или функцией, в которой она объявлена. Далее вы найдёте много подобных примеров.

3.10.2. Оператор whith

Оператор **whith** создаёт локальную область видимости для полей записи, либо для полей и методов некоторого класса, например:

```
type

  T0 = class(TObject)
    m1: integer;
    m2: integer;
    m3: integer;
  end;

var  V0 : T0;
begin
  ...
  // Вместо
  V0.m1:= 1;  V0.m2:= 2;  V0.m3:= 3;
  // Можно записать так:
  with V0 do begin
    m1:= 1;    m2:= 2;    m3:= 3;
  end;
  ...
end
```

Такие сокращения разгружают текст и облегчают понимание программы.

3.10.3. Квалификатор Self

Ключевое слово **Self** означает ссылку на текущий объект внутри методов класса, например:

```
function TNode.GenGammaOut: TSet;
begin
  ...
  // здесь текущий объект передаётся в качестве параметра в процедуру Insert
  Result.Insert(Self);
  ...
end;
```

3.10.4. Перегрузка операторов

Эта возможность языка состоит в том, что вызов некоторой функции заменяется одним из общепринятых математических знаков (+, -, * и т.д.). Так, к примеру, функцию вставки объекта в множество можно «перелицевать» в оператор сложения. Перегрузка операторов с одной стороны улучшает читабельность программы, а с другой — усложняет поиск ошибок, когда что-то идёт не так. И потому пока не сложилось единого мнения о пользе или вреде перегрузки операторов. В данной книге перегрузка не используется.

3.11. Итоги

- Объектные переменные **Delphi** отражают двойственную природу реальных дискретных объектов: с одной стороны они элементарны, а с другой — могут представлять собой сколь угодно сложные структуры данных.
- В языке **Delphi** чётко различаются два понятия: *класс* — это объектный тип, а *объект* — переменная этого типа.
- Все объектные переменные в **Delphi** являются указателями на динамические переменные. Помимо объявления, объекты требуется также создавать и уничтожать.
- Объектные переменные в **Delphi** разыменованы по умолчанию, стрелки «^» не требуются.
- При создании объекта конструктором все его поля автоматически очищаются.
- Для контроля и приведения типов объектных переменных используют методы корневого класса **TObject**, а также операции **as** и **is**.
- Реальный объём памяти, занимаемый объектом, возвращает метод класса **InstanceSize**.
- Ограничение областей видимости полей и методов, а также применение свойств снижает риск непреднамеренного искажения данных и повышает надёжность программ.

3.12. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
√	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
√	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 4
Базовый объект

Начинаем воплощать нашу мечту — создавать универсальный объект, сочетающий, казалось бы, не сочетаемое: с одной стороны это будет элементарный объект, а с другой — он может представлять сколь угодно сложную структуру данных. Современные объектные технологии дают нам такую возможность.

Разумеется, что одним объектным классом нам не обойтись: для решения последующих задач потребуется иерархия весьма сложных объектов. Забегая немного вперёд, обратимся к рис. 4-1, где представлена эта иерархия. Все будущие классы будут порождены от базового класса **TItem** — *элемент*. Это значит, что такие сложные структуры, как очереди, стеки, множества, графы и прочие будут обладать свойствами элементарного объекта. При необходимости мы сможем, к примеру, создать множества, элементами которых будут другие множества, и помещать их в очередь или стек. Или создавать множества, состоящие из графов, очередей, или создавать очереди очередей и т.д. Короче, любые ограничения на сложность объектов отныне не существуют.

4.1. Базовый класс

Поскольку классу **TItem** уготована роль предка всех прочих классов, в ходе проектирования важно определить наиболее общие методы, свойственные всем его потомкам. Практика показала, что в объектах-потомках будут востребованы три базовые возможности:

- возможность выводить информацию о себе в текстовый файл и на экран;
- возможность сравнивать себя с другим объектом;
- возможность порождать свою копию.

Отображать объект в текстовой форме потребуется и для вывода результатов, и при отладке. Возможность сравнивать объекты нужна при вставке их в множества, ведь дубликаты в множествах запрещены, а упорядочение приветствуется. Необходимость копирования объясняется тем, что оператор присваивания копирует только указатель на объект, но не сам объект, а это порой требуется. Всё это приводит к следующему объявлению базового класса (см. модуль **Root** в приложении B):

```
TItem = class (TObject)
  public
    function Compare(arg: TItem): TCompare; virtual;  // сравнение
    function Copy: TItem;  virtual; abstract;         // копирование
    procedure Print(var aFile: TextFile); virtual; abstract; // вывод в файл
    procedure Expo;  // вывод на экран
end;
```

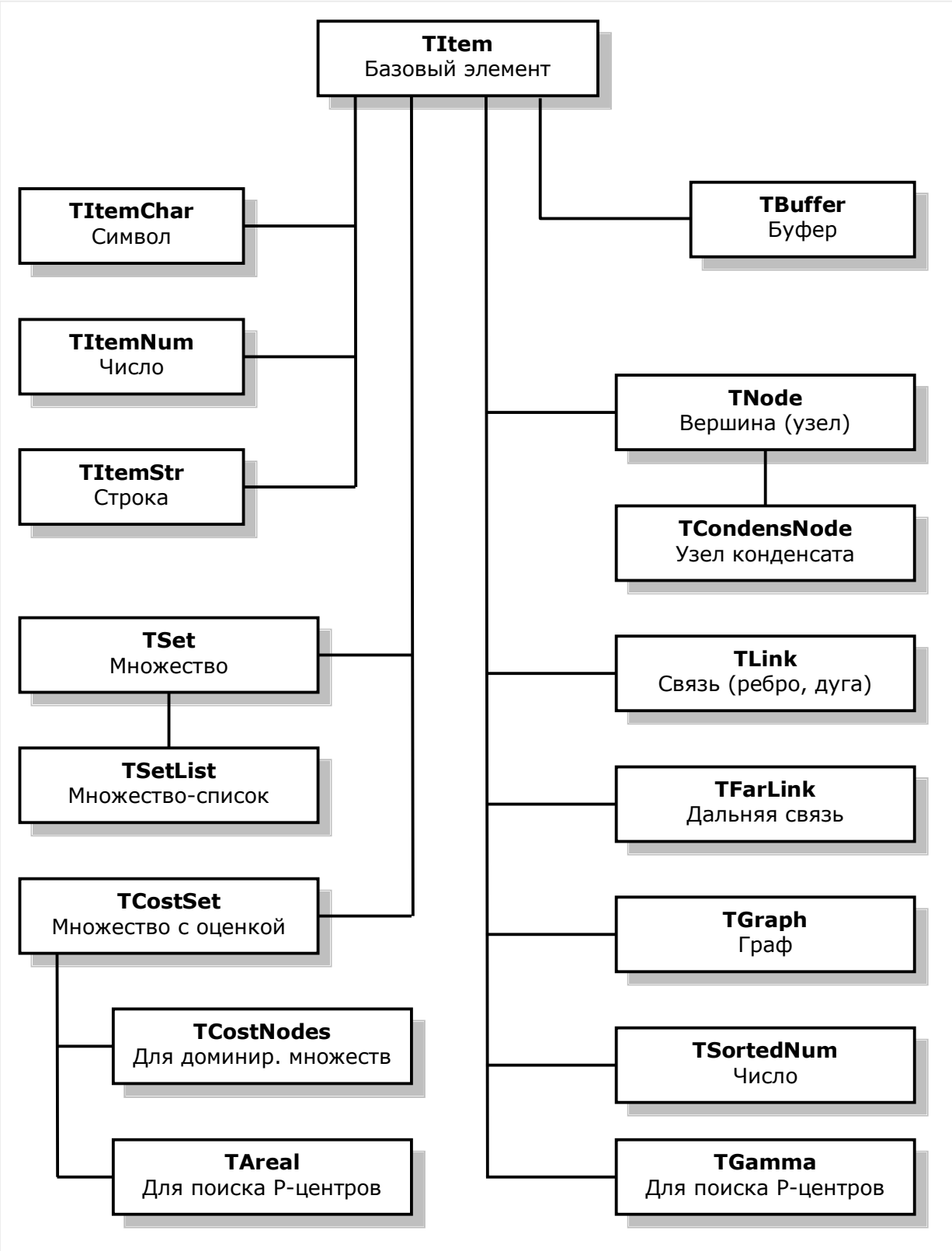


Рис. 4-1 — Схема наследования основных классов

4.1.1. Вывод в текстовой форме

Вывод в текстовый файл выполняется методом **Print**, принимающим один параметр — открытый текстовый файл. Кодировать метод в базовом классе не имеет смысла, потому он объявлен абстрактным.

Однако метод **Expo**, предназначенный для вывода объекта на экран, может быть реализован сразу:

```
procedure TItem.Expo;        // Метод отображения на экране
begin
  Print(Output);    // Вывод на экран
end;
```

Метод **Expo** объявлен статическим, а это значит, что в будущем он переопределяться не будет. Кажется странным, что он вызывает нереализованный пока метод **Print**, и компилятор не возражает против этого. Но таков общий подход в ООП: если предку нужен какой-то метод, реализуемый только лишь в потомках, в предке объявляем его абстрактным.

4.1.2. Сравнение объектов

Сравнивать объекты придётся очень часто, особенно работая с множествами. Как правило, интересующие нас объекты не являются числами, однако нам придётся их сравнивать на равенство/неравенство и на больше/меньше. С этой целью в модуле **Root** объявлено перечисление:

```
// Возможные результаты сравнения объектов

  TCompare = (cmpEq,      // объекты совпадают или эквивалентны
              cmpLess,    // меньше
              cmpGreate,  // больше
              cmpIncomp   // несравнимы (Incomparable)
             );
```

Метод **Compare** будет возвращать одно из этих значений. В корневом предке он реализован так:

```
function TItem.Compare(arg: TItem): TCompare;
begin
  if arg = Self
    then Result:= cmpEq        // совпадают
    else Result:= cmpIncomp;   // несравнимы
end;
```

В будущих потомках он будет, как правило, переопределяться.

4.1.3. Копирование объекта

Метод копирования **Copy** объявлен виртуальным и абстрактным, т.е. будет реализован в потомках.

4.2. Демонстрационные классы

Скромные возможности базового класса очевидны: ведь он не содержит никакой информации, и потому сам по себе ни на что не годен. Зато служит фундаментом для многих других классов, — мы продемонстрируем это на примере трёх реальных классов: *символ*, *число* и *строка* (см. модуль **Items** в Приложении С).

Класс **TItemChar** содержит одно информационное поле — символ. В нём уже реализованы методы, которые в предке были абстрактными.

Листинг 4-1 — Объявление и реализация класса *символ*

```
TItemChar = class (TItem)
    private
      mData: Char;
    public
      constructor Create(arg: Char);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: Char;
  end;

implementation

constructor TItemChar.Create(arg: Char);
begin
  inherited Create;
  mData:= arg;
end;

function TItemChar.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemChar) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
  if (arg as TItemChar).mData = mData
    then Result:=cmpEq
    else if (arg as TItemChar).mData > mData
            then Result:= cmpLess
            else Result:= cmpGreate;
end;

function TItemChar.Copy: TItem;
begin
  Result:= TItemChar.Create(mData);
```

```
end;

procedure TItemChar.Print(var aFile: TextFile);
begin
  Write(aFile, mData:2)
end;

function TItemChar.GetData: Char;
begin
  Result:= mData
end;
```

Объекты **TItemNum** (число) и **TItemStr** (строка) реализованы аналогично, за подробностями обращайтесь к модулю **Items**. Ниже показано применение этих трёх объектов.

Листинг 4-2 — Демонстрация полиморфизма на примере трёх объектов

```
uses
  Root in '..\Common\Root.pas',
  Items in '..\Common\Items.pas';

var  Buf : array [1..3] of TItem;
       i : integer;
begin
    Buf[1]:= TItemChar.Create('A');      // символ
    Buf[2]:= TItemNum.Create(10);        // число
    Buf[3]:= TItemStr.Create('Class');   // строка
    for i:= 1 to 3 do begin
      Buf[i].Expo;
      Writeln;
      Buf[i].Free;
    end;
    Readln;
end.
```

Хотя элементы массива **Buf** относятся к типу **TItem**, программа помещает туда разнотипные элементы-потомки. Обычные массивы этого «не любят», но здесь всё в порядке, — сказывается полиморфизм ООП.

4.3. Итоги

- Объектные технологии позволяют отразить противоречивую сущность дискретных объектов: с одной стороны, такие объекты могут обладать внутренней структурой, с другой — рассматриваются как элементарные, неделимые сущности.
- Вся гамма дискретных объектов, используемых для решения последующих задач, представляет иерархию объектов, происходящих от корневого объекта класса **TItem** — «элемент».
- Корневой прародитель всех элементов обладает только методами, свойственными всем его потомкам. Эти методы обеспечивают:
    - вывод объекта в файл и на экран (**Print**, **Expo**);
    - сравнение элементов (**Compare**);
    - создание копии элемента (**Copy**).

4.4. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
√	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
√	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 5
Универсальный буфер

В этой главе разработаем первый полезный объект, необходимый в последующих проектах — универсальный буфер.

5.1. Требования

Для размещения данных в оперативной памяти язык **Delphi** и сопутствующие библиотеки предлагают богатый арсенал средств: статические и динамические массивы, коллекции, списки. Но мы разработаем своё собственное вместилище элементов, обладающее нужными нам свойствами и возможностями, а именно:

- вместимость, ограниченную лишь объёмом памяти;
- возможность доступа к элементам по индексу, подобно массиву;
- возможность последовательного перебора элементов — сканирование;
- вставку и извлечение элементов по принципу очереди;
- вставку и извлечение элементов по принципу стека;
- реверс элементов в буфере — перестановку их в обратном порядке;
- очистку буфера с уничтожением и без уничтожения элементов.

Класс универсального буфера произведём от базового класса **TItem**, следовательно, объект-буфер будет обладать всеми его возможностями.

5.2. Объявление

Соответствующее нашим намерениям объявление класса **TBuffer** представлено ниже (см. модуль **Items** в Приложении С).

```
TBuffer = class (TItem)
    private
      mCount: integer;         // счётчик элементов
      mHead, mQue: PStackRec; // указатели на начало и конец буфера
      mCurrent : PStackRec;   // указатель на текущий элемент при переборе
      procedure Clr(aDestroy: boolean);
    public
      destructor Destroy;  override;
      function Copy: TItem; override; // создание копии
      procedure Put(arg: TItem);      // занесение в очередь
      function Get: TItem;            // извлечение из очереди
      procedure Push(arg: TItem);     // занесение в стек
      function Pop: TItem;            // извлечение из стека
      function Top: TItem;            // Возвращает элемент на вершине стека
      procedure Reversion;            // реверс буфера
      function GetByIndex(aIndex: integer): TItem; // доступ по индексу
      function GetCount: integer;     // количество элементов в буфере
      procedure Clear;                // очистка без уничтожения объектов
      procedure ClrAndDestroy;        // очистка с уничтожением объектов
      procedure Print(var aFile: TextFile); override; // вывод в файл
```

```
      // Последовательный перебор элементов (сканирование)
      function GetFirst: TItem;
      function GetNext: TItem;
      // Поиск (проверка наличия) элемента в буфере
      function IsPresent(arg : TItem): boolean;
  end;
```

5.3. Реализация

Заявленный объект-буфер можно реализовать разными средствами. Здесь воспользуемся односвязным списком, чем достигнем поставленных целей относительно просто. Класс **TBuffer** размещён в модуле **Items**, листинг с основными его методами представлен ниже.

Листинг 5-1 — TBuffer -- класс для буфера, очереди и стека

```
// Вспомогательная структура для организации односвязного списка

PStackRec = ^TStackRec;
  TStackRec = record
    mItem : TItem;            // элемент списка
    mNext : PStackRec;        // ссылка на следующий
  end;

// Создание копии буфера из существующего

function TBuffer.Copy: TItem;
var t: TItem;
    i: integer;
begin
  Result:= TBuffer.Create;
  for i:= 1 to GetCount do begin
    t:= Get;
    Put(t);
    TBuffer(Result).Put(t);
  end;
end;

// Уничтожение буфера без уничтожения элементов

destructor TBuffer.Destroy;
begin
  Clear;
  inherited;
end;

// Очистка буфера

procedure TBuffer.Clr(aDestroy: boolean);
var i : integer;
    t : TItem;
begin
  for i:= 1 to mCount do begin
    t:= Get;
    if aDestroy then t.Free
  end;
end;

// Очистка с уничтожением элементов
```

```
procedure TBuffer.ClrAndDestroy;
begin
  Clr(true)
end;

// Очистка без уничтожения

procedure TBuffer.Clear;
begin
  Clr(false)
end;

// Получение счётчика элементов

function TBuffer.GetCount: integer;
begin
  Result:= mCount;
end;

// Push - помещение в стек (в начало буфера)

procedure TBuffer.Push(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= mHead;
  mHead:= r;
  Inc(mCount);
  // Обновляем указатель на конец списка
  if not Assigned(mQue) then mQue:= r;
end;

// Put - помещение в очередь (в конец буфера)

procedure TBuffer.Put(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= nil;
  // Присоединяем к концу списка
  if Assigned(mQue) then mQue^.mNext:= r;
  mQue:= r;
  Inc(mCount);
  // Обновляем указатель на начало
  if not Assigned(mHead) then mHead:= r;
end;

// Извлечение из очереди и стека (из начала списка )

function TBuffer.Get: TItem;
var r: PStackRec;
begin
  Result:= nil;
  if not Assigned(mHead) then Exit;
  r:= mHead;
  Result:= mHead^.mItem;
  mHead:= mHead^.mNext;
  Dispose(r);
  Dec(mCount);
```

```
  // Обновляем указатель на хвост
  if mCount=0 then mQue:= nil;
end;

// Извлечение из стека (синоним Get)

function TBuffer.Pop: TItem;
begin
  Result:= Get;
end;

// Доступ к очереди по индексу

function TBuffer.GetByIndex(aIndex: integer): TItem;
var  r: PStackRec;
begin
  Result:= nil;
  if (aIndex<1) or (aIndex>mCount) then Exit;
  r:= mHead;
  while Assigned(r) do begin
    Dec(aIndex);
    if aIndex=0 then Break;
    r:= r^.mNext;
  end;
  if Assigned(r) then Result:= r^.mItem;
end;

// Реверс буфера (перестановка в обратном порядке)

procedure TBuffer.Reversion;
var B : TBuffer;
    t : TItem;
begin
  B:= TBuffer.Create;
  t:= Get;
  while Assigned(t) do begin   B.Push(t);  t:= Get;  end;
  t:= B.Get;
  while Assigned(t) do begin   Put(t);   t:= B.Get;  end;
  B.Free;
end;

// Печать буфера

procedure TBuffer.Print(var aFile: TextFile);
var r: PStackRec;
begin
  Writeln(aFile, '(');
  r:= mHead;
  while Assigned(r) do begin
    r^.mItem.Print(aFile);
    r:= r^.mNext;
  end;
  Writeln(aFile, ') BufCount= ', mCount);
end;
```

5.4. Демонстрация

Следующая программа демонстрирует возможности универсального буфера.

Листинг 5-2 — Программа для демонтрации возможностей буфера

```
{$APPTYPE CONSOLE}
uses
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas';

var Buf : TBuffer;
      T : TItem;
begin
  Buf:= TBuffer.Create;  // создание буфера

  // Создание и вставка элементов в конец буфера:
  // символы
  Buf.Put(TItemChar.Create('A'));
  Buf.Put(TItemChar.Create('B'));
  Buf.Put(TItemChar.Create('C'));
  // числа
  Buf.Put(TItemNum.Create(10));
  Buf.Put(TItemNum.Create(20));
  Buf.Put(TItemNum.Create(30));
  // Вывод на экран:
  Buf.Expo; Writeln;

  // Доступ по индексу:

  T:= Buf.GetByIndex(1);  // 1-й элемент
  T.Expo;
  T:= Buf.GetByIndex(4);  // 4-й элемент
  T.Expo;
  Writeln;

  // Реверс:
  Buf.Reversion;       // 1-й реверс
  Buf.Expo; Writeln;   // вывод
  Buf.Reversion;       // 2-й реверс
  Buf.Expo; Writeln;   // вывод

  // Круговой сдвиг на три позиции:
  // (выбор с начала и вставка в конец)
  Buf.Put(Buf.Get);
  Buf.Put(Buf.Get);
  Buf.Put(Buf.Get);
  Buf.Expo; Writeln;   // вывод

  // Извлечение из стека, вывод и уничтожение объектов

  while Buf.GetCount<>0 do begin
    T:= Buf.Pop;
    T.Expo;
    T.Free;
  end;
  Writeln;

  // Очистка буфера с уничтожением объектов:
  Buf.ClrAndDestroy;
  Buf.Free;
  Readln;
end.
```

5.5. Итоги

- Универсальный буфер — это *элемент*, предназначенный для хранения других элементов (в том числе других буферов).
- Универсальный буфер обладает возможностями безразмерного динамического массива, очереди и стека.

Глава 6
Представление множеств

Теперь приступаем к постройке *объекта-множества.* Множество — это инструмент классификации наших представлений об окружающем мире — лежит в основе современной математики. Объекты-множества потребуются для решения последующих задач на *графах*.

6.1. Множества в Паскале

Стандартный тип *множество* (**set**) встроен в язык Паскаль, для него предусмотрены базовые операции: объединение, вычитание, пересечение, сравнение. Но возможности этих множеств весьма ограничены, а именно:

- элементами множеств в Паскале могут быть только простые типы данных: символы и числа;
- максимальное количество элементов (мощность множества) невелико и обычно ограничено 256 элементами.

Для решения последующих задач нужен такой тип множества (класс), где эти оковы исчезнут. Методы создаваемого нами класса обеспечат наряду со стандартными операциями ряд вспомогательных действий. Ёмкость множества будет ограничена лишь объёмом памяти, а элементами множеств будут любые объекты, включая сами множества.

6.2. Объявление абстрактного множества TSet

Возможности создаваемого класса видны из его объявления, где перечислены все операции над множествами. Класс *множество* будет произведён от класса *элемент*, а это значит, что сами множества можно трактовать и как элементарные объекты. Вот объявление класса:

```
// Класс "абстрактное множество"

  TSet = class (TItem)
    protected
      mCount: Longint;  // количество элементов множества
      procedure Clr(aDestroy: boolean); virtual; abstract;
    public
      // Базовые операции с множествами:
      procedure Add(arg : TSet); virtual; abstract;  // объединение
      procedure Sub(arg : TSet); virtual; abstract;  // вычитание
      procedure Mul(arg : TSet); virtual; abstract;  // пересечение
      procedure ExOr(arg : TSet); virtual; abstract; // исключающее ИЛИ
      function  TestIntersect(arg: TSet): boolean;   // проверка пересечения
      function  CompareSet(arg: TSet): TCompare;
      function  Compare(arg: TItem): TCompare; override;
      function  Exist(arg : TItem): boolean; virtual; abstract;
      // Очистка
      procedure Clear;          // очистка без уничтожения элементов
      procedure ClrAndDestroy;  // очистка с уничтожением элементов
```

```
      // Вставка, удаление, копирование элементоа
      function  Insert(arg: TItem): boolean; virtual; abstract;
      procedure Delete(arg: TItem); virtual; abstract;
      procedure CopyItems(arg: TSet);
      // Последовательный перебор элементов
      function  GetFirst: TItem; virtual; abstract; // выбрать первый
      function  GetNext: TItem; virtual; abstract;  // выбрать следующий
      // Выбор по индексу и по элементу
      function  GetItem(index: integer): TItem;
      function  GetObject(aItem: TItem): TItem;
      // Сохранение-восстановление позиции перебора
      procedure PositionPush;  virtual; abstract;
      procedure PositionPop;  virtual; abstract;
      // Прочие операции
      procedure CoverToDissect; // Преобразование покрытия в разбиение
      function  GetCount: integer;
      function  Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      destructor Destroy;  override;
  end;
```

Многие методы объявлены абстрактными, а это значит, что в базовом классе **TSet** они не будут реализованы, что не запрещает нам предварительно ознакомиться с ними.

6.2.1. Основные операции

Основные операции с множествами представлены в табл. 6-1, где для сравнения даны эквивалентные конструкции, записанные на Паскале.

Табл. 6-1 — Основные операции с множествами

Наименование	Эквивалент в Паскале	Вызов метода класса
Объединение	A:= A + B	A.Add (B)
Вычитание	A:= A - B	A.Sub (B)
Пересечение	A:= A * B	A.Mul (B)
Исключающее ИЛИ	A:= A + B – A * B	A.ExOr (B)
Проверка пересечения	A * B <> [ ]	A.TestIntersect (B)
Сравнение	A = B A < B A > B	A.CompareSet (B)
Проверка элемента	x in A	A.Exist (x)
Вставка элемента	A:= A + [x]	A.Insert (x)
Удаление элемента	A:= A – [x]	A.Delete (x)
Копирование всех элементов множества	A:= B	A.CopyItems (B)
Очистка множества	A:= [ ]	A.Clear A.ClrAndDestroy

Очищать множество можно двояко: либо с уничтожением, либо без уничтожения элементов множества. Деструктор множества по умолчанию очищает его без уничтожения включенных в него элементов.

6.2.2. Перебор элементов

В классическом множестве не существенен порядок его элементов. Однако на практике такой порядок обычно существует, как и потребность в переборе элементов, и в доступе к отдельным элементам множества по индексу (как в массиве). Представленные ниже методы класса дают эти возможности.

Табл. 6-2 — Методы доступа к элементам множества TSet

Метод класса	Описание
A.GetCount	Возвращает количество элементов множества.
A.GetItem(i)	Возвращает i-й элемент множества, где i = 1...GetCount. Если индекс выходит за допустимые пределы, возвращается пустое значение NIL.
A.GetFirst	Устанавливает позицию внутреннего указателя на первый элемент множества и возвращает первый элемент, либо NIL для пустого множества.
A.GetNext	Продвигает позицию внутреннего указателя на следующий элемент множества и возвращает следующий элемент, либо NIL, если следующий элемент не существует.
A.PositionPush	Сохраняет текущее значение внутренней позиции чтения.
A.PositionPop	Восстанавливает предыдущее значение внутренней позиции чтения.
A.GetItem(x)	Возвращает указатель на элемент множества, совпадающий с объектом x.

Первыми двумя методами можно перебирать элементы множества по схеме, характерной для массива:

```
for i:=1 to A.GetCount do begin
  x:= A.GetItem(i);
  { обработать элемент x }
end;
```

На первый взгляд кажется, что этого достаточно на все случаи, но мы не касались ещё реализации множеств... Забегая вперёд, отмечу, что в нашей реализации выбор i–го элемента влечёт перебор по цепочке предыдущих i-1 элементов, а это накладно. Проблему решают два метода, применяемые по такой схеме:

```
x:= A.GetFirst;                // взять первый элемент
while Assigned(x) do begin
  { обработать элемент x }
  x:= A.GetNext;               // взять следующий элемент
end;
```

Методы **GetFirst** и **GetNext** используют внутренний указатель на текущий элемент. Но тут возникает другая проблема: как быть с вложенными переборами? Её решают методы сохранения и восстановления текущей позиции чтения:

```
X:= A.GetFirst;     // взять первый
while Assigned(x) do begin
  A.PositionPush;  // сохранить позицию чтения
  { обработать элемент X, при этом позиция чтения может нарушиться }
  A.PositionPop;    // восстановить позицию чтения
  x:= A.GetNext;    // взять следующий
end;
```

Или в других ситуациях так:

```
A.PositionPush;     // сохранить позицию чтения
X:= A.GetFirst;     // взять первый
while Assigned(x) do begin
  { обработать элемент X }
  X:= A.GetNext;    // взять следующий
end;
A.PositionPop;      // восстановить позицию чтения
```

Наконец метод **GetItem(X)** выбирает из множества элемент, эквивалентный элементу **X** (в смысле сравнения методом **Compare**). Если элемент **X** уже является элементом массива, то возвращается сам элемент **X.** Если эквивалентного элемента в множестве нет, возвращается **NIL**.

6.2.3. Прочие методы

Прочие методы реализуют методы, унаследованные от базового класса *элемент* (такие, как **Copy**, **Print**, **Destroy**). Метод **CoverToDissect** будет рассмотрен позже.

6.3. Реализация класса TSet

В объявлении класса **TSet** многие методы объявлены абстрактными, поскольку ещё не определена структура данных, на которой будет построен этот класс. Возможны несколько таких структур, каждая из которых обладает своими преимуществами и недостатками: динамический массив, дерево, список. Лучшую из них в конкретной задаче может выявить только эксперимент. Хотя объекты абстрактного класса **TSet** никогда не будут созданы и применены, роль этого класса — быть родителем реальных классов-множеств. Эта абстракция даёт свободу тем, кто захочет реализовать множества как-то иначе, а не так, как сделано в этой книге. И все эти возможные реализации будут годны для решения последующих задач.

И всё же, не смотря на абстрактность класса **TSet**, часть его методов реализована уже здесь. Этим мы упрощаем работу создателям классов-потомков, поскольку, чем больше методов реализовано в предке, тем меньше забот с потомками.

Методы, реализованные в классе **TSet**, объявлены статическими, ознакомиться с ними можно по листингу модуля **Root** (Приложение B). В качестве примера ниже дана реализация метода сравнения множеств:

```
// Сравнение двух множеств
// Возвращаемые значения:
// cmpEq      - множества совпадают
// cmpLess    - первое является подмножеством второго
// cmpGreate  - второе является подмножеством первого
// cmpIncomp  - множества не поглощают друг друга

function TSet.Compare(arg: TItem): TCompare;
var p, q : TItem;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // один и тот же объект
  // Если аргумент не множество, то сравнивать нельзя
  if not (arg is TSet) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;

  // Сравнение элементов множеств
  PositionPush;                // сохр. позицию в данном множестве
  (arg as TSet).PositionPush;  // сохр. позицию в аргументе
  p:= GetFirst;                // взять первый элемент в данном множестве
  q:= (arg as TSet).GetFirst;  // взять первый элемент в аргументе
  while Assigned(p) and Assigned(q) do begin  // пока можно сравнивать
    Result:= p.Compare(q);                    // сравнение элементов
    if Result <> cmpEq then Break;            // выход, если не одинаковы
    p:= GetNext;  q:=(arg as TSet).GetNext;   // следующая пара
  end;
  if Result=cmpEq then begin                // если последние эл. совпали
    if GetCount < (arg as TSet).GetCount
      then Result:= cmpLess                 // данный меньше аргумента
    else if GetCount > TSet(arg).GetCount
      then Result:= cmpGreate               // данный больше аргумента
  end;
  (arg as TSet).PositionPop;  // восстановить позицию в аргументе
  PositionPop;                // восстановить позицию в данном множестве
end;
```

6.4. Класс TSetList

Напомню, что постройка абстрактного класса **TSet** преследовала две цели:

- объявить все методы, свойственные множествам, не связывая их с конкретной реализацией;
- реализовать некоторые методы в базовом классе для облегчения последующей разработки потомков.

Теперь можно сконструировать потомка (одного из возможных), в котором будут реализованы все методы множества. Для хранения элементов множества воспользуемся относительно простой структурой — *ОДНОСВЯЗНЫМ СПИСКОМ*. Возможно, что это решение не оптимально, но в большинстве случаев оно вполне приемлемо.

В объявление класса-потомка поместим нужные для списка поля, а все абстрактные методы класса-предка переопределим:

```
// Структура для организации односвязного списка

  PListRec = ^TListRec;
  TListRec = record
    mItem : TItem;
    mNext : PListRec;
  end;

// Множество, реализованное списком

  TSetList = class (TSet)
    protected
      mHead: PListRec;    // голова списка
      procedure Clr(arg: boolean); override;
    private
      mCurrent : PListRec; // указатель на текущий элемент
      mStack : TBuffer;  // стек для хранения позиций чтения mCurrent
      procedure Del(var p, q : PListRec);
      procedure Ins(p: PListRec; var q: PListRec; arg: TItem);
    public
      // переопределённые методы предка
      destructor Destroy;  override;
      function Exist(arg : TItem): boolean; override;
      function Insert(arg: TItem): boolean; override;
      procedure Delete(arg: TItem); override;
      function GetFirst: TItem; override;
      function GetNext: TItem; override;
      procedure PositionPush;  override;
      procedure PositionPop;   override;
      procedure Add(arg : TSet); override;
      procedure Sub(arg : TSet); override;
      procedure Mul(arg : TSet); override;
      procedure ExOr(arg : TSet); override;
  end;
```

Надо отметить, что многие переопределённые здесь методы можно было бы тоже объявить статическими и реализовать в родительском классе. Но такое решение не учло бы тонкостей работы с односвязным списком и повлекло бы заметное снижение быстродействия.

Для ознакомления с переопределёнными методами отсылаю вас к листингу модуля **SetList**, а в качестве примера привожу метод объединения множеств:

```
procedure TSetList.Add(arg: TSet);
var p, q : PListRec;
    t : TItem;
    compare : TCompare;
begin
  if not Assigned(arg) then Exit;
  if not (arg is TSet) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    compare:= cmpEq;
    while Assigned(p) do begin
      compare:= p^.mItem.Compare(t);
      case compare of
        cmpLess  : begin q:= p; p:= p^.mNext end;
```

```
        cmpEq     : begin t:= arg.GetNext; break end;
        cmpGreate,
        cmpIncomp: Break;
      end;
    end;
    if not Assigned(p) or (compare = cmpGreate) then begin
      Ins(p,q,t);
      if Assigned(t) then t:= arg.GetNext;
    end;
  end;
  arg.PositionPop;
end;
```

В следующей главе будут продемонстрирована работа с классами **TSet** и **TSetList**.

6.5. Итоги

- Встроенный в язык Паскаль тип *множество* ограничен типами хранимых элементов — символами и числами, а также количеством таких элементов.
- В абстрактном классе **TSet** объявлены все присущие ему операции. Абстрактное множество может содержать любые объекты, ёмкость его ограничена лишь объёмом доступной памяти.
- В классе **TSet** не определён механизм хранения элементов, поэтому часть его методов объявлена абстрактными. Объекты класса **TSet** никогда не создаются.
- В классе-потомке **TSetList** определён механизм хранения элементов — это односвязный список. В потомке переопределены и реализованы основные методы обработки множеств наиболее эффективным для выбранной структуры способом.

Глава 7
Операции с множествами

В предстоящих задачах будут интенсивно применяться множества, и потому здесь опишем основные операции над ними, уделив внимание ряду важных технических деталей.

7.1. Множество символов

Начнём с примера (листинг 7-1), где элементами множеств будут объекты-символы (не элементарные символы, как в стандартном Паскале). Описание класса **TItemChar** дано в модуле **Items**, далее следуют подробные комментарии к листингу.

Листинг 7-1 — Действия с множеством символов

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Root in '..\Common\Root.pas',
  Items in '..\Common\Items.pas';

// Создание множества символов на основе строки

function MakeSet(const arg: string): TSet;
var i: integer;
    t: TItem;
begin
  Result:= CreateSet;   // создание пустого множества
  for i:= 1 to Length(arg) do begin
    t:= TItemChar.Create(UpCase(arg[i]));  // создаём объект-символ
    if not Result.Insert(t)  // если не удалось вставить в множество
      then t.Free;           // то удаляем дубликат
  end;
end;

var A, B, C : TSet;    // множества объявлены абстрактными

begin
    // Создаём три множества:
    A:= MakeSet('Pascal');  // A = { A,C,L,P,S }
    B:= MakeSet('Delphi');  // B = { D,E,H,I,L,P }
    C:= CreateSet;          // C = { }
    Write('      A = '); A.Expo;
    Write('      B = '); B.Expo;
    // Объединение:
    Write('    A+B = ');
    C.CopyItems(A);
    C.Add(B);
    C.Expo;                { A,C,D,E,H,I,L,P,S }
    // Разность:
    Write('    A-B = ');
    C.CopyItems(A);
    C.Sub(B);
    C.Expo;                { A,C,S }
    // Пересечение:
    Write('    A*B = ');
```

```
    C.CopyItems(A);
    C.Mul(B);
    C.Expo;                 { L,P }
    // Исключающее ИЛИ:
    Write('A xor B = ');
    C.CopyItems(A);
    C.ExOr(B);
    C.Expo;                 { A,C,D,E,H,I,S }

    // Освобождение множеств:
    C.Free;            // здесь элементы не уничтожаем
    B.ClrAndDestroy;   // предварительно уничтожаем элементы
    B.Free;            // затем множество
    A.ClrAndDestroy;   // предварительно уничтожаем элементы
    A.Free;            // затем множество
    Readln;
end.
```

Итак, в начале программы видим функцию, создающую множество символов, выбираемых из параметра **arg**:

```
function MakeSet(const arg: string): TSet;
```

Обратите внимание, что типом результата является абстрактный класс **TSet**. *Важно:* при объявлении функций, переменных и параметров всегда будет использоваться именно абстрактный класс **TSet**. Это избавит программы от привязки к конкретной реализации класса-множества. Первый оператор в теле функции создаёт пустое множество, которое будет возвращено в качестве результата:

```
  Result:= CreateSet;    // создание пустого множества
```

Откуда взялась функция **CreateSet**? Она определена в модуле **Root** следующим образом:

```
function CreateSet: TSet;
begin
  Result:= TSetList.Create;
end;
```

Здесь кроется маленькая хитрость: тело этой функции — единственное место, где вызывается конструктор потомка, реализующего класс-множество (в данном случае — **TSetList**). Если в будущем множество будет реализовано как-то иначе, достаточно будет исправить в модуле **Root** только одну эту строку.

Вернёмся к функции **MakeSet**, где в цикле создаются элементы-символы **t** и вставляются в множество-результат. Обратите внимание на условный оператор:

```
    if not Result.Insert(t)   // если не удалось вставить в множество
      then t.Free;            // то удаляем дубликат
```

Если при вставке в множество там окажется дубликат (копия вставляемого элемента), то вставка не случится, и тогда надо удалить ненужный объект-дубликат.

Объявление следующих далее трёх глобальных переменных по указанным выше причинам (для отвязки от реализации класса) опять содержит абстрактный класс **TSet**.

```
var A, B, C : TSet;     // множества объявлены абстрактными
```

В теле главной программы демонстрируются основные операции, которые выполняются с множеством **C**, куда предварительно копируются элементы из множества **A**. Обратите внимание, что метод **CopyItems** копирует указатели на объекты из другого множества, но не создаёт копий объектов. То же касается и метода **Add**, добавляющего элементы из другого множества.

Последнее важное замечание относится к удалению объектов-множеств.

```
    C.Free;            // здесь элементы не уничтожаем
    B.ClrAndDestroy;   // предварительно уничтожаем элементы
    B.Free;            // затем множество
    A.ClrAndDestroy;   // предварительно уничтожаем элементы
    A.Free;            // затем множество
```

При уничтожении множества **C** нет нужды уничтожать содержащиеся в нём объекты, поскольку там хранятся лишь копии указателей на них. Что касается множеств **A** и **B**, то здесь методом **ClrAndDestroy** следует предварительно уничтожить сами объекты-символы, иначе они останутся в памяти, засоряя её вплоть до завершения программы.

7.2. Множество строк

Следующий пример демонстрирует те же операции, выполняемые с множеством объектов-строк — слов, извлечённых из текстовой строки.

Листинг 7-2 — Действия с множеством слов (строк)

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas';

// Создание множества слов, извлечённых из строки:

function MakeSet(const arg: string): TSet;
var i: integer;
    t: TItem;
    s: string;
begin
  Result:= CreateSet;   // создание пустого множества
```

```
  i:=1;
  while i<= Length(arg) do begin
    // ищем первый не пробел
    while (arg[i]=#32) and (i< Length(arg)) do Inc(i);
    s:='';
    // читаем слово до пробела
    while (arg[i]<>#32) and (i<= Length(arg)) do begin
      s:= s + Upcase(arg[i]);
      Inc(i);
    end;
    Inc(i);
    t:= TItemStr.Create(s);  // создаём объект-строку
    if not Result.Insert(t)  // если не удалось вставить в множество
      then t.Free;           // то удаляем дубликат
  end;
end;

var A, B, C : TSet;   // множества объявлены абстрактными

begin
    Assign(Output, 'Out.txt'); Rewrite(Output);
    A:= MakeSet('while if not begin var function string  end');
    B:= MakeSet('while if not integer const function string');
    C:= CreateSet;   // C = {}
    Write('      A = '); A.Expo;
    Write('      B = '); B.Expo;
    // Объединение:
    Write('    A+B = ');
    C.CopyItems(A);    C.Add(B);    C.Expo;
    // Разность:
    Write('    A-B = ');
    C.CopyItems(A);    C.Sub(B);    C.Expo;
    // Пересечение:
    Write('    A*B = ');
    C.CopyItems(A);    C.Mul(B);    C.Expo;
    // Исключающее ИЛИ:
    Write('A xor B = ');
    C.CopyItems(A);    C.ExOr(B);    C.Expo;

    // Освобождение множеств:
    C.Free;            // здесь элементы не уничтожаем
    B.ClrAndDestroy;   // предварительно уничтожаем элементы
    B.Free;
    A.ClrAndDestroy;   // предварительно уничтожаем элементы
    A.Free;
    Close(Output);
    Readln;
end.
```

Программа вывела в файл **Out.txt** следующий результат (после двоеточия даётся количество элементов множества):

```
      A = { BEGIN END FUNCTION IF NOT STRING VAR WHILE } : 8
      B = { CONST FUNCTION IF INTEGER NOT STRING WHILE } : 7
    A+B = { BEGIN CONST END FUNCTION IF INTEGER NOT STRING VAR WHILE } : 10
    A-B = { BEGIN END VAR } : 3
    A*B = { FUNCTION IF NOT STRING WHILE } : 5
A xor B = { BEGIN CONST END INTEGER VAR } : 5
```

7.3. Сравнение множеств

Следующий пример демонстрирует сравнение множеств методом **Compare** и проверку на пересечение методом **TestIntersect**.

Листинг 7-3 — Сравнение множеств

```
{$APPTYPE CONSOLE}

uses  Items in '..\Common\Items.pas',
      Root in '..\Common\Root.pas';

// Создание множества символов на основе строки

function MakeSet(const arg: string): TSet;
var i: integer;
    t: TItem;
begin
  Result:= CreateSet;   // создание пустого множества
  for i:= 1 to Length(arg) do begin
    t:= TItemChar.Create(UpCase(arg[i]));  // создаём объект-символ
    if not Result.Insert(t)  // если не удалось вставить в множество
      then t.Free;           // то удаляем дубликат
  end;
end;

// Вывод результата сравнения

procedure Show(arg: TCompare);
begin
  case arg of
    cmpEq:     Writeln('Eq');
    cmpLess:   Writeln('Less');
    cmpGreate: Writeln('Greate');
    cmpIncomp: Writeln('Incomp');
  end;
end;

var A, B, C, D : TSet;    // множества объявлены абстрактными

begin
    // Проверка на подмножество и надмножество
    A:= MakeSet('ABCD');
    B:= CreateSet;
    B.CopyItems(A);
    Show(A.Compare(B));  // Eq
    B:= MakeSet('BD');
    C:= MakeSet('DF');
    D:= MakeSet('FE');
    Show(A.Compare(B));  // Greate  (A - это надмножество B)
    Show(B.Compare(A));  // Less    (B - это подмножество A)
    Show(A.Compare(C));  // Incomp
    Show(A.Compare(D));  // Incomp
    // Проверка на пересечение
    Writeln(A.TestIntersect(B));  // TRUE - пересекаются
    Writeln(B.TestIntersect(A));  // TRUE - пересекаются
    Writeln(A.TestIntersect(D));  // FALSE - не пересекаются
    Readln;
end.
```

Метод сравнения **Compare** возвращает одно из четырёх значений:

cmpEq	множества совпадают
cmpLess	первое является подмножеством второго
cmpGreate	второе является подмножеством первого
cmpIncomp	множества не поглощают друг друга

Метод проверки на пересечение **TestIntersect** возвращает **TRUE**, если два множества пересекаются, то есть, содержат хотя бы один общий элемент.

7.4. Итоги

- Объявления множеств — переменных, параметров и типов функций — выполняются через абстрактный класс **TSet**, что избавляет программы от привязки к конкретной реализации класса-множества.
- Объекты-множества будут создаваться функцией **CreateSet** из модуля **Root**. Тело этой функции — единственное место, где указан конкретный класс, реализующий множество (TSetList).
- После вставки элемента методом **Insert** надо анализировать возвращаемый результат. Если метод вернул **FALSE**, значит, множество уже содержит такой элемент, и тогда можно удалить объект-дубликат.
- При удалении множества методом **Free** автоматически вызывается метод **Clear**, очищающий множество, но не уничтожающий его элементы (поскольку элементы могут состоять в нескольких множествах сразу). Для уничтожения элементов вызывается метод **ClrAndDestroy**.
- При сравнении множеств могут быть получены следующие результаты: cmpEq — множества совпадают, cmpLess, cmpGreate — одно является подмножеством другого, cmpIncomp — множества не совпадают и не поглощают друг друга.
- Методы класса **TSet** универсальны, они не привязаны к конкретным классам объектов, составляющих множество. Таким образом, операции с множествами действуют независимо от того, объекты какого типа содержатся в множествах.

Глава 8
Чудовище экспоненты

В ходе решения некоторых задач возникает проблема, суть которой и методы её решения обсудим в этой главе.

8.1. Непростая простота

Представьте следующую картину. Вы проверили новую программу на небольшом наборе данных, скажем, на массиве из пяти элементов. Затем, «скормив» ей в следующем тесте 10-ти элементный массив, вдруг наблюдаете небольшое «торможение» программы. Ещё не чуя беды, вы предлагаете программе набор из двадцати элементов и... что это? Проходит минута, две, десять... Натужно гудит вентилятор процессора, а результата всё нет! Программа зациклилась? Но прервав её и выполнив по шагам, убеждаетесь, что это не так. Вероятней всего, алгоритм этой программы обладает *экспоненциальной* сложностью, в данном случае — *по времени*.

Бывают примеры иного рода, когда программа завершается относительно скоро, но аварийно — по причине нехватки памяти. Тогда говорят об *экспоненциальной* сложности алгоритма *по памяти*. Рассмотрим в этой связи два примера.

Пусть требуется вывести на экран числа от 1 до **N**. Это легко выполнимо для весьма больших значений **N**, а время решения зависит от этого числа линейно. Теперь изменим формулировку: выведем числа от 1 до 10^N. То есть, параметр **N** теперь является показателем степени при 10. Легко догадаться, что уже при небольших значениях параметра **N** компьютеру потребуется заметное время. Когда же параметр составит десятки, или сотни... нет, не стану продолжать, и так понятно, что это задача *экспоненциально* сложна по времени.

Вторую задачу поставим так: сгенерировать и сохранить в памяти список чисел от 1 до 10^N. Ясно, что это тоже *экспоненциально* сложная задача: и по времени, и по памяти, — даже для умеренных значений параметра **N** она завершится по причине нехватки памяти.

Итак, как ни могуч ваш компьютер, существуют задачи, принципиально на нём не решаемые, хотя сами по себе алгоритмы задач весьма просты. Попытаемся разобраться в *классах сложности* алгоритмов.

8.2. Кто круче?

Понятие «сложность» применительно к алгоритмам можно трактовать двояко. Человеку сложность видится в нагромождении конструкций языка: циклов, ветвлений, структур данных. Компьютер смотрит на это иначе и оценивает сложность потребными ресурсами: количеством операций и объёмом необходимой

памяти. Эта разница «взглядов» ясно видна из показанных выше примеров. Строго говоря, компьютер «напрягает» не сложность, а *трудоёмкость* алгоритма. Однако в литературе укоренился термин *сложность*, поэтому в дальнейшем будем говорить о *классах сложности* алгоритмов.

Ясно, что чем больше объём данных, тем больше требуется времени на их обработку. На практике важно знать, насколько крута эта зависимость, именно *крутизна* делит алгоритмы на классы сложности. Ниже мы рассмотрим несколько таких классов в порядке возрастания их сложности.

8.2.1. Константная сложность

Пусть задача состоит в том, чтобы выбрать из массива размерности **N** элемент с индексом **i**. Известно, что время выполнения этой операции не зависит ни от размера массива **N**, ни от индекса **i**, то есть, является константой. Этот факт можно выразить формулой:

$$T(N) = Const$$

где T — время выполнения алгоритма, **N** — объём данных.

Алгоритм константной сложности — мечта программиста!

8.2.2. Линейная сложность

Время исполнения таких алгоритмов зависит от объёма данных линейно, что можно выразить формулой:

$$T(N) = C \bullet N$$

где C — некоторая константа, определяемая характеристиками компьютера и другими факторами. Вот три примера алгоритмов этого класса: а) линейный поиск в массиве; б) выбор **i**–го элемента в односвязном списке; в) перебор всех элементов массива.

8.2.3. Квадратичная сложность

К этому классу относится алгоритм перебора элементов квадратной матрицы, некоторые алгоритмы сортировки массива. Соответствующая им формула такова:

$$T(N) = C \bullet N^2$$

8.2.4. Сложности б**о**льших степеней

Квадратичная сложность — не предел. Некоторые алгоритмы обработки матриц обладают кубической сложностью и сложностью 4-й степени. Иногда

встречаются сложности 5-й и 6-й степеней. Все такие сложности, включая линейную и квадратичную можно выразить формулой:

$$T(N) = C \bullet N^{k}$$

где $k = 1, 2, 3$ и т. д.

8.2.5. Логарифмическая и промежуточные сложности

Наиболее известный алгоритм логарифмической сложности — это двоичный поиск в сортированном массиве, где время поиска выражается формулой:

$$T(N) = C \bullet Log(N)$$

Алгоритмы этого класса весьма привлекательны, ведь логарифмическая функция растёт очень медленно. В поисках совершенства программист стремится свести линейную сложность к логарифмической, квадратичную — к линейной, или промежуточной между линейной и квадратичной. Так, например, сложность алгоритма быстрой сортировки Хоара существенно меньше квадратичной, и выражается формулой:

$$T(N) = C \bullet N \bullet Log(N)$$

8.2.6. Полиномиальная сложность

Подведём промежуточный итог нашего обзора: на рис. 8-1 представлены графики нескольких из представленных выше функций. Здесь функции б**о**льшей крутизны при возрастании **N** рано или поздно обгоняют менее «крутые». По этой причине при сравнении классов сложности пренебрегают постоянным множителем, учитывая лишь показатель степени и логарифм. В итоге получается следующая градация классов в порядке возрастания сложности:

1. $Const$
2. $C \bullet Log(N)$
3. $C \bullet N$
4. $C \bullet N \bullet Log(N)$
5. $C \bullet N^2$
6. $C \bullet N^2 \bullet Log(N)$

и т.д.

Перечисленные выше классы составляют категорию реально выполнимых *полиномиальных* алгоритмов. Это значит, что такие задачи могут быть решены на компьютере за разумное время для всех разумных наборов данных. По крайней мере, не сегодня, так завтра. Теперь обратимся к более «крутым» классам задач.

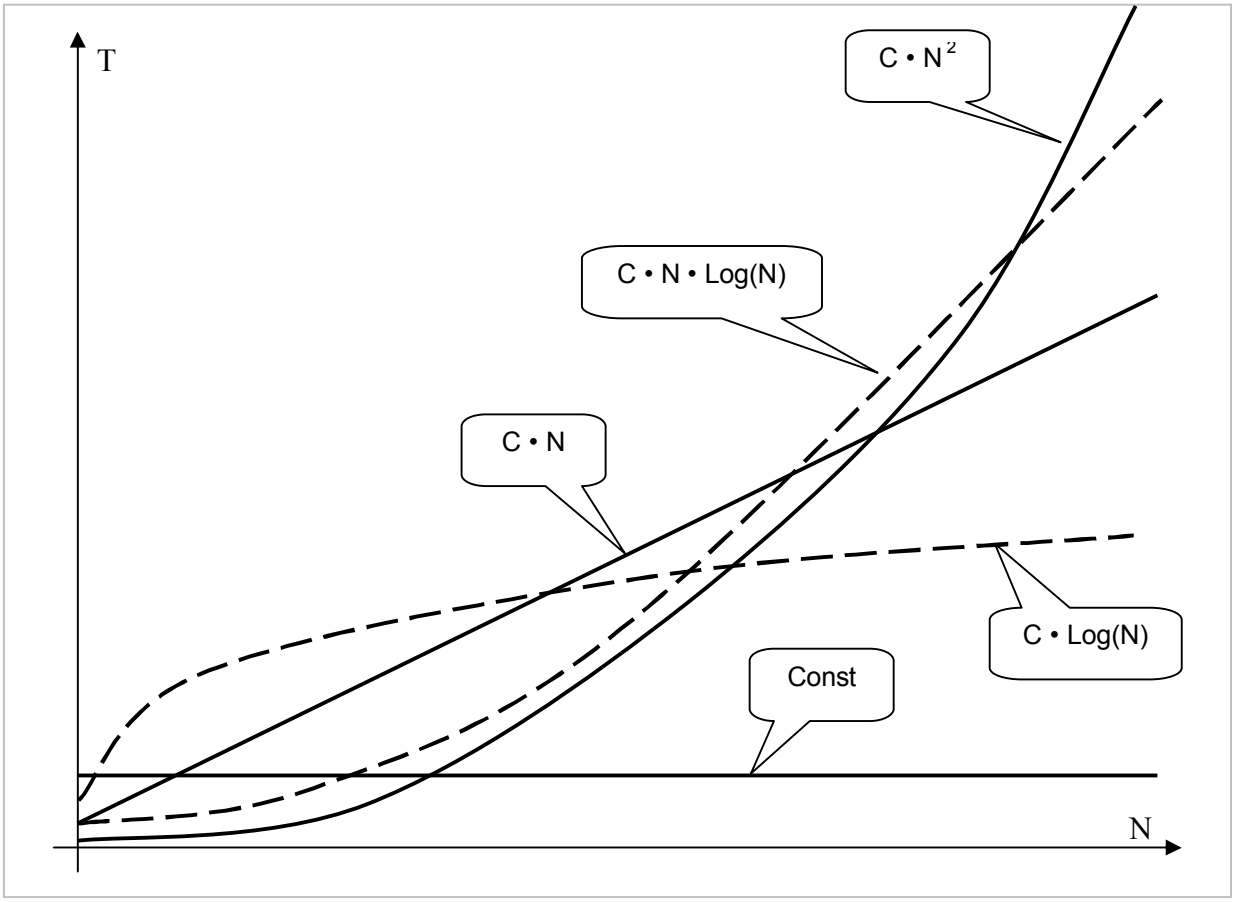


Рис. 8-1 — Графики зависимости трудоёмкости от объёма данных N для нескольких классов алгоритмов

8.2.7. Экспоненциальная сложность

Такую сложность приписывают алгоритмам, время исполнения которых (или объём необходимой им памяти) растут как показательная функция, или ещё круче:

$T(N) = C^N$ — *показательная функция*

$T(N) = C \bullet N!$ — *факториал*

$T(N) = C \bullet N^N$

Как бы круто ни рос полином, но с возрастанием **N** экспонента или факториал рано или поздно превзойдёт любой полином с любым постоянным коэффициентом. Программистов, это не радует, поскольку решение задач такой сложности для требуемых объёмов данных зачастую невозможно. Или почти невозможно. Далее мы рассмотрим две идеи, дающие надежду на преодоление этой проблемы: *поиск с возвратом* и *жадные* алгоритмы.

8.3. Поиск с возвратом

Экспоненциально сложные алгоритмы часто сопряжены с обработкой древовидных структур, где на каждом уровне дерева возможен выбор одного из **N** вариантов, а количество уровней составляет **M**. Время решения с перебором всех вариантов здесь будет пропорционально N^M. Если величины **N** и **M** связаны соотношением **M=C•N**, тогда время поиска решения оказывается пропорциональным N^N. Рассмотрим несложный пример такой задачи.

Пусть дерево на рис. 8-1 изображает схему некой горнолыжной трассы. Воображаемый спортсмен стартует из верхней точки — из корня дерева, и стремится к подножью горы через узлы уровней **A** и **B** к листьям дерева на уровне **C**. Время движения между соседними узлами указано внутри кружков. Смысл задачи в том, чтобы найти кратчайший маршрут между корнем и листом дерева.

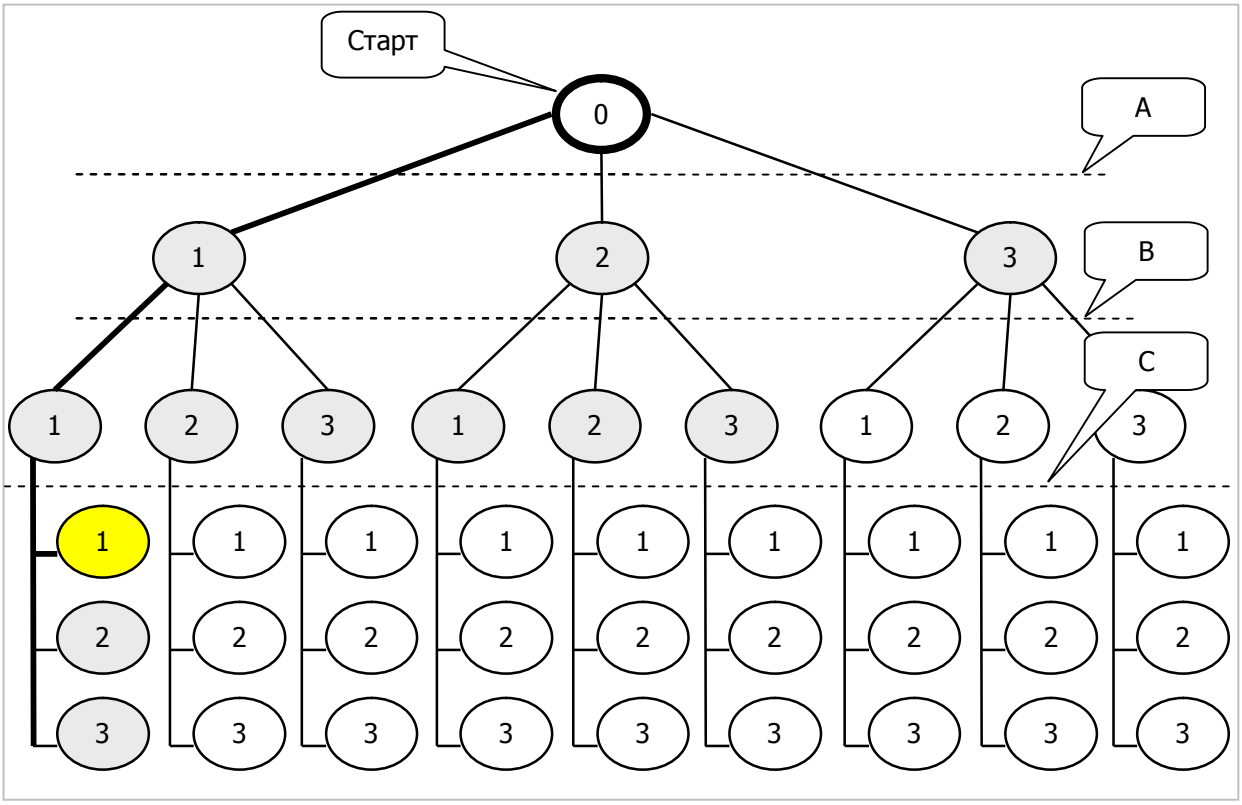


Рисунок 8-1 — Дерево поиска ("счастливый" случай)

Если перебирать все возможные варианты, количество которых зависит от ветвистости дерева и числа уровней, то задача обретёт экспоненциальную сложность вида N^M. Однако если времена будут расставлены так, как показано на рисунке (в возрастающем порядке), а поиск вести сверху вниз и слева направо, то можно существенно ускорить перебор за счёт *поиска с возвратом*.

Действуя так, левую узел уровня **A** лыжник достигнет за 1 минуту. Спустившись к уровню **B**, он достигнет крайнеё левого узла ещё через 1 минуту. Отсюда он попадает на уровень **C**, и здесь — о, удача! — первый узел снова

оказывается ближайшим. В итоге первый спуск занял три минуты, запомним этот результат (на рисунке он отмечен жёлтым). Не будучи уверенным в его оптимальности, спортсмен пробует другие маршруты, дающие времена соответственно **1+1+2** и **1+1+3** минуты (отмечены серым).

Далее спортсмен возвращается к вершине, и пробует другие маршруты. Двигаясь от узла **A1** к узлу **B2** (точки нумеруем слева направо), он затратит **1+2** минуты, — этот результат уже равен текущему лучшему. Стало быть, спускаться отсюда на уровень **C** нет смысла. Это же решение он примет во всех последующих узлах, где текущее время окажется равным, либо большим текущего лучшего результата. В итоге, вместо посещения **3+9+27=39** узлов, лыжник посетит лишь **3+6+3=12**, они отмечены серым. Не посещёнными остались **27** узлов и листьев, они окрашены белым.

Здесь трёхкратный выигрыш достигнут ввиду удачной сортировки узлов. А если времена расположить в обратном порядке? Тогда лучший маршрут будет найден в последней попытке, и никакого выигрыша не случится. В отсутствие сортировки лучший и худший исходы маловероятны. Тем не менее, при больших наборах случайных данных выигрыш времени может исчисляться многими порядками — веская причина вооружиться методом *поиска с возвратом*.

8.4. Принцип жадности

Итак, хотя поиск с возвратом многократно ускоряет перебор вариантов, он не гарантирует двух вещей. Во-первых, не исключает худшего случая, — перебора всех узлов. Во-вторых, что ещё важнее, в некоторых ситуациях затраты времени на поиск решения будут неприемлемы. Пусть подобная задача решается системой управления воздушной обороной, которая выбирает одну из нескольких целей — наиболее опасную. При этом учитывается много факторов, что делает задачу экспоненциально сложной. Стремясь к точному решению, система может упустить драгоценное время, что лишает смысла весь расчёт. Не поискать ли решения, пусть не идеального, но быстрого? Уж лучше сбить одну воздушную цель, чем пропустить их все.

Вернёмся к горнолыжному примеру, изменив данные на уровне **B**, как это показано на рис. 8-2. Теперь идеальный спуск следует по цепочке **A3** → **B1** → **C1**, а соответствующее время составит **3+1+1=5** минут. Однако найти этот путь будет непросто, — придётся перебрать много вариантов. Дадим лыжнику лишь одну попытку, но позволим на каждом уровне выбирать только один из путей, который сулит на данном этапе (уровне) лучшее время.

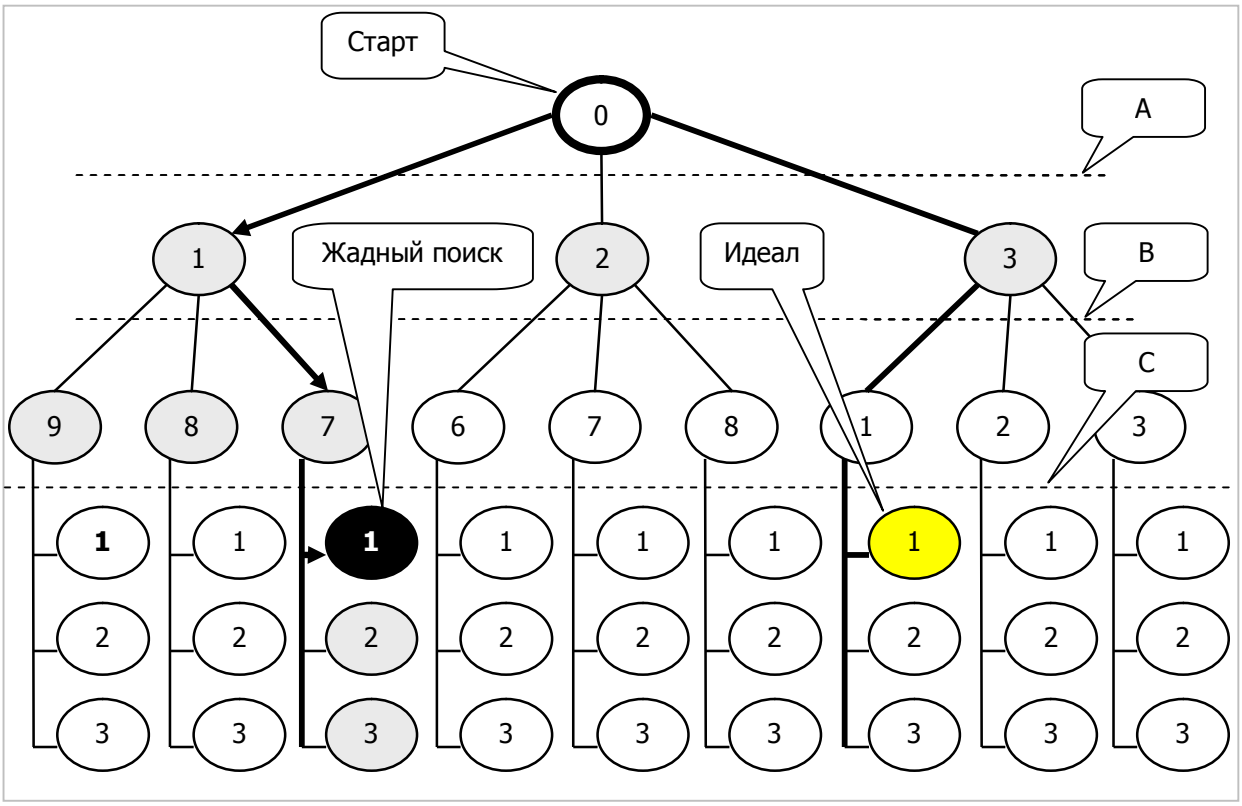


Рис. 8-2 — Дерево поиска с изменённым уровнем B

Следуя этому принципу *жадности*, на первом уровне он выберет узел **A1** (1 минута). Отсюда ему уже не попасть на оптимальную трассу, но жадность диктует своё: далее он последует в узел **B7** (7 минут), а затем в **C1** (1 минута), показав результат **1+7+1=9** минут, — не самый быстрый, но далеко и не худший. Зато как быстро решена задача! Всё потому, что на каждом уровне рассматриваются только **N** узлов, что при **M** уровнях даёт оценку сложности **N•M**. Другими словами, время поиска здесь зависит линейно от ветвистости и глубины дерева. К тому же оно стабильно, не подвластно случайному распределению данных по дереву.

На практике результат жадного поиска бывает не намного хуже оптимального. В будущих опытах на случайных наборах данных мы получим результаты, уступающие оптимальным на 20-30%, что не так уж плохо. Иногда жадный поиск даёт идеальный результат, и почти никогда не даёт худшего. Окончательную оценку жадного принципа дают лишь эксперименты на конкретных данных.

8.5. Комбинации

Итак, мы рассмотрели два способа борьбы с экспоненциальной сложностью: *поиск с возвратом* и принцип *жадности*, каждый из которых обладает достоинствами и недостатками. Можно ли совместить их?

Возьмём ту же систему воздушной обороны. Пусть время, отведенное на обработку данных и выбор подходящей цели, ограничено значением **T**. Тогда можно сначала применить жадный алгоритм и найти пусть не идеальное, но приемлемое решение. Затем запустить поиск с возвратом, позволив ему работать в пределах времени **T**, причём в качестве первого приближения взять результат жадного поиска. Если за время **T** перебор не завершится, то взять текущий лучший результат перебора.

8.6. Итоги

- Время обработки данных пропорционально их объёму, но крутизна этой зависимости определяется алгоритмом. В этой связи все алгоритмы делятся на *классы сложности*.
- По возможности предпочтительно сводить решения к алгоритмам низших классов: логарифмическим, линейным и т.д.
- Классы сложности, где зависимость времени решения от объёма данных определяется полиномом и (или) логарифмом, называют *p-классами*. Соответствующие им задачи решаются на компьютере за разумное время с разумным расходом памяти.
- Классы сложности, где время решения растёт от объёма данных по экспоненте или быстрее, называют *экспоненциальными*. Задачи такого рода, за недостатком ресурсов, обычно не могут быть решены «в лоб».
- Для преодоления экспоненциального барьера используют два принципа: *поиск с возвратом* и *жадность*.
- *Поиск с возвратом* состоит в отбрасывании ветвей, заведомо не ведущих к лучшему результату. На практике метод сокращает время решения на много порядков.
- Принцип *жадности* состоит в выборе на каждом этапе только одного из возможных вариантов, кажущегося лучшим на данном этапе. Это порождает очень быстрые алгоритмы, не гарантирующие идеального результата.
- Жадные алгоритмы иногда дают идеальный результат, и почти никогда не дают худшего.
- В некоторых ситуациях комбинируют *жадный* метод и *поиск с возвратом*, ограничивая его по времени.

8.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
√	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
√	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 9
Разбиение множеств

9.1. Зачем разбивают множества?

Зачем разбивают множества? — познания ради. Этим всегда занималась природа: в стремлении выжить, мозг животных разбивал множества понятий на полезные подмножества, и от успеха этой «учёбы» зависела их судьба. Так, к примеру, глаз лягушки разделяет всё движущееся на два подмножества: мелкие предметы — пища, а крупные — опасность. Лягушке этого довольно. Высшие животные оказывают большую разборчивость и способны различать тысячи разных объектов, неосознанно формируя из них подмножества. Тем же, но сознательно, заняты учёные. К примеру, растения и животных классифицируют по ряду признаков, разбивая их на подмножества. Вероятно, по той же причине ребёнок разбивает игрушку.

А для чего это нам? Будучи истинными учёными, мы попытаемся сформировать все возможные разбиения некоторого множества. Эти разбиения, как исходный материал, пригодятся нам в последующих задачах. С этой целью создадим процедуры генерации случайных разбиений с заданными параметрами.

9.2. Что такое разбиение?

Это известно каждому, кто хоть однажды бил посуду. Разбиение — это то, из чего можно собрать целое, причём без оставления лишних «деталей». Сколькими способами можно разбить целое? Кувшин бьётся бесчисленными способами. Но вариантов «битья» дискретных объектов — множеств, хоть и велико, но конечно.

Рассмотрим разбиения множества из трёх символов: { a, b, c }. Сколькими способами оно бьётся? Во-первых, поскольку множество является собственным подмножеством, разбиение может состоять из самого множества и пустого подмножества, которое обычно игнорируют. Во-вторых, его можно разбить на подмножества по одному символу в каждом. Есть и другие варианты разбиения, все они показаны в табл. 9-1.

Табл. 9-1 — Варианты разбиения множества { a, b, c }

Разбиение	Количество подмножеств
{ {a, b, c} }	1
{ {a}, {b, c} }	2
{ {a, b}, {c} }	2
{ {a, c}, {b} }	2
{ {a}, {b}, {c} }	3

Во всех вариантах *разбиение* — это *множество подмножеств*, сумма которых даёт целое множество, причём все подмножества одного разбиения

взаимно не пересекаются. Пустое подмножество неявно присутствует в любом разбиении.

Для облегчения текста далее принята сокращённая форма записи множества множеств; в этой краткой форме удалены внутренние запятые и скобки, что показано в следующей сравнительной таблице.

Табл. 9-2 — Развёрнутая и краткая формы записи множества множеств

Развёрнутая форма	Краткая форма
{ {a, b, c} }	{ abc }
{ {a}, {b, c} }	{ a, bc }
{ {a, b}, {c} }	{ ab, c }
{ {a, c}, {b} }	{ ac, b }
{ {a}, {b}, {c} }	{ a, b, c }

9.3. Все разбиения

Сейчас поставим и *почти* решим следующую задачу: составить *все* возможные разбиения данного множества. Каждое разбиение будет представлять собой множество подмножеств. Попутно выясним возможное количество разных разбиений, и тогда выяснится, почему задача решается лишь «почти».

Идея алгоритма основана на последовательном расширении исходного множества, начиная с пустого. Пусть на каком-то этапе известны все разбиения множества из предыдущих **N-1** элементов. Тогда все разбиения для множества из **N** элементов можно получить отсюда двумя способами:

1. *Вставить* очередной элемент в одно из уже сформированных подмножеств разбиения, — так последовательной вставкой элементов получим *несколько* разбиений.
2. *Добавить* к существующим подмножествам ещё одно подмножество, включающее лишь один элемент, — так получим ещё *одно* разбиение.

В табл. 9-3 дана последовательность построения всех разбиений для множества из четырёх элементов. Попутно строятся разбиения для множеств из одного, двух и трёх элементов (в таблице используется краткая форма записи множества подмножеств).

Табл. 9-3 — Построение всех разбиений последовательным расширением исходного множества

Исходные разбиения	Добавляемый элемент	Способы формирования и результаты разбиений		Кол-во разбиений
		1) Вставкой	2) Добавлением	
1-й элемент (a)				
{ }	a	нет	{ a }	1
2-й элемент (b)				
{ a }	b	{ ab }	{ a, b }	2
3-й элемент (c)				
{ ab }	c	{ abc }	{ ab, c }	5
{ a, b }		{ ac, b }	{ a, b, c }	
		{ a, bc }		
4-й элемент (d)				
{ abc }	d	{ abcd }	{ abc, d }	15
{ ac, b }		{ acd, b }	{ ac, b, d }	
		{ ac, bd }		
{ a, bc }		{ ad, bc }	{ a, bc, d }	
		{ a, bcd }		
{ ab, c }		{ abd, c }	{ ab, c, d }	
		{ ab, cd }		
{ a, b, c }		{ ad, b, c }	{ a, b, c, d }	
		{ a, bd, c }		
		{ a, b, cd }		

Итак, начнём с пустого множества, — его разбиение тоже пусто. Поэтому при обработке первого элемента нельзя создать разбиения путём вставки — некуда вставлять. Здесь ограничимся вторым способом: добавим множество из единственного элемента {a}.

При обработке второго элемента применимы оба способа, поэтому здесь получаем два разбиения: { ab } и { a, b }, — здесь и далее я использую краткую форму записи подмножеств. На следующем шаге эти два разбиения будут «сырьём» для построения разбиений множества из трёх элементов и т.д.

В правом столбце табл. 9-3 показано количество получаемых разбиений. Эти числа называют *числами Белла*, они вычисляются как сумма *чисел Стирлинга* второго рода (подробнее об этом читайте в рекомендуемой литературе). В табл. 9-4 дана зависимость чисел Белла от количества элементов в разбиваемом множестве (его мощности). Количество возможных разбиений растёт с ростом множества чрезвычайно быстро (быстрее экспоненты). Стало быть, задача формирования *всех* разбиений множества *экспоненциально сложна*, и может быть решена на практике лишь для небольших множеств.

Табл. 9-4 — Числа Белла B(n)

Мощность множества N	Количество возможных разбиений B(N)
1	1
2	2
3	5
4	15
5	52
6	203
7	877
8	4 140
9	21 147
10	115 975
11	678 570
12	4 213 597
13	27 644 437
14	190 899 322
15	1 382 958 545
16	10 480 142 147
17	82 864 869 804
18	682 076 806 159
19	5 832 742 205 057
20	51 724 158 235 372

Впрочем, экспоненциальная сложность задачи не помешает нам создать решающую её процедуру. Точнее, это будет функция, принимающая аргумент-множество и возвращающая другое множество, состоящее из *всех* разбиений исходного множества. Иными словами, результат будет «множеством в кубе», поскольку его элементами будут разбиения, то есть множества, а те, в свою очередь, будут множествами подмножеств. Так, к примеру, для исходного множества {a,b} будет сформирован результат { { {a,b} }, { {a}, {b} } } — множество множеств, содержащих подмножества (дано в развёрнутой форме).

Перед вами листинг функции **GenAllDissections** из модуля **Dissect**; некоторые моменты прокомментирую подробней.

Листинг 9-1 — Формирование всех разбиений множества

```
function GenAllDissections(aSet: TSet): TSet;
var Que   : TBuffer;  // буфер-очередь
    SSGet : TSet;     // извлекаемое множество подмножеств
    SSPut : TSet;     // добавляемое множество подмножеств
    //-----------------------------------------
    // Процедура обработка всех подмножеств
    // из множества множеств SSGet

    procedure LocalHandle(aItem: TItem);
    var k: integer;
        S: TSet;     // текущее множество из множества подмножеств
    begin
      // Обработка очередного извлечённого множества подмножеств SSGet,
      // перебираем его элементы-множества S:
      for k:= 1 to SSGet.GetCount do begin
          SSPut:= SSGet.Copy as TSet;  // копия извлеч. множества подмножеств
          S:= SSPut.GetItem(k) as TSet; // очередное множество из МП
          SSPut.Delete(S);   // удаляем из него очередное подмножество
          S.Insert(aItem);   // к очередному добавляем текущий элемент
          SSPut.Insert(S);   // полученное вставляем в множество подмножеств
          Que.Put(SSPut);    // новое множество подмножеств заносим в очередь
      end;
      // Добавление нового подмножества к SSGet
      SSPut:= SSGet.Copy as TSet;  // копия извлеч. множества подмножеств
      S:= CreateSet;       // создаём множество
      S.Insert(aItem);     // из единственнного элемента aItem
      SSPut.Insert(S);     // добавляем к множеству подмножеств
      Que.Put(SSPut);      // новое множество подмножеств заносим в очередь
    end;
    //-----------------------------------------

var   i, j  : integer;  // счётчики
      Item: TItem;      // текущий элемент исходного множества

begin
  Que:= TBuffer.Create;   // создание буфера для очереди
  SSPut:= CreateSet;      // пустое множество подмножеств
  Que.Put(SSPut);         // заносим в очередь
  // Цикл по всем элементам исходного множества:
  for i:= 1 to aSet.GetCount do begin
    Item:= aSet.GetItem(i);
    // Цикл обработки очереди:
    for j:=1 to Que.GetCount do begin
      SSGet:= Que.Get as TSet;  // берём очередное множество из очереди
      LocalHandle(Item);        // создаём из него ряд других
      SSGet.ClrAndDestroy;      // очищаем от подмножеств
      SSGet.Free;               // и уничтожаем
    end;
  end;
  Result:= CreateSet; // создание пустого результата
  // Перенос из буфера в множество-результат
  while Que.GetCount>0 do Result.Insert(Que.Get);
  Que.Free;  // удаление буфера
end;
```

В основном теле функции создаётся рабочий буфер **Que**, — он будет служить очередью обрабатываемых разбиений. Вначале в очередь помещается пустое множество **SSPut** — для «затравки» процедуры.

Далее следует внешний цикл обработки элементов множества. Для каждого элемента организуем внутренний цикл, в котором обрабатываем все элементы очереди (все разбиения), находящиеся в ней на момент входа в этот внутренний цикл (при старте туда занесено одно пустое множество). Извлечённый из буфера элемент-разбиение **SSGet** обрабатываем локальной процедурой **LocalHandle**, а затем уничтожаем вместе с его содержимым.

Локальная процедура **LocalHandle** создаёт несколько новых разбиений из переданного ей разбиения по правилам, рассмотренным выше, и вставляет их в ту же самую очередь. Стало быть, очередь постепенно разрастается по закону чисел Белла. После обработки всех элементов исходного множества результат переносим из очереди **Que** в множество **Result**, а очередь **Que** удаляем.

Теперь рассмотрим тестирующую программу. Внешне она проста, но некоторые моменты нуждаются в разъяснениях (листинг 9-2).

Листинг 9-2 — Тестирующая программа для получения всех разбиений множества

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Dissect in '..\Common\Dissect.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  Graph in '..\Common\Graph.pas',
  SetUtils in '..\Common\SetUtils.pas',
  Assembly in '..\Common\Assembly.pas';

// Вспомогательная процедура удаления подмножеств внутри разбиений

procedure FreeSubSets(arg: TSet);
var  S : TSet;  // очередное разбиение (множество подмножеств)
begin
  S:= arg.GetFirst as TSet;   // выбор первого разбиения
  while Assigned(S) do begin
    S.ClrAndDestroy;          // удаление всех подмножеств разбиения
    S:= arg.GetNext as TSet;  // выбор следующего разбиения
  end;
end;

var
    Power   : integer;  // Мощность исходного множества
    Univers : TSet;     // Исходное множество объектов -- УНИВЕРСУМ
    Res: TSet;          // все разбиения множества

begin  {----- Main -----}
  repeat
    Write('Power= '); Readln(Power);  // Ввод мощности множества
    if Power<2 then Break;
```

```
    // Генерация множества символов заданной мощности
    Univers:= GenerateChars(Power);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - -');

    Res:= GenAllDissections(Univers);  // генерация всех разбиений
    Res.Expo;
    Writeln('- - - - - - - - - - - - -');

    FreeSubSets(Res);    // удаление подмножеств внутри всех разбиений
    Res.ClrAndDestroy;   // удаление самих разбиений
    Res.Free;            // удаление множества разбиений

    Univers.ClrAndDestroy; // очистка базового "универсума"
    Univers.Free;          // удаление базового "универсума"
  until false;
end.
```

Прежде всего, обратим внимание на глобальную переменную **Univers** — универсум. Это исходное множество, разбиения которого надо получить. Отметим, что все создаваемые далее подмножества этого универсума будут содержать ссылки на его элементы (а не копии элементов). Стало быть, элементы универсума должны существовать, пока существуют подмножества этого универсума. Отсюда следует, что универсум может быть уничтожен лишь по завершении программы.

Примечание. Напомню, что *универсумом* называют множество всех мыслимых объектов, но в контексте нашей программы мы сужаем это понятие.

Итак, после ввода пользователем мощности множества, создаётся универсум символов функцией **GenerateChars** из модуля **SetUtils**. Затем формируется множество всех разбиений и выводится на экран. На этом можно было бы завершить программу, если бы наша система поддерживала автоматическую сборку «мусора». Такие системы существуют, но **Delphi** к ним не относится. Поэтому в реальных проектах требуется аккуратность по части освобождения памяти от ненужных объектов. Забывчивость в этом деле ведёт к засорению памяти, а преждевременное удаление объектов — к аварии программы. Поэтому покажем здесь технику освобождения объектов, добавив ещё несколько строк.

В целом правило удаления объектов несложно: объекты удаляют в порядке, обратном их созданию. Универсум был создан первым, и потому будет удалён последним. Но над удалением результата **Res** надо подумать. Вызов деструктора через метод **Free** удалит лишь саму переменную-множество, но не элементы множества, и тогда разбиения останутся в памяти. Предварительный вызов метода **ClrAndDestroy**, казалось бы, решает проблему — удаляет элементы-разбиения, но сами эти элементы тоже содержат множества — подмножества универсума, и они останутся, засорив память. Поэтому удаление начато с подмножеств во вспомогательной процедуре **FreeSubSets**. Отмечу, что это не единственный приём очистки, альтернативой может быть конструирование на базе множества наследника, в деструкторе которого будут учтены тонкости его очистки.

В заключение показан образец вывода программой всех пяти разбиений трёхэлементного множества:

```
{ a b c } : 3 — исходное множество
- - - - - - - - - - - - -
{
  {
    { a b c } : 3
  } : 1

  {
    { a b } : 2
    { c } : 1
  } : 2

  {
    { b } : 1
    { a c } : 2
  } : 2

  {
    { a } : 1
    { b c } : 2
  } : 2

  {
    { a } : 1
    { b } : 1
    { c } : 1
  } : 3
} : 5
- - - - - - - - - - - - -
```

9.4. Случайные разбиения

Итак, представленная выше программа генерирует *все* возможные разбиения множества. Однако практическая ценность этого достижения не высока. Попытка получить все разбиения универсума всего лишь из восьми элементов посеет сомнения в мощи вашего компьютера. О более крупных множествах и говорить нечего: вопрос будет лишь в том, что иссякнет раньше: память компьютера или ваше терпение.

Но кому нужны столь чудовищные разбиеня? На практике имеют дело с разбиениями вполне разумных масштабов, мощность которых исчисляется сотнями, тысячами, миллионами. Такие разбиения формируются предикатами, продиктованными самой жизнью. Комбинируя разные предикаты и применяя их к разным подмножествам, можно получать и *разбиения*, и пересекающиеся подмножества универсума — *покрытия*.

Но здесь не о предикатах. Создадим процедуру, которая из универсума будет создавать некоторое количество его подмножеств с заданными характеристиками, причём формировать этот набор случайным образом. Результат потребуется в двух последующих главах, где будет решаться обратная задача — соединение подмножеств в целое.

Создание случайных разбиений сродни битью глиняных горшков. Возьмём несколько одинаковых горшков (каждый горшок — это универсум). Разбив очередной горшок, отбросим случайно выбранную треть его обломков, а остальные две трети сложим в мешок. Горшки будут биться так, чтобы черепки получались не слишком крупными, и не слишком мелкими, — эти граничные размеры осколков задаются параметрами программы. Вот и всё.

Можно ли этих черепков собрать целый горшок? В отношении реальных горшков — маловероятно. Но мы бьём цифровые «горшки», «черепки» которых дискретны, и потому, обладая достаточным количеством таких «осколков», вполне возможно склеить хотя бы один целый.

Сначала рассмотрим вспомогательную функцию **GenRandSet** модуля **Dissect**, создающую одно случайное разбиение переданного ей множества. Она исполняет роль молотка, разбивающего горшок. Параметры **aMin** и **aMax** определяют минимальный и максимальный размеры «черепков».

Листинг 9-3 — Вспомогательная функция, генерирующая одно случайное разбиение

```
//  Генерация случайного разбиения исходного множества aSet

function GenRandSet(aMin, aMax: integer; aSet: TSet): TSet;
var n, k, j: integer;
    t: TItem;
    S: TSet;
begin
  Result:= CreateSet;                 // создаём множество-результат
  repeat
    n:= aMin + Random(aMax-aMin);     // случайная мощность подмножества
    S:= CreateSet;                    // создаём подмножество
    for j:= 1 to n do begin           // и заполняем случайным выбором элем-ов
      k:= 1 + Random(aSet.GetCount); // случайный индекс в исходном множестве
      t:= aSet.GetItem(k);            // случайный элемент исходного множества
      S.Insert(t);                    // вставляем в множество
      aSet.Delete(t);                 // и удаляем из исходного
      if aSet.GetCount=0 then Break; // выход из цикла, если множ. исчерпано
    end;
    if S.GetCount>0           // если множество не пусто,
      then Result.Insert(S)   // то вставляем в результат
      else S.Free;            // а иначе удаляем
  until aSet.GetCount=0;      // пока не исчерпано исходное множество
end;
```

Эта функция вызывается из функции **GenSubsRand**, в результате работы которой создаётся буфер, заполненный подмножествами. Образно говоря, эта функция создаёт мешок и складывает в него черепки. В качестве параметров передаются: а) исходное множество-универсум, б) количество генерируемых разбиений и в) предельные размеры «черепков» в процентах от целого.

Листинг 9-4 — Функция GenSubsRand,
генерирующая заданное количество случайных подмножеств

```
// Генерирует заданное в aCnt количество подмножеств исходного множества.
// Параметры задают минимальную и максимальную (%) мощность подмножеств

function GenSubsRand(aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TBuffer;
var
  nMin, nMax: Integer;
  Copy: TSet;  // рабочая копия исходного множества aSet
  SS: TSet;    // очередное разбиение
  S : TSet;    // очередное множество внутри разбиения

begin
  // Перевод процентов в количество

  nMin:= (aSet.GetCount * aMin) div 100;   // aMin %
  if nMin<1 then nMin:=1;
  if nMin>=aSet.GetCount then nMin:=aSet.GetCount-1;

  nMax:= (aSet.GetCount * aMax) div 100;   // aMax %
  if nMax<nMin then nMax:=nMin;
  if nMax>aSet.GetCount then nMax:=aSet.GetCount;

  Result:= TBuffer.Create;  // создаём буфер результата

  // Цикл создания заданного количества подмножеств

  while aCnt>0 do begin
    Copy:= aSet.Copy as TSet;   // создаём копию исходного множества
    repeat
      SS:= GenRandSet(nMin, nMax, Copy);  // создаём случайное разбиение
      // Перебор подмножеств случайного разбиения:
      S:= SS.GetFirst as TSet;   // первое подмножество из разбиения
      while Assigned(S) and (aCnt>0) do begin
        if Random(3)<>0 then begin
          // копия очередного "осколка"
          // с вероятностью 2/3 вставляется в результат
          Result.Put(S.Copy);
          Dec(aCnt);
        end;
        S:= SS.GetNext as TSet;  // следующее подмножество из разбиения
      end;
      SS.ClrAndDestroy;  // очищаем разбиение
      SS.Free;           // и удаляем его
    until (aCnt=0) or (Copy.GetCount=0);
    Copy.Free;   // удаляем копию исходного множества
  end; // while
end;
```

Далее представлена тестирующая программа.

Листинг 9-5 — Программа для генерации случайных подмножеств

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
  Univers : TSet;      // Исходное множество объектов -- УНИВЕРСУМ
  Buf: TBuffer;      // Буфер с подмножествами

begin
    // создаём универсум из 20-ти объектов:
    Univers:= GenerateChars(20);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - -');
    // Формируем 10 подмножеств,
    // размер максимального "черепка" не превышает 40% от целого
    Buf:= GenSubsRand(Univers, 10, 0 {%}, 40 {%});
    Buf.Expo;
    Writeln('- - - - - - - - - - - - -');
    Buf.ClrAndDestroy;
    Buf.Free;
    Univers.Free;
    Readln;
end.
```

Вот результат работы этой программы:

```
{ a b c d e f g h i j k l m n o p q r s t } : 20
- - - - - - - - - - - - -
BufCnt= 10
{ d e g i k l o } : 7
{ s } : 1
{ b h j n q r t } : 7
{ c l m n o } : 5
{ f h r s } : 4
{ e g i } : 3
{ f l n t } : 4
{ c f j n p } : 5
{ k m q } : 3
{ h } : 1
- - - - - - - - - - - - -
```

9.5. Генерация разбиения из текстового файла

Модуль **Dissect** содержит ещё одну функцию — **GenSubsFromText**. Она генерирует псевдослучайные подмножества символов из текстового файла. Читая символ за символом, функция накапливает очередное подмножество до тех пор, пока следующий символ не окажется «лишним», т.е. уже существующим в подмножестве. В этот момент накопление прерывается, подмножество заносится в буфер, накопитель подмножества очищается и всё повторяется. Одновременно формируется универсум — множество всех прочитанных из файла символов.

Листинг 9-6 — Функция создания подмножеств и универсума из файла

```
function GenSubsFromText(const aName: String;  // имя файла
                         var aSet: TSet;       // результат-УНИВЕРСУМ
                         aCnt: integer         // количество подмножеств
                         ): TBuffer;
type TChrSet = set of char;
    // - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Формирование объекта-множества символов
    // из классического множества символов
    function MakeFromSet(const aSet: TChrSet): TSet;
    var c: char;
        t: TItemChar;
    begin
      Result:= CreateSet;
      for c in aSet do begin
        t:= TItemChar.Create(c);
        Result.Insert(t)
      end;
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - -
var F: Text;            // текстовый файл
    S: string;          // строка файла
    c: char;            // очередной символ
    i: integer;         // индекс
    set1 : TChrSet;     // множество для УНИВЕРСУМа
    set2 : TChrSet;     // множество для подмножества
    subset: TSet;       // очередное подмножество
begin
  Result:= TBuffer.Create;
  set1:= [];
  set2:= [];
  Assign(F, aName); Reset(F);
  while not Eof(F) do begin
    if Result.GetCount >= aCnt then break;
    Readln(F, S);
    for i:= 1 to Length(S) do begin
      // Обработка очередной строки файла:
      if Result.GetCount >= aCnt then break;
      c:= Char(Ord(S[i]) or $20);        // low case (буквы 'a'..'z')
      if c in ['a'..'z'] then begin
        set1:= set1 + [c];               // для УНИВЕРСУМА
        if c in set2 then begin          // если символ встретился повторно
           subset:= MakeFromSet(set2);   // создаём подмножество
           Result.Put(subset);           // и заносим в результат
           set2:= [c];                   // начинаем следующее подмножество
        end else begin                   // если символ встретился впервые
           set2:= set2 + [c];            // расширяем текущее подмножество
```

```
        end;
      end;
    end;
  end;
  Close(F);
  subset:= MakeFromSet(set2);  // оставшееся множество
  Result.Push(subset);         // вставляем в результат
  aSet:= MakeFromSet(set1);    // и создаём универсум
end;
```

Работу функции можно проверить программой, подобной этой:

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
  Univers : TSet;      // УНИВЕРСУМ
  Buf: TBuffer;      // Буфер с подмножествами

begin
    Buf:= GenSubsFromText('Set_08c.dpr',  // имя файла
                          Univers,        // результат-УНИВЕРСУМ
                          10              // количество подмножеств
                        );
    Univers.Expo;
    Writeln('- - - - - - - - - - - - -');
    Buf.Expo;
    Buf.ClrAndDestroy;
    Buf.Free;
    Univers.Free;
    Writeln('- - - - - - - - - - - - -');
    Readln;
end.
```

9.6. Итоги

- Разбиение множеств на подмножества происходит в ходе естественной умственной деятельности.
- Подмножества одного *разбиения* в сумме дают исходное множество и взаимно не пересекаются.
- Количество разбиений одного множества растёт с ростом его мощности по закону, обгоняющему экспоненту, и потому получить все возможные разбиения возможно только для небольших множеств.
- Случайное разбиение множеств моделирует битьё горшков, такие разбиения будут использоваться в последующих главах.

9.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
√	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
√	8	Липский В.	Комбинаторика для программистов	Стр. 43-50
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 10
Задача о наименьшем разбиении (ЗНР)

10.1. Зачем собирают множества

В предыдущей главе созданы процедуры для разбития множества на ряд подмножеств. Обычно подмножества являются продуктом естественной умственной деятельности, и порой требуется решать обратную задачу: *собрать* нечто целое из этих «обломков». Пусть мастеру потребовался ряд слесарных инструментов, которые продаются только в наборах. Тогда он будет искать самое дешёвое множество наборов, содержащее все нужные ему инструменты. Некоторые задачи на графах сводятся к сборке множества из подмножеств. Ввиду важности этой проблемы уделим ей должное внимание в этой и следующей главе.

10.2. Разбиения и покрытия

Прежде, чем сформулировать и решить «сборочные» задачи, определим два близких, хотя и разных понятия: *разбиение* и *покрытие*. Оба являются множествами подмножеств. *Разбиение* — это такая совокупность взаимно *непересекающихся* подмножеств, из которых можно собрать исходное множество (универсум). *Покрытие* — это то же самое, однако входящие в него подмножества *могут* взаимно *пересекаться* (накладываться).

Названия говорят сами за себя: *разбиение* — это результат битья посуды. Из осколков разных горшков теоретически можно собрать целый, если ни один черепок не «напирает» на смежный. *Покрытие* сродни устиланию пола обрывками некогда целых одинаковых ковров, которые можно уложить с сохранением прежнего рисунка, и допуская *наложение* отдельных совпадающих участков. Различие этих понятий демонстрирует рис. 10-1. Отметим, что *разбиение* множества является одновременно и его *покрытием*, но не наоборот.

10.3. ЗНР и ЗНП

Теперь сформулируем две задачи *сборки* множества из его подмножеств. Пусть дана совокупность подмножеств некого универсума, — подобие мешка с черепками от битых горшков. В первом варианте *задачи о наименьшем разбиении* (ЗНР) требуется подобрать минимальное *количество взаимно непересекающихся* подмножеств, дающих в сумме исходный универсум. Во втором варианте этой же задачи каждому подмножеству назначена *цена*, и надо подобрать совокупность подмножеств с наименьшей общей *стоимостью*. Второй вариант вырождается в первый, если всем «черепкам» назначить единичную цену, поэтому всё внимание уделим второму варианту.

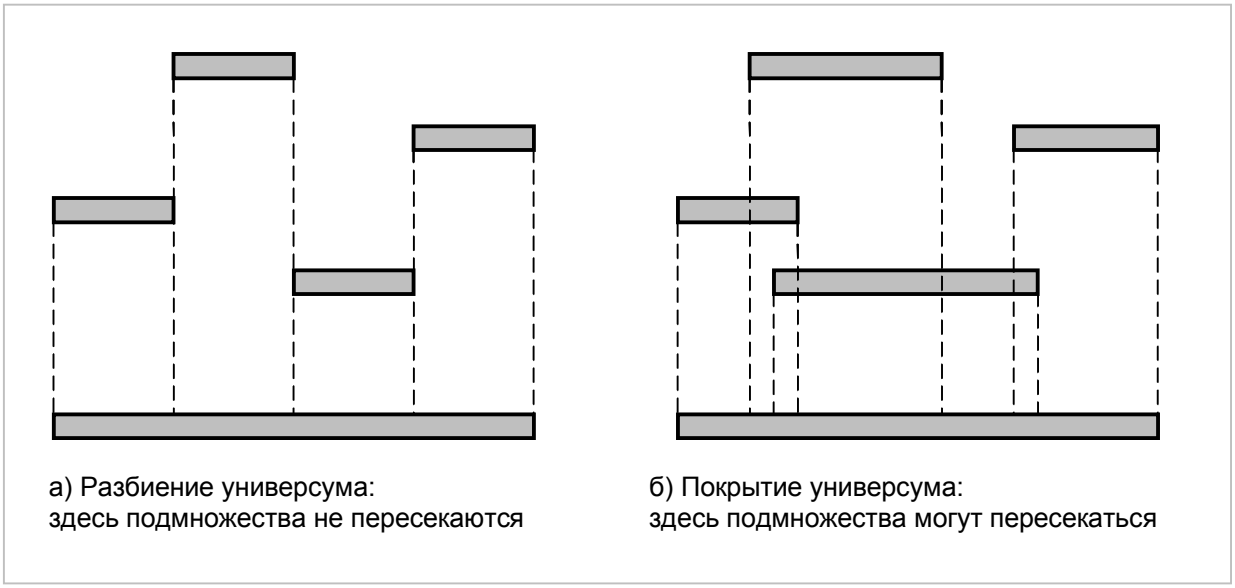


Рис. 10-1 — Разбиение (а) и покрытие (б)

Сходным образом ставятся и *задачи о наименьшем покрытии* (ЗНП). Дана та же совокупность подмножеств, — подобие фрагментов ковра. Надо найти либо наименьшее *количество* подмножеств, покрывающих универсум, либо множество подмножеств («лоскутов») с наименьшей *стоимостью*. В ЗНП не запрещается взаимное пересечение подмножеств. Так же, как и для ЗНР, первый вариант задачи будет частным случаем второго, который мы и будем решать.

На первый взгляд, подобрать *покрытие* проще, чем *разбиение*, ведь требование взаимного не пересечения подмножеств тут не предъявляется. Реально ровно наоборот: количество возможных вариантов покрытий больше, и потому перебор этого «богатства» длится дольше. Компьютеру ЗНП даётся существенно труднее ЗНР, в следующей главе вы убедитесь в этом. Далее в этой главе будет обсуждаться решение задачи о наименьшем разбиении — ЗНР.

10.4. Исходные данные

Начнём с описания исходных данных. Воображаемый «мешок с черепками» представим набором подмножеств, где каждому подмножеству назначена цена. Нечто подобное генерировали случайным образом функции **`GenSubsRand`** и **`GenSubsFromText`** (см. модуль **`Dissect`**). Но там не было цен. Теперь назначим каждому подмножеству число, указывающее его цену. Здесь возможны разные решения, мы объединим эти данные агрегатированием, то есть через создание нового класса **`TCostSet`**, содержащего оценку и множество в качестве своих полей. Класс унаследован от базового объекта **`TItem`**, ниже дано его объявление, а за подробностями отсылаю к модулю **`Root`**.

```
// Класс "множество с оценкой"

  TCostSet = class (TItem)
    protected
      procedure Clr(aDestroy: boolean);
    public
      mCost: integer;       // оценка, стоимость элемента
      mSet : TSet;          // подмножество
      mFlag  : boolean;     // флажок для пометок
      mDestroy : boolean;   // определяет способ уничтожения объекта
      constructor Create(aCost: integer; aSet: TSet; aDestroy : boolean);
      constructor CreateEmpty;
      destructor  Destroy;  override;
      function    Copy: TItem; override;
      procedure   Clear;
      procedure   ClrAndDestroy;
      function    Compare(arg: TItem): TCompare; override;
      procedure   Print(var aFile: TextFile); override;
      procedure   Append(arg : TCostSet);
      procedure   Insert(arg : TCostSet);
  end;
```

Итак, роль отдельного черепка исполняет объект класса **TCostSet**. «Мешком» послужит объект-буфер класса **TBuffer**. Две упомянутые выше функции генерируют буферы, наполненные подмножествами **TSet**, но нам требуется другое наполнение — наполнение объектами класса **TCostSet**. Модуль **SetUtils** содержит функцию **GenCostItems**, преобразующую первый род буфера во второй. Наряду с исходным буфером, ей передаётся параметр, определяющий стратегию назначения цены элементам буфера, — возможность варьировать стратегией пригодится при испытании алгоритмов ЗНР и ЗНП. Предусмотрены пять вариантов стратегии (см. листинг 10-1).

Листинг 10-1 — Функция формирования буфера подмножеств с ценами.

```
// Преобразует исходный набор элементов в набор элементов с ценами
// aBuf  - исходный набор
// aCost - стратегия оценки:
//   0 - случайные оценки в диапазоне от 1 до 99
//   1 - все оценки равны единице
//   2 - случайные оценки, пропорциональные мощности подмножества
//   3 - оценка равна мощности подмножества
//   4 - оценка пропорциональна квадрату мощности подмножества
function GenCostItems(aCost: integer; aBuf: TBuffer): TBuffer;
var S: TSet;
    cost: integer;
    n: integer;
begin
  Result:= TBuffer.Create;
  S:= aBuf.Get as TSet;
  while Assigned(S) do begin
    n:= S.GetCount;
    case aCost of
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
     else cost:= 1 + Random(99)
    end;
```

```
    Result.Put(TCostSet.Create(cost, S, true));
    S:= aBuf.Get as TSet;
  end;
end;
```

Стратегия «1» состоит в присвоении единичных цен всем подмножествам, — она нужна для вариантов ЗНР и ЗНП без учёта их стоимости. Три следующие стратегии формируют цены, так или иначе зависящие от мощности подмножеств. Стратегия «0» и все прочие дают случайные цены в диапазоне от 1 до 99.

Другой «кузницей» исходных данных служит функция **ReadSetsFromText** (модуль **SetUtils**), — она читает данные из текстового файла, в котором цены уже указаны. В конце концов, обе функции нужны лишь для испытания создаваемых алгоритмов, подробности найдёте в листинге модуля **SetUtils**. Теперь переходим к обсуждению алгоритма ЗНР.

10.5. Предварительные соображения

10.5.1. Перебор

Предстоящие задачи сродни сборке машины без чертежа методом проб и ошибок. Каков порядок сложности этой задачи? Пусть дана совокупность из **N** «осколков». Взяв один из «черепков», что даёт **N** вариантов дальнейшего объединения, попытаемся присоединить к нему любой из оставшихся **N-1**. Затем к этим двум — любой из оставшихся **N-2** и т.д. Тогда количество вариантов перебора определится выражением:

$$O(N) = N \cdot (N-1) \cdot (N-2) \cdot \ldots$$

Перед нами факториал, стало быть, задаче свойственна *экспоненциальная* сложность. Используя принцип досрочного возврата, можно многократно уменьшить число комбинаций, и всё же сложность задачи останется *экспоненциальной*, и потому нам следует позаботиться ещё о двух вещах.

10.5.2. А существует ли решение?

Ввиду большого объёма работы, следует предварительно выяснить, а стоит ли за неё браться? Ведь нет гарантии, что из предлагаемого набора подмножеств можно собрать хотя бы одно разбиение или покрытие. К счастью, проверить *невозможность* решения очень легко и мы это обязательно сделаем.

10.5.3. Дубликаты

В связи с «крутизной» факториала каждый лишний элемент в наборе исходных данных существенно усложняет задачу. А такие элементы при «битье горшков» вполне вероятны. Ведь разрушение наших виртуальных горшков даёт порой одинаковые «черепки», и тогда удаление дубликатов может снизить сложность на несколько порядков.

10.6. Возможно ли решение?

Это первое, что нас интересует. Ответ даёт функция **TestCoverOrDissect** (здесь и далее обсуждаем модуль **Assembly**). Идея состоит в последовательном объединении элементов исходного буфера до тех пор, пока либо получим требуемый универсум, либо исчерпаем буфер. Если не получилось накопить универсум, то решение предстоящих задач ЗНР и ЗНП невозможно (листинг 10-2).

Листинг 10-2 — Проверка возможности решения ЗНР и ЗНП

```
function TestCoverOrDissect(aUniv: TSet;      // универсум
                            aBuf: TBuffer  // буфер подмножеств
                           ): boolean;
var  CS : TCostSet;  // очередное подмножество с оценкой
    Sum : TSet;      // накопитель универсума
      i : integer;
begin
  Result:= false;
  Sum:= CreateSet;
  // Суммируем подмножества буфера до тех пор,
  // пока либо сумма достигнет универсума, либо исчерпается буфер
  for i:=1 to aBuf.GetCount do begin
    CS:= aBuf.GetByIndex(i) as TCostSet;
    Sum.Add(CS.mSet);
    if Sum.Compare(aUniv) in [cmpEq, cmpGreate] then begin
      Result:= True;
      Break;
    end;
  end;
  Sum.Free;
end;
```

Отметим, что хотя результат **TRUE** гарантирует существование хотя бы одного *покрытия*, он не гарантирует наличие *разбиения*.

10.7. Удаление дубликатов

Здесь на основе исходного буфера создадим новый, где каждый «черепок» будет уникальным. Если несколько элементов буфера содержат одинаковые подмножества, то в результирующий буфер попадёт только один из этих элементов, причём самый «дешёвый» (листинг 10-3). Обратите внимание на то, что функция создаёт новый буфер, а исходный остаётся нетронутым.

Листинг 10-3 — Просеивание дубликатов большей или равной стоимости

```
function GenFilter(aBuf: TBuffer): TBuffer;
var Ci, Cj : TCostSet;
      i, j : integer;
      Comp : TCompare;
begin
  // Предварительный сброс флажков:
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    Ci.mFlag:= false;
  end;
```

```
  // Установка флажков для пропускаемых элементов
  // Отмечаем совпадающие подмножества с бОльшнй либо равной ценой
  for i:= 1 to aBuf.GetCount-1 do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    if Ci.mFlag then Continue;              // если обработан
    for j:= i+1 to aBuf.GetCount do begin
      Cj:= aBuf.GetByIndex(j) as TCostSet;
      if Cj.mFlag then Continue;            // если обработан
      Comp:= Ci.mSet.Compare(Cj.mSet);      // сравниваем подмножества
      if Comp = cmpEq then begin
        // Если сопадают, сравниваем цены:
        if Ci.mCost >= Cj.mCost
          then Ci.mFlag:= true    // i-й будет пропущен
          else Cj.mFlag:= true;   // j-й будет пропущен
      end;
    end;
  end;

  Result:= TBuffer.Create;
  // Перепись в буфер результата элементов со сброшенным флажком
  // Прокручиваем циклически исходный буфер aBuf
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.Get as TCostSet;
    aBuf.Put(Ci);
    // Если флаг не установлен, то заносим в буфер результата
    if not Ci.mFlag  then Result.Put(Ci);
  end;
end;
```

10.8. Подготовка блоков

Итак, убедившись в том, что решение ЗНР *вероятно* возможно, и отсеяв ненужные дубликаты, подумаем о разумном переборе вариантов.

Учтём то обстоятельство, что *разбиение* может содержать лишь по одному элементу универсума (подмножества разбиения не должны пересекаться). Пусть множество-универсум составляют элементы **{ A,B,C,D... }**. Создадим изначально пустой накопитель универсума. Поместив в этот накопитель любой «обломок», содержащий элемент **A** (и, вероятно, другие элементы), можно затем не рассматривать в качестве кандидатов иные подмножества, содержащие элемент **A**. Тем отсекается много заведомо негодных вариантов. Если в накопителе универсума уже содержится элемент **B**, то не надо рассматривать «осколки», содержащие этот элемент, а иначе рассмотреть подмножества, содержащие элемент **B**, и не содержащие **A**. Так же поступим со всеми последующими элементами универсума. Эти рассуждения подводят к мысли предварительно сгруппировать «черепки» следующим образом.

Возьмём столько пустых коробок, сколько элементов содержится в универсуме. Пометим коробки именами элементов: **A,B,C** и т.д.. Затем разложим черепки из мешка в коробки следующим образом. В первую сложим все обломки, содержащие элемент **A**, и тогда в мешке их больше не останется. Во вторую коробку из оставшихся обломков сложим те, что содержат элемент **B**. Так же поступим с прочими обломками, пока мешок не опустеет. Обратите внимание на

тенденцию: в последующих коробках количество черепков, как правило, уменьшается, вплоть до того, что некоторые из них останутся пустыми.

Объект, представляющий собой эту коробку, будет объявлен и реализован классом «блок»:

Листинг 10-4 — Блок — класс множества для решения ЗНР и ЗНП

```
type
  TSetBlock = class (TSetList)
    mLabel : TItem;     // Метка блока
    constructor Create(aLabel: TItem);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: Text); override;
  end;

constructor TSetBlock.Create(aLabel: TItem);
begin
  inherited Create;
  mLabel:= aLabel;
end;

function TSetBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;  // порядок элементов совпадает с порядком вставки
end;

procedure TSetBlock.Print(var aFile: Text);
begin
  mLabel.Print(aFile);
  inherited;
end;
```

Обратите внимание: метод сравнения **Compare** всегда возвращает значение **cmpLess**, поэтому внутри множества блоки располагаются в порядке их вставки. Теперь покажем функцию, формирующую блоки (листинг 10-5).

Листинг 10-5 — Создание блоков для поиска разбиений

```
// aSet  - универсум
// aBuf  - совокупность разных его подмножеств (оценённых)

function GenDissectBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
var i, j: integer;
    t: TItem;          // очередной элемент из универсума
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet;                   // создаём множество-результат
  for i:=1 to aUniv.GetCount do begin  // цикл по элементам универсума
    if aBuf.GetCount=0 then Break;      // стоп при исчерпании буфера
    t:= aUniv.GetItem(i);          // выбираем очередной элемент универсума
    Block:= TSetBlock.Create(t); // и создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.Get as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)        // если оно содержит текущий элемент
        then Block.Insert(S)    // то помещаем подмножество S в блок
```

```
        else aBuf.Put(S);        // а иначе возвращаем подмножество в буфер
    end;
    if Block.GetCount>0          // если блок не пуст
      then Result.Insert(Block) // помещаем его в множество
      else Block.Free;           // а иначе удаляем пустой
  end;
  aBuf.Free;  // освобождаем буфер
end;
```

Продемонстрируем результаты создания вышеупомянутых наборов данных и блоков. Изначально универсум представляет собой множество из 15-ти элементов:

```
{ a b c d e f g h i j k l m n o } : 15
```

Вызов **`GenSubsRand(Univers,100,0,40)`** создал сотню представленных ниже случайных «осколков» этого универсума:

```
{ a } : 1
{ c d f g l } : 5
{ j m } : 2
{ d m } : 2
{ h j l o } : 4
{ a } : 1
{ g } : 1
{ i } : 1
{ k } : 1
{ b e h m } : 4
{ d i j k o } : 5
{ l } : 1
{ a c f g n } : 5
{ h o } : 2
{ d } : 1
{ e k } : 2
{ a f l } : 3
{ g i m } : 3
{ b } : 1
{ a } : 1
{ k } : 1
{ d i } : 2
{ f o } : 2
{ j n } : 2
{ c g } : 2
{ a l m o } : 4
{ d e g i n } : 5
{ o } : 1
{ b d i k l } : 5
{ h n } : 2
{ e f } : 2
{ j } : 1
{ i m } : 2
{ j } : 1
```

```
{ e } : 1
{ b f } : 2
{ h i l } : 3
{ d e j m o } : 5
{ a } : 1
{ c d k l n } : 5
{ g j } : 2
{ d f } : 2
{ m } : 1
{ i } : 1
{ b g k } : 3
{ h m } : 2
{ c e o } : 3
{ l } : 1
{ j } : 1
{ h i k l o } : 5
{ a d g j n } : 5
{ b m } : 2
{ e f } : 2
{ c } : 1
{ g l n } : 3
{ c d m } : 3
{ a g j m } : 4
{ i } : 1
{ c h n } : 3
{ f k } : 2
{ a } : 1
{ f l o } : 3
{ j } : 1
{ d e h l o } : 5
{ a c m } : 3
{ g } : 1
{ b f i k n } : 5
{ e i j } : 3
{ f l } : 2
```

```
{ d } : 1
{ b c f h i } : 5
{ a g j k } : 4
{ l o } : 2
{ c i l n } : 4
{ b j k } : 3
{ f o } : 2
{ a d e g h } : 5
{ m } : 1
{ b g } : 2
{ d n } : 2
{ a j } : 2
{ h m } : 2
{ c } : 1
{ a b d h m } : 5
{ j } : 1
{ b m } : 2
{ a f h i } : 4
{ d e l n o } : 5
{ c g k } : 3
{ c g k } : 3
{ h i } : 2
{ o } : 1
{ b d l } : 3
{ e } : 1
{ c d j k o } : 5
{ e f h i m } : 5
{ g } : 1
{ b l } : 2
{ f g h i l } : 5
{ e j m n } : 4
```

Вызов функции **GenCostItems(0, SubsBuf)** создал набор тех же подмножеств со случайными ценами:

```
Cost=   6  { a } : 1
Cost=  10  { c d f g l } : 5
Cost=  36  { j m } : 2
Cost=  88  { d m } : 2
Cost=  21  { h j l o } : 4
Cost=  94  { a } : 1
Cost=  59  { g } : 1
Cost=  98  { i } : 1
Cost=  79  { k } : 1
Cost=  39  { b e h m } : 4
Cost=   3  { d i j k o } : 5
Cost=   3  { l } : 1
Cost=  61  { a c f g n } : 5
Cost=  78  { h o } : 2
Cost=  47  { d } : 1
Cost=  51  { e k } : 2
Cost=  74  { a f l } : 3
Cost=  93  { g i m } : 3
Cost=  28  { b } : 1
Cost=  16  { a } : 1
Cost=  14  { k } : 1
Cost=  99  { d i } : 2
Cost=  14  { f o } : 2
Cost=  34  { j n } : 2
Cost=  16  { c g } : 2
Cost=  42  { a l m o } : 4
Cost=  50  { d e g i n } : 5
Cost=  11  { o } : 1
Cost=  38  { b d i k l } : 5
Cost=  30  { h n } : 2
Cost=  28  { e f } : 2
Cost=   7  { j } : 1
Cost=  30  { i m } : 2
Cost=  84  { j } : 1
Cost=  24  { e } : 1
Cost=  38  { b f } : 2
Cost=  58  { h i l } : 3
Cost=   6  { d e j m o } : 5
Cost=  27  { a } : 1
Cost=  40  { c d k l n } : 5
Cost=   8  { g j } : 2
Cost=  76  { d f } : 2
Cost=  42  { m } : 1
Cost=   3  { i } : 1
Cost=  71  { b g k } : 3
Cost=  52  { h m } : 2
Cost=  44  { c e o } : 3
Cost=  72  { l } : 1
Cost=   1  { j } : 1
Cost=  10  { h i k l o } : 5
Cost=  24  { a d g j n } : 5
Cost=  53  { b m } : 2
```

```
Cost=  80  { e f } : 2
Cost=  62  { c } : 1
Cost=  27  { g l n } : 3
Cost=  83  { c d m } : 3
Cost=  48  { a g j m } : 4
Cost=  46  { i } : 1
Cost=   5  { c h n } : 3
Cost=  67  { f k } : 2
Cost=  67  { a } : 1
Cost=  43  { f l o } : 3
Cost=  24  { j } : 1
Cost=  58  { d e h l o } : 5
Cost=  35  { a c m } : 3
Cost=  10  { g } : 1
Cost=  85  { b f i k n } : 5
Cost=  58  { e i j } : 3
Cost=  84  { f l } : 2
Cost=   3  { d } : 1
Cost=  67  { b c f h i } : 5
Cost=  67  { a g j k } : 4
Cost=  84  { l o } : 2
Cost=  36  { c i l n } : 4
Cost=  82  { b j k } : 3
Cost=  80  { f o } : 2
Cost=  91  { a d e g h } : 5
Cost=   8  { m } : 1
Cost=  27  { b g } : 2
Cost=  51  { d n } : 2
Cost=  74  { a j } : 2
Cost=  78  { h m } : 2
Cost=  45  { c } : 1
Cost=  39  { a b d h m } : 5
Cost=  17  { j } : 1
Cost=  89  { b m } : 2
Cost=  47  { a f h i } : 4
Cost=  14  { d e l n o } : 5
Cost=  92  { c g k } : 3
Cost=  62  { c g k } : 3
Cost=  59  { h i } : 2
Cost=   9  { o } : 1
Cost=  45  { b d l } : 3
Cost=   4  { e } : 1
Cost=  70  { c d j k o } : 5
Cost=  14  { e f h i m } : 5
Cost=  20  { g } : 1
Cost=  10  { b l } : 2
Cost=  35  { f g h i l } : 5
Cost=  62  { e j m n } : 4
```

После отсева дубликатов в наборе осталось 87 элементов, из которых были сформированы блоки, указанные втабл. 10-1 (в скобках указаны мощности блоков).

Табл. 10-1 — Разбиение набора подмножеств на блоки

Метка	Блок	Метка	Блок
a **(12)**	Cost= 24 { a d g j n } : 5 Cost= 6 { a } : 1 Cost= 39 { a b d h m } : 5 Cost= 42 { a l m o } : 4 Cost= 35 { a c m } : 3 Cost= 47 { a f h i } : 4 Cost= 48 { a g j m } : 4 Cost= 61 { a c f g n } : 5 Cost= 67 { a g j k } : 4 Cost= 91 { a d e g h } : 5 Cost= 74 { a f l } : 3 Cost= 74 { a j } : 2	*h* **(7)**	Cost= 10 { h i k l o } : 5 Cost= 21 { h j l o } : 4 Cost= 30 { h n } : 2 Cost= 58 { h i l } : 3 Cost= 52 { h m } : 2 Cost= 59 { h i } : 2 Cost= 78 { h o } : 2
b **(12)**	Cost= 10 { b l } : 2 Cost= 38 { b d i k l } : 5 Cost= 39 { b e h m } : 4 Cost= 67 { b c f h i } : 5 Cost= 27 { b g } : 2 Cost= 45 { b d l } : 3 Cost= 85 { b f i k n } : 5 Cost= 38 { b f } : 2 Cost= 71 { b g k } : 3 Cost= 53 { b m } : 2 Cost= 82 { b j k } : 3 Cost= 28 { b } : 1	*i* **(2)**	Cost= 3 { i } : 1 Cost= 30 { i m } : 2
c **(10)**	Cost= 5 { c h n } : 3 Cost= 10 { c d f g l } : 5 Cost= 40 { c d k l n } : 5 Cost= 16 { c g } : 2 Cost= 36 { c i l n } : 4 Cost= 70 { c d j k o } : 5 Cost= 44 { c e o } : 3 Cost= 62 { c g k } : 3 Cost= 83 { c d m } : 3 Cost= 45 { c } : 1	*j* **(3)**	Cost= 1 { j } : 1 Cost= 34 { j n } : 2 Cost= 36 { j m } : 2
d **(10)**	Cost= 3 { d i j k o } : 5 Cost= 6 { d e j m o } : 5 Cost= 14 { d e l n o } : 5 Cost= 3 { d } : 1 Cost= 50 { d e g i n } : 5 Cost= 58 { d e h l o } : 5 Cost= 51 { d n } : 2 Cost= 76 { d f } : 2 Cost= 88 { d m } : 2 Cost= 99 { d i } : 2	*k* **(1)**	Cost= 14 { k } : 1
e **(6)**	Cost= 14 { e f h i m } : 5 Cost= 4 { e } : 1 Cost= 28 { e f } : 2 Cost= 62 { e j m n } : 4 Cost= 58 { e i j } : 3 Cost= 51 { e k } : 2	*l* **(2)**	Cost= 3 { l } : 1 Cost= 84 { l o } : 2
f **(5)**	Cost= 35 { f g h i l } : 5 Cost= 14 { f o } : 2 Cost= 43 { f l o } : 3 Cost= 67 { f k } : 2 Cost= 84 { f l } : 2	*m* **(1)**	Cost= 8 { m } : 1
g **(4)**	Cost= 8 { g j } : 2 Cost= 27 { g l n } : 3 Cost= 10 { g } : 1 Cost= 93 { g i m } : 3	*o* **(1)**	Cost= 9 { o } : 1

Табл. 10-1 не содержит блока с меткой «**n**», поскольку все подмножества с этим элементом оказались в предшествующих блоках. Во-вторых, обратите внимание на порядок следования подмножеств внутри каждого блока, — он

определён методом **TCostSet.Compare**, за полным текстом которого отсылаю в модуль **Root**. Этот метод ранжирует подмножества внутри блока в порядке возрастания *удельной стоимости* его элементов, что выражено следующим оператором:

```
    if mCost * (arg as TCostSet).mSet.GetCount <
       (arg as TCostSet).mCost * mSet.GetCount
      then  Result:= cmpLess
      else  ...
```

Здесь при сравнении удельных стоимостей трудоёмкое деление с плавающей точкой заменено быстрым умножением целых чисел. То есть, вместо выражений

(A/B) < (C/D)

сравниваются выражения

(A•D) < (C•B)

Теперь подмножества в блоках расположились в порядке возрастания удельной стоимости элементов. Для примера, первым трём элементам первого блока присущи следующие удельные стоимости:

```
      24/5 = 4.8
       6/1 = 6.0
      39/5 = 7.8
```

Указанный порядок следования подмножеств позволит прибегнуть к *поиску с возвратом*. Размещение «дешёвых» подмножеств в начале блока повышает шансы раннего обнаружения минимального разбиения.

10.9. Поиск с возвратом

Итак, распределив осколки по коробкам, мы тем самым на много порядков снизили сложность задачи. Теперь максимальное число возможных вариантов перебора выражается произведением мощностей блоков, что для приведенной выше таблицы даёт число **1.45E+08**. Не так уж мало для скромного набора исходных данных, и это подвигает нас к дальнейшему улучшению алгоритма — применению *поиска с возвратом*, знакомство с которым состоялось в главе 8.

Поскольку целью подбора является минимальная стоимость разбиения, текущую накопленную стоимость примем за критерий возврата. Учредим две переменные: в первой будет накапливаться текущая стоимость промежуточных (неполных) разбиений, а во второй — храниться стоимость лучшего разбиения, найденного на текущий момент. Изначально, когда накопленный универсум пуст, текущая стоимость равна нулю. Стоимость наилучшего разбиения пока неизвестна, и вначале занесём туда заведомо большое число (условную бесконечность **MaxInt**).

Далее приступим к перебору подмножеств, начав с первого блока. Прежде всего, проверим, содержится ли метка блока в накопленном универсуме. Если да, то блок пропускаем и рекурсивно переходим к следующему. Понятно, что первый блок не миновать, переберём его элементы. Взяв очередное подмножество, смотрим, не будет ли его стоимость вкупе с уже накопленной меньше минимальной. При поиске первого из возможных разбиений это условие всегда выполняется, потому проверяем следующее условие: пересекается ли очередное подмножество с накопленным универсумом. Если нет, то присоединяем его к универсуму и включаем в разбиение, после чего проверяем полноту накопленного универсума. Если накопленный универсум полон (совпадает с заданным в параметре функции), то текущее разбиение и его стоимость запоминаются в качестве лучших на данный момент. Если же накопитель универсума не полон, рекурсивно переходим к обработке следующего блока, и так вплоть до последнего.

Функция, реализующая описанный алгоритм, представлена ниже (листинг 10-6). В основном теле сначала проверяется возможность решения: если оно невозможно, работа завершается. Затем удаляются дубликаты, создаются блоки и несколько вспомогательных переменных, впоследствии вызывается локальная рекурсивная процедура обработки блоков, которая и выполняет основную работу.

Отметим, что если предварительная оценка исходного набора подмножеств оставляет шанс найти разбиение, оно не обязательно существует, и тогда функция вернёт пустое множество. Если же равноценных разбиений окажется несколько, функция вернёт первое найденное.

Листинг 10-6 — Решение задачи о наименьшем разбиении (ЗНР)

```
function CollectMinDissect(aUniv: TSet;     // целевой универсум
                           aBuf: TBuffer    // буфер с подмножествами
                           ): TCostSet;
var Buf     : TBuffer;     // Отфильтрованная копия входного буфера
    Univ    : TSet;        // Накопитель универсума
    BestCost: integer;     // Лучшая стоимость
    Blocks  : TSet;        // Вспомогательные блоки
    Res : TCostSet;        // Исходное пустое разбиение
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блока

  procedure BlockHandle(aBlock: integer;  // номер блока
                        aCost: integer;   // накопленная сумма
                        aSum: TSet;       // накопленные элементы
                        aRes: TCostSet    // накопленные подмножества
                        );
  var Block: TSetBlock; // текущий блок
      Sum : TSet;       // Накопленное множество элементов
      Res : TCostSet;   // Накопленное разбиение (множество подмножеств)
       CS : TCostSet;   // очередное оценённое подмножество из блока
  begin
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Если элемент текущего блока содержится в накопителе
    if aSum.Exist(Block.mLabel) then begin
      // то пропускаем этот блок и входим в следующий (если не последний)
      if aBlock < Blocks.GetCount then begin
        BlockHandle(aBlock+1, aCost, aSum, aRes) // вход в следующий блок
```

```
      end
    end else begin
      // перебор подмножеств текущего блока
      CS:= Block.GetFirst as TCostSet;
      {$ifdef _TRACE_} TestCount:= TestCount+Block.GetCount;{$endif}
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        // Если сумма не превышена
        // и CS не пересекается с накопл. универсумом, то обрабатываем
        if ((aCost+CS.mCost) < BestCost) and
           not aSum.TestIntersect(CS.mSet) then begin
          // Пристраиваем к разбиению очередной блок:
          Sum:= aSum.Copy as TSet;      // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного разбиения
          Sum.Add(CS.mSet);      // накапливаем универсум
          Res.Insert(CS);        // пристраиваем к разбиению очередной блок
          // Проверяем, накоплен ли целевой универсум
          if Sum.GetCount = aUniv.GetCount then begin
            // Здесь найдено очередное разбиение.
            // Поскольку оно лучше прежнего,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;    // новая лучшая сумма
            Result.Free;                    // удаляем прежнее разбиение
            Result:= Res.Copy as TCostSet; // и создаём новое лучшее
            {$ifdef _TRACE_ -- трассировка }
            Inc(TestNumber);  // порядковый номер результата
            Writeln(TestNumber:3, BestCost:7, aBlock:8);
            {$endif}
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res)
          end; // else
          // Перед выбором следующего подмножества в блоке освобождаем:
          // возвращаем из стека предыдущие значения
          Res.Free;    // накопленное разбиение
          Sum.Free;    // накопленный универсум
        end; // if
        CS:= Block.GetNext as TCostSet;
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------
begin

  Result:= TCostSet.CreateEmpty;  // Создаём пустое множество-разбиение

  // Проверяем возможность найти разбиение, иначе выход

  if not TestCoverOrDissect(aUniv, aBuf) then Exit;

  Buf:= GenFilter(aBuf);  // Создаём фильтрованную копию входного буфера
  {$ifdef _TRACE_ -- трассировка }
    Writeln('--- Collect Min Dissect ---');
    // Печать результата фильтрации:
    Writeln('aBuf / Buf = ', aBuf.GetCount, ' / ', Buf.GetCount);
  {$endif}

  Blocks:= GenDissectBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  // Подготовка переменных:
  BestCost:= MaxInt;          // Лучшая сумма
  Univ := CreateSet;          // Накопитель универсума
```

```
  Res:= TCostSet.CreateEmpty; // Создаём пустое множество-разбиение

  // Рекурсивная обработка блоков:
  BlockHandle(1, 0, Univ, Res);

  // Очистка:
  Res.Free;                   // Исходное пустое разбиение
  Univ.Free;                  // Освобождаем накопитель
  Blocks.ClrAndDestroy;       // Удаляем блоки
  Blocks.Free;                // и множество блоков

  {$ifdef _TRACE_ -- трассировка }
     Writeln('TestDiff  = ', TestDiff :11);
     Writeln('TestCount = ', TestCount :11);
     if TestCount>0
      then Writeln('Diff/Count= ', TestDiff / TestCount :11);
  {$endif}

end;
```

10.10. Тестирование и статистика

Для тестирования решения ЗНР служит следующая программа.

Листинг 10-7 — Программа для тестирования ЗНР

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
    Univers : TSet;     // Универсум - исходное множество объектов
    SubsBuf : TBuffer;  // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;  // Буфер подмножеств с оценками
    Res: TCostSet;      // Результат - минимальное разбиение

begin
    // Создание универсума из 15-ти элементов
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - - -');
    // Генерация случайных разбиений универсума
    SubsBuf:= GenSubsRand(Univers, 100, 0, 40);
    Writeln('- - - - - - - - - - - - - -');
    {
     Добавление к подмножествам оценок,
     Первый параметр определяет способ назначения оценок:
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
```

```
    }
    CostBuf:= GenCostItems(0, SubsBuf);
    SubsBuf.Free;  // исхлдный буфер уже не нужен
    CostBuf.Expo;
    Writeln('- - - - - - - - - - - - -');
    // Формирование минимального разбиения (решение ЗНР):
    Res:= CollectMinDissect(Univers, CostBuf);
    Res.Expo;
    // Очистка памяти:
    Res.Free;
    CostBuf.ClrAndDestroy;
    CostBuf.Free;
    Univers.ClrAndDestroy;
    Univers.Free;
    Write('OK');  Readln;
end.
```

Результат работы программы выводится в следующем виде:

```
Cost= 70
{
 Cost=   3    { d i j k o } : 5
 Cost=   5    { c h n } : 3
 Cost=  10    { b l } : 2
 Cost=   6    { a } : 1
 Cost=   8    { m } : 1
 Cost=  10    { g } : 1
 Cost=  28    { e f } : 2
} : 7
```

В первой строке показана стоимость разбиения, а в последующих — множество его подмножеств. Здесь можно вывести и дополнительную статистику, если в модуле **Assembly** определить имя **_TRACE_** так, как показано ниже:

```
{$define _TRACE_}
```

Тогда в ходе поиска разбиения будет выведена дополнительная информация в следующем виде:

```
aBuf / Buf = 100 / 76
Num  BestCost  Block
  1    200       9
  2    116      14
  3     70      13
TestDiff   =  1.45E+0008
TestCount =  1.03E+0003
Diff/Count=  1.41E+0005
```

Вначале показано количество элементов в буфере до и после фильтрации дубликатов. Затем тремя колонками выводятся номер найденного разбиения, его стоимость и номер блока, где оно обнаружено. В конце приводятся три числа:

- **TestDiff** — теоретическая сложность полного перебора, вычисленная через произведение мощностей блоков (количество вариантов перебора);
- **TestCount** — количество реально обработанных подмножеств;

- **`Diff/Count`** — соотношение этих чисел (коэффициент выигрыша).

Интересно проследить зависимость трудоёмкости задачи от стратегии назначения цен (см. табл. 10-2). В таблице приведены данные для множества из 15-ти элементов и массива из 76 подмножеств, оставшихся после фильтрации дубликатов. Очевидно, что компьютеру легче справиться с элементами, цена которых задана случайно. Чуть сложнее даётся массив с одинаковыми ценами, а трудней всего такой, где цена совпадает с мощностью подмножества (стратегия 3).

Табл. 10-2 — Влияние ценовой стратегии

Параметр стратегии	Стратегия (Cost=)	TestDiff	TestCount	Diff/Count
0	1 + Random(99)	1.45E+08	1.03E+03	1.41E+05
1	1	1.45E+08	3.92E+03	3.71E+04
2	1 + (n div 2) + Random(n)	1.45E+08	1.63E+04	8.89E+03
3	n	1.45E+08	5.75E+04	2.52E+03
4	1 + (n-1)*(n-1)	1.45E+08	2.69E+04	5.39E+03

Разумеется, что приведенные выше статистические оценки следует воспринимать лишь как грубый ориентир, более точное сравнение возможно лишь испытаниями с засечкой времени.

10.11. Итоги

- К сборке универсума из подмножеств сводятся многие практически важные задачи.
- Множество подмножеств, в сумме дающих универсум, является либо *разбиением* (отдельные его подмножества не пересекаются), либо *покрытием* (подмножества могут пересекаться). Разбиение является частным случаем покрытия.
- Практический интерес представляют *минимальные* разбиения и покрытия, причём задача может ставиться двояко: ищется либо минимальное *количество* подмножеств, либо набор с минимальной *стоимостью*. Первый вариант задачи решается через второй путём назначения подмножествам единичных цен.
- Задачи поиска минимальных разбиений (ЗНР) и покрытий (ЗНП) обладают экспоненциальной сложностью.
- Для снижения сложности задач в исходном наборе подмножеств удаляют дубликаты, *разбивают на блоки*, и применяют *поиск с возвратом*.

10.12. Что почитать

№		Автор(ы)	Название	Главы, страницы
√	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 53
√	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 11
Задача о наименьшем покрытии (ЗНП)

Продолжим сборку множеств, и займёмся поиском наименьших *покрытий* (ЗНП). Рассмотрим пример, где капитан судна нанимает экипаж, выбирая среди матросов, каждый из которых владеет одной или несколькими профессиями. Если требование компактности экипажа первично, он постарается набрать минимум матросов, но так, чтобы покрыть все нужды в специалистах. Если же матросы требуют для себя разные уровни оплаты, а цель капитана — экономия средств, то он наберёт по возможности «дешёвый» экипаж, покрывающий, однако, всю потребность в профессионалах. Этим двум целям соответствуют два варианта задачи о наименьшем покрытии: 1) найти минимальное *количество* подмножеств покрытия, и 2) найти набор подмножеств с *минимальной общей стоимостью*.

Решение ЗНП во многом похоже на поиск минимального разбиения (ЗНР): здесь тоже предварительно выясняют возможность покрытия, а также удаляют ненужные дубликаты подмножеств. Но на последующих шагах учитывают то, что подмножествам, входящим в покрытие, дозволено пересекаться, а это влияет и на формирование блоков, и на их перебор.

11.1. Составление блоков

При «реставрации» ковра его обрывки можно частично накладывать друг на друга, пересекать. И потому содержимое известных по предыдущей главе меченых коробок с лоскутами должно быть несколько иным. В первую сложим *все* клочки, в состав которых входит элемент **A**, во вторую — *все* клочки с элементом **B**, и т.д. Понятно, что реальный клочок может попасть только в одну из коробок. Если клочок содержит и **A,** и **B**, то он попадёт только в коробку с меткой **A**. Но мы работаем с информацией, и указатели на объекты позволяют помещать их сразу в несколько «коробок».

Итак, предварительная подготовка меченых блоков даст нам множество «коробок», число которых совпадает с мощностью универсума. Если встречаемость элементов в разных подмножествах равновероятна, то и мощность блоков будет примерно одинакова (вспомним, что для ЗНР она в целом убывала). Всё потому, что одно подмножество вставляется сразу в несколько блоков, и это влечёт чудовищную трудоёмкость перебора в ЗНП. Отметим, что подобно ЗНР, подмножества внутри блоков расположим в порядке не убывания их удельной стоимости (дешёвые в начале), а это повышает вероятность раннего обнаружения наилучшего покрытия.

В следующем листинге представлена функция, генерирующая множество меченых блоков для поиска наименьшего покрытия.

Листинг 11-1 — Создание блоков для поиска минимального покрытия

```
function GenCoverBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
var i, j: integer;
    t: TItem;          // очередной элемент из aSet
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet; // создаём буфер результата
  for i:=1 to aUniv.GetCount do begin // Цикл по элементам универсума
    t:= aUniv.GetItem(i);         // выбираем очередной элемент универсума
    Block:= TSetBlock.Create(t); // создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.GetByIndex(j) as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)       // если оно содержит текущий элемент
        then Block.Insert(S)   // то помещаем подмножество S в блок
    end;
    Result.Insert(Block) // вставляем блок в результирующее множество
  end;
end;
```

Ниже показаны четыре из пятнадцати сформированных блоков:

```
a  {
 Cost=  24      { a d g j n } : 5
 Cost=   6      { a } : 1
 Cost=  39      { a b d h m } : 5
 Cost=  42      { a l m o } : 4
 Cost=  35      { a c m } : 3
 Cost=  47      { a f h i } : 4
 Cost=  48      { a g j m } : 4
 Cost=  61      { a c f g n } : 5
 Cost=  67      { a g j k } : 4
 Cost=  91      { a d e g h } : 5
 Cost=  74      { a f l } : 3
 Cost=  74      { a j } : 2
  } : 12

 b  {
 Cost=  10      { b l } : 2
 Cost=  38      { b d i k l } : 5
 Cost=  39      { a b d h m } : 5
 Cost=  39      { b e h m } : 4
 Cost=  67      { b c f h i } : 5
 Cost=  27      { b g } : 2
 Cost=  45      { b d l } : 3
 Cost=  85      { b f i k n } : 5
 Cost=  38      { b f } : 2
 Cost=  71      { b g k } : 3
 Cost=  53      { b m } : 2
 Cost=  82      { b j k } : 3
 Cost=  28      { b } : 1
  } : 13

 c  {
 Cost=   5      { c h n } : 3
 Cost=  10      { c d f g l } : 5
 Cost=  40      { c d k l n } : 5
```

```
 Cost=  16      { c g } : 2
 Cost=  36      { c i l n } : 4
 Cost=  35      { a c m } : 3
 Cost=  61      { a c f g n } : 5
 Cost=  67      { b c f h i } : 5
 Cost=  70      { c d j k o } : 5
 Cost=  44      { c e o } : 3
 Cost=  62      { c g k } : 3
 Cost=  83      { c d m } : 3
 Cost=  45      { c } : 1
  } : 13

 d  {
 Cost=   3      { d i j k o } : 5
 Cost=   6      { d e j m o } : 5
 Cost=  10      { c d f g l } : 5
 Cost=  14      { d e l n o } : 5
 Cost=   3      { d } : 1
 Cost=  24      { a d g j n } : 5
 Cost=  38      { b d i k l } : 5
 Cost=  39      { a b d h m } : 5
 Cost=  40      { c d k l n } : 5
 Cost=  50      { d e g i n } : 5
 Cost=  58      { d e h l o } : 5
 Cost=  70      { c d j k o } : 5
 Cost=  45      { b d l } : 3
 Cost=  91      { a d e g h } : 5
 Cost=  51      { d n } : 2
 Cost=  83      { c d m } : 3
 Cost=  76      { d f } : 2
 Cost=  88      { d m } : 2
 Cost=  99      { d i } : 2
  } : 19
```

11.2. Поиск с возвратом

Перебор подмножеств внутри блоков подобен перебору при поиске разбиения. Создаются те же вспомогательные переменные, и точно так же перебираются подмножества: «по горизонтали» — внутри блока, и «по вертикали» — рекурсивным входом в следующий блок. Если накопитель универсума уже содержит какой-то элемент, то блок, меченный этим элементом, пропускается. Единственное отличие состоит в отсутствии проверки на пересечение очередного подмножества с накопленным универсумом (листинг 11-2).

Листинг 11-2 — Задача о наименьшем покрытии (ЗНП)

```
function CollectMinCover(aUniv: TSet;       // целевой универсум
                         aBuf: TBuffer    // буфер подмножеств с оценками
                        ): TCostSet;
var Buf: TBuffer;        // отфильтрованный входной буфер
    Univ     : TSet;       // Накопленный универсум
    BestCost: integer;     // Лучшая сумма
    Blocks   : TSet;       // Вспомогательные блоки
    Res : TCostSet;        // Исходное пустое покрытие
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блоков
```

```
  procedure BlockHandle(aBlock, aCost: integer; aSum: TSet; aRes: TCostSet);
  var Block : TSetBlock; // текущий блок
      Sum : TSet;        // Накопленный универсум
      Res : TCostSet;    // Накопленное покрытие (множество подмножеств)
       CS : TCostSet;    // очередное оценённое подмножество из блока
  begin
    // Выбираем блок из множества блоков
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Проверяем наличие метки текущего блока в накопителе универсума
    if aSum.Exist(Block.mLabel) then begin
      // Метка уже есть, пропускаем данный блок
      // и входим в следующий, если текущий не последний
      if aBlock<Blocks.GetCount then BlockHandle(aBlock+1,aCost,aSum,aRes)
    end else begin
      // Начало обработки очередного блока
      CS:= Block.GetFirst as TCostSet; // первое подмножество блока
      // Перебор подмножеств блока
      {$ifdef _TRACE_} TestCount:= TestCount+Block.GetCount;{$endif}
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        if (aCost+CS.mCost) < BestCost then begin // сумма меньше ?
          Sum:= aSum.Copy as TSet;      // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного покрытия
          Sum.Add(CS.mSet);      // накапливаем универсум
          Res.Insert(CS);        // и покрытие
          // Проверяем, накоплен ли целевой универсум (покрытие):
          if Sum.GetCount = aUniv.GetCount then begin
            // Найдено очередное покрытие,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;    // новая лучшая сумма
            Result.Free;                    // удаляем прежнее покрытие
            Result:= Res.Copy as TCostSet; // и сохраняем копию нового
            {$ifdef _TRACE_ -- трассировка }
            Inc(TestNumber);  // порядковый номер очередного решения
            Writeln(TestNumber:3, BestCost:7, aBlock:8);
            {$endif}
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res);
          end; // else
          // Перед выбором следующего подмножества из блока удаляем:
          Res.Free;  // накопленное покрытие
          Sum.Free;  // накопленный универсум
        end; // then
        CS:= Block.GetNext as TCostSet;  // следующий элемент блока
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------
begin
  Result:= TCostSet.CreateEmpty; // Создаём пустое покрытие
  // Проверяем, если покрытие недостижимо, то выход:
  if not TestCoverOrDissect(aUniv, aBuf) then Exit;
  Buf:= GenFilter(aBuf); // Создаём фильтрованную копию входного буфера
  {$ifdef _TRACE_ -- трассировка }
    Writeln('--- Collect Min Cover ---');
    // Печать результата фильтрации:
    Writeln('aBuf / Buf = ', aBuf.GetCount, ' / ', Buf.GetCount);
  {$endif}
  Blocks:= GenCoverBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  Buf.Free;                             // Буфер уже не нужен, удаляем
```

```
  {$ifdef _TRACE_ -- трассировка }
    // Для статистики:
    Writeln('Num  BestCost  Block');
    TestNumber:= 0;                    // порядковый номер решения
    TestCount:= 0;                     // количество просмотренных элементов
    TestDiff:= CalcDifficult(Blocks);  // теоретическая сложность
  {$endif}
  Res:= TCostSet.CreateEmpty; // Начальное пустое покрытие
  Univ := CreateSet;          // Накопитель элементов (универсум)
  BestCost:= MaxInt;          // Лучшая сумма
  BlockHandle(1, 0, Univ, Res);  // Рекурсивная обработка блоков
  // Очистка:
  Res.Free;              // Начальное пустое покрытие
  Univ.Free;             // удаляем накопитель элементов (универсум)
  Blocks.ClrAndDestroy; // удаляем блоки
  Blocks.Free;
 {$ifdef _TRACE_ -- трассировка }
    Writeln('TestDiff  = ', TestDiff :11);  // теоретическая сложность
    Writeln('TestCount = ', TestCount:11);  // фактическая сложность
    // Соотношение
    if TestCount>0 then Writeln('Diff/Count= ', TestDiff / TestCount :11);
 {$endif}
end;
```

11.3. Тестирование

Для тестирования ЗНП применена следующая программа:

Листинг 11-3 — Программа для тестирования ЗНП

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
    Univers : TSet;        // Универсум - исходное множество объектов
    SubsBuf : TBuffer;  // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;  // Буфер подмножеств с оценками
    Res: TCostSet;         // Результат - минимальное покрытие
begin
    // Создание универсума
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - -');
    // Генерация случайных разбиений универсума
    SubsBuf:= GenSubsRand(Univers, 100, 0, 40);
    {
     Добавление к подмножествам оценок,
     Первый параметр определяет способ назначения оценок:
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
```

```
     4: cost:= 1 + (n-1)*(n-1);
    }
    CostBuf:= GenCostItems(0, SubsBuf);
    SubsBuf.Free;  // исхлдный буфер уже не нужен
    // Формирование минимального покрытия:
    Res:= CollectMinCover(Univers, CostBuf);
    Res.Expo;
    // Очистка памяти:
    Res.Free;
    CostBuf.ClrAndDestroy;    CostBuf.Free;
    Univers.ClrAndDestroy;    Univers.Free;
    Write('OK');
    Readln;
end.
```

Если в модуле **Assembly** определить имя **_TRACE_**, это повлечёт вывод дополнительной статистики (операторы вывода статистики в листингах не показаны, обратитесь к тексту модуля). Так получены следующие результаты:

```
{ a b c d e f g h i j k l m n o } : 15
- - - - - - - - - - - - - - -
--- Collect Min Cover ---
aBuf / Buf = 100 / 76   -- число элементов до и после удаления дубликатов
Num  BestCost  Block
  1     58       9
  2     56      11
  3     40       6
TestDiff  =  3.53E+0017
TestCount =  9.43E+0002
Diff/Count=  3.74E+0014
Cost= 40
{
 Cost=   3    { d i j k o } : 5
 Cost=   6    { d e j m o } : 5
 Cost=   5    { c h n } : 3
 Cost=  10    { c d f g l } : 5
 Cost=  10    { b l } : 2
 Cost=   6    { a } : 1
} : 6
```

Если сравнить эти результаты с полученными при поиске разбиений (ЗНР), то бросается в глаза чудовищное количество возможных комбинаций: **3.53E+17** против **1.45E+08**. Интересно, также, исследовать влияние ценовой стратегии на трудоёмкость задачи, в табл. 11-1 дан результат этих испытаний.

Табл. 11-1 — Влияние ценовой стратегии

Параметр стратегии	Стратегия (Cost=)	TestDiff	TestCount	Diff/Count
0	1 + Random(99)	3.53E+17	9.43E+02	3.74E+14
1	1	3.53E+17	3.62E+04	9.75E+12
2	1 + (n div 2) + Random(n)	3.53E+17	2.06E+05	1.71E+12
3	n	3.53E+17	7.06E+06	5.00E+10
4	1 + (n-1)*(n-1)	3.53E+17	1.06E+06	3.32E+11

Наибольшая трудоёмкость присуща линейной зависимости цены от мощности подмножества (параметр 3), — здесь программа мечется, что тот персонаж в известной интермедии про раков, выбирая между мелкими по три и крупными по пять. Но вчера.

11.4. Градиентный поиск

Предварительные оценки показывают (испытания это подтвердят), что поиск наилучшего покрытия может недопустимо затянуться. Как быть системам реального времени, где нужны быстрые решения? Можно ли сократить время, частично пожертвовав качеством решения? По отношению к покрытию это значит согласие принять не самый дешёвый набор подмножеств. Пусть он будет немного дороже идеала, но главное — быстро!

Вспомним о *жадном* принципе, который часто применяют в задачах перебора. Применительно к ЗНП его иногда называют *градиентным поиском* или *градиентным спуском*. Последнее напоминает о стратегии горнолыжника, который, не зная трассы, стремится быстрее спуститься с вершины к подножью. В каждой точке трассы он выбирает направление с наибольшим уклоном, поддерживая, таким образом, максимальную скорость. Это не гарантирует ему рекордного результата, поскольку может увеличить длину трассы, что повлечёт потерю времени. Однако что ему остаётся? Его подход не лишён смысла, и мы им воспользуемся.

Прежде всего, отметим, что подготовка блоков для ЗНП остаётся прежней, но обработка их изменится. Вместо многократных «нырков» внутрь блоков и обратно, как это делалось в рекурсивной процедуре, обработаем их всего лишь за один проход «слева направо» и «сверху вниз».

Взяв очередной блок, проверим, содержится ли его метка в накопителе универсума? Если да, то блок игнорируем и переходим к следующему. Если нет, то ищем в блоке одно единственное подмножество, добавление которого к накопителю даст наилучший эффект. Для этого вычислим разность между очередным подмножеством и универсумом и определим удельную стоимость этой прибавки. Так, перебрав все подмножества блока, найдём подмножество, дающее наибольшую прибавку, отнесённую к единице его стоимости, — это подмножество и добавим к накопителю.

Функция градиентного поиска, дающая быстрое, хоть и не всегда идеальное решение ЗНП, представлена ниже.

Листинг 11-4 — Градиентный поиск покрытия

```
function CollectGradCover(aBase: TSet;
                          aBuf: TBuffer
                         ): TCostSet;
var    Buf : TBuffer;    // отфильтрованный входной буфер
      Univ : TSet;       // накопитель универсума
    Blocks : TSet;       // блоки
```

```
     Block : TSetBlock;  // текущий блок
    BestCS : TCostSet;   // текущее лучшее покрытие
  //-------------------------------------------------------------
  function GetBest: TCostSet;
  var      CS : TCostSet;   // очередное подмножество с оценкой
            S : TSet;       // рабочее подмножество
    BestDelta : integer;    // лучшее приращение
     BestCost : integer;    // лучшая цена
            i : integer;
  begin
    Result:= nil;
    S:= CreateSet;  // создаём рабочее множество
    BestDelta:=0;   // лучшая мощность
    BestCost:= 1;   // лучшая цена
    // перебор элементов блока:
    for i:= 1 to Block.GetCount do begin
      CS:= Block.GetItem(i) as TCostSet;  // очередное подмнож. с оценкой
      S.CopyItems(CS.mSet);               // копируем подмножество
      S.Sub(Univ);                        // оставляем приращение множества
      // Если данное приращение лучшее, то запоминаем
      if S.GetCount * BestCost > CS.mCost * BestDelta then begin
        Result:= CS;              // лучшее множество
        BestCost:= CS.mCost;      // лучшая цена
        BestDelta:= S.GetCount;   // лучшее приращение
      end;
    end;
    S.Free;  // удаляем рабочее множество
  end;
  //-------------------------------------------------------------

var    i : Integer;

begin
  Result:= TCostSet.CreateEmpty;  // Покрытие = пустое множество
  // Если покрытие невозможно, вернуть пустой результат
  if not TestCoverOrDissect(aBase, aBuf) then Exit;

  Buf:= GenFilter(aBuf); // Отфильтрованная копия входного буфера
  Blocks:= GenCoverBlocks(aBase, Buf);  // Вспомогательные блоки
  Buf.Free;                       // Буфер уже не нужен
  Univ:= CreateSet;               // Создаём накопитель универсума
  for i:= 1 to Blocks.GetCount do begin
    Block:= Blocks.GetItem(i) as TSetBlock;
    if Univ.Exist(Block.mLabel)  // если метка блока найдена в накопителе
      then Continue;             // то пропускаем блок
    BestCS:= GetBest;            // выбор лучшего подмножества в блоке
    if Assigned(BestCS) then begin
      // Если лучшее подмножество в блоке найдено:
      Univ.Add(BestCS.mSet);     // накапливаем универсум
      Result.Insert(BestCS);     // и покрытие
    end;
  end;
  Univ.Free;              // освободить накопитель универсума
  Blocks.ClrAndDestroy;   // очистить вспомогательные блоки
  Blocks.Free;
end;
```

Для проверки функции, и сравнения градиентного поиска с точным решением ЗНП служит следующая программа:

Листинг 11-5 — Программа для сравнения двух методов поиска покрытия

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
var
    Univers : TSet;      // Универсум - исходное множество объектов
    SubsBuf : TBuffer;   // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;   // Буфер подмножеств с оценками
    Res: TCostSet;       // Результат - минимальное покрытие
begin
    // Создание универсума
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - -');
    // Генерация случайных разбиений универсума
    SubsBuf:= GenSubsRand(Univers, 100, 0, 40);
    {
     Добавление к подмножествам оценок,
     Первый параметр определяет способ назначения оценок:
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
    }
    CostBuf:= GenCostItems(1, SubsBuf);
    SubsBuf.Free;  // исходный буфер уже не нужен
    // Точное формирование минимального покрытия:
    Res:= CollectMinCover(Univers, CostBuf);
    Res.Expo;
    Res.Free;
    Writeln('- - - - - - - - - - - - -');
    // Быстрое формирование минимального покрытия:
    Res:= CollectGradCover(Univers, CostBuf);
    Res.Expo;
    // Очистка памяти:
    Res.Free;
    CostBuf.ClrAndDestroy;     CostBuf.Free;
    Univers.ClrAndDestroy;     Univers.Free;
    Write('OK');
    Readln;
end.
```

Отметим, что «ценовая политика», существенно влияющая на скорость поиска в ЗНП, никак не влияет на градиентный поиск.

11.5. Временные испытания

Теперь сравним быстродействие решённых нами задач поиска минимальных разбиений и покрытий так, как это делают в спорте: засечкой времени. Тем мы учтём все выполняемые операции. Тестирующая программа представлена ниже.

Листинг 11-6 — Программа для сравнения времени решений ЗНР и ЗНП

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
var
    Univers : TSet;      // Универсум - исходное множество объектов
    SubsBuf : TBuffer;   // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;   // Буфер подмножеств с оценками
    Res: TCostSet;       // Результат - минимальное покрытие
    Start, Time : TDateTime;
begin
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - - - - - - - - - - -');
    // Генерация разбиений
    SubsBuf:= GenSubsRand(Univers, 150, 0, 40);
    // Добавление к подмножествам оценок
    {
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
    }
    CostBuf:= GenCostItems(0, SubsBuf);
    SubsBuf.Free;

    Start:= Now;
    Res:= CollectMinDissect(Univers, CostBuf);
    Time:= MilliSecondsBetween(Start, Now);
    Res.Expo; Res.Free;
    Writeln('Time (ms) = ', Time:6:0);
    Writeln('- - - - - - - - - - - - - - - - - - - - - -');

    Start:= Now;
    Res:= CollectMinCover(Univers, CostBuf);
    Time:= MilliSecondsBetween(Start, Now);
    Res.Expo; Res.Free;
    Writeln('Time (ms) = ', Time:6:0);
    Writeln('- - - - - - - - - - - - - - - - - - - - - -');

    Start:= Now;
    Res:= CollectGradCover(Univers, CostBuf);
    Time:= MilliSecondsBetween(Start, Now);
    Res.Expo; Res.Free;
    Writeln('Time (ms) = ', Time:6:0);
    Writeln('- - - - - - - - - - - - - - - - - - - - - -');
    CostBuf.Free;
    Univers.ClrAndDestroy;    Univers.Free;
    Write('OK');
    Readln;
end.
```

Исходный набор содержит 150 «осколков», из которых после отсева дубликатов остаются 115. Для пяти ценовых стратегий были получены результаты, представленные в табл. 11-2 и табл. 11-3 .

Табл. 11-2 — Время поиска разбиений (ЗНР) и покрытий (ЗНП), ms

Параметр стратегии	Стратегия (Cost=…)	ЗНР	ЗНП	
			Точное решение	Градиентный поиск
0	1 + Random(99)	15	31	16
1	1	31	124	15
2	1 + (n div 2) + Random(n)	406	2124	16
3	n	7’015	171’957	15
4	1 + (n-1)*(n-1)	2’327	22’219	15

Табл. 11-3 — Стоимость найденных разбиений и покрытий

Параметр стратегии	Стратегия (Cost=…)	ЗНР	ЗНП	
			Точное решение	Градиентный поиск
0	1 + Random(99)	64	44	57
1	1	4	4	4
2	1 + (n div 2) + Random(n)	12	12	13
3	n	15	15	15
4	1 + (n-1)*(n-1)	15	15	15

Поскольку абсолютное время зависит от мощи компьютера, будем оценивать соотношение времён. Из первой таблицы видно, что разбиение ищется существенно быстрее покрытия, поскольку блоки в ЗНР содержат меньше элементов. Сравнение двух методов ЗНП выводит в чемпионы градиентный поиск. А что у чемпиона с качеством? По Табл. 11-3 видно, что в трёх случаях из пяти градиентный поиск дал идеальный результат, в одном случае — всего на 8% хуже (13 против 12), и ещё однажды он уступил на 30% (57 против 44). Для исчерпывающих выводов нужно больше данных, но градиентный поиск явно заслуживает внимания.

11.6. Комбинированный поиск

Итак, ЗНП решена двумя способами: первый точный, но долгий, второй быстрый, но неточный. Можно ли объединить выгоды того и другого? Вспомним, что точное решение ЗНП даёт ряд промежуточных результатов, постепенно приближающихся к идеалу. Следовательно, есть смысл сначала быстро найти результат градиентным методом, а затем дать ход точному перебору, ограничив его, при необходимости, неким интервалом времени. Если за это время перебор не завершится, то в качестве результата взять последний промежуточный результат.

11.7. Итоги

- Задачи поиска наименьшего покрытия (ЗНП) похожи на задачи поиска наименьших разбиений (ЗНР). Отличие состоит в том, что подмножества, составляющие покрытие, могут взаимно пересекаться.
- При формировании блоков для решения ЗНП каждое подмножество включается во все блоки, помеченные его элементами.
- При переборе подмножеств внутри блока не проверяется взаимное пересечение подмножества-кандидата с накопленным универсумом.
- Ввиду большого количества элементов в блоках, решение ЗНП может быть недопустимо долгим. Это время сильно зависит также от цен подмножеств.
- Альтернативное решение ЗНП — это чрезвычайно быстрый градиентный поиск, не гарантирующий, однако, точного результата.
- В системах, где время поиска решения ограничено, возможно комбинирование градиентного поиска с точным перебором.

11.8. Что почитать

№		Автор(ы)	Название	Главы, страницы
√	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 53
√	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 12
Внутреннее представление графа

Этой главой открывается повествование о *графах*. Теория графов получила развитие с тех пор, когда математики и философы поняли важность не только свойств предметов, но также их взаимных отношений. Не потому ли нас увлекают детективы, где вскрываются тайные связи между предметами, событиями и людьми? Примерим к себе роль детектива, ведь задачи на графах не уступают детективному расследованиям ни в сложности, ни в увлекательности!

Граф часто представляют рисунком, наглядно показывающим *связи* между объектами. Для существования связи, как правило, требуются, по меньшей мере, *два* объекта, — такие связи называют *бинарными*. Существует также рефлексивная связь, которая замыкается на том же объекте, но и она считается бинарной. Обычно объект связан с несколькими другими, однако, каждая такая связь в отдельности тоже бинарная. Предметом рассмотрения будут графы, в которых между любой парой объектов существует *не более одной* связи — такие графы называют *простыми*.

Впрочем, простота их обманчива, — графами моделируется масса разнообразных отношений. Связи между объектами могут быть и симметричными, и направленными. Связи могут обладать рядом качеств: весом, длиной, пропускной способностью. К тому же и связываемые объекты (вершины графа) обычно наделены какими-то свойствами. В пример можно привести сеть дорог, где соединяемые города характеризуются населённостью, а дороги — длиной и количеством полос. Причудливые «узоры», образуемые связями, порождают море интересных и практически важных задач. В этой главе будет сделан первый шаг к решению таких задач: перенесём граф-картинку в память компьютера.

12.1. С точки зрения математиков

Математики определяют граф G как упорядоченное множество, составленное из двух множеств, и записывают это так:

$$G = < X, A >$$

где X — множество вершин графа, A — множество его рёбер (или дуг — для ориентированного графа). Угловые скобки показывают, что множество упорядочено, иначе говоря, множество вершин указано первым. Эта формальная запись привлекательна своим лаконизмом, но не годится для компьютерных манипуляций с графами. Необходимо разработать структуру, пригодную для эффективного представления графа в памяти компьютера.

12.2. Матричные представления

В книгах по дискретной математике нередко приводят матричные (табличные) представления графа, а именно: матрицу *инциденций* и/или матрицу *смежности*.

12.2.1. Матрица инциденций

На рис. 12-1 показан орграф с пронумерованными дугами, а в табл. 12-1 — матрица его инциденций.

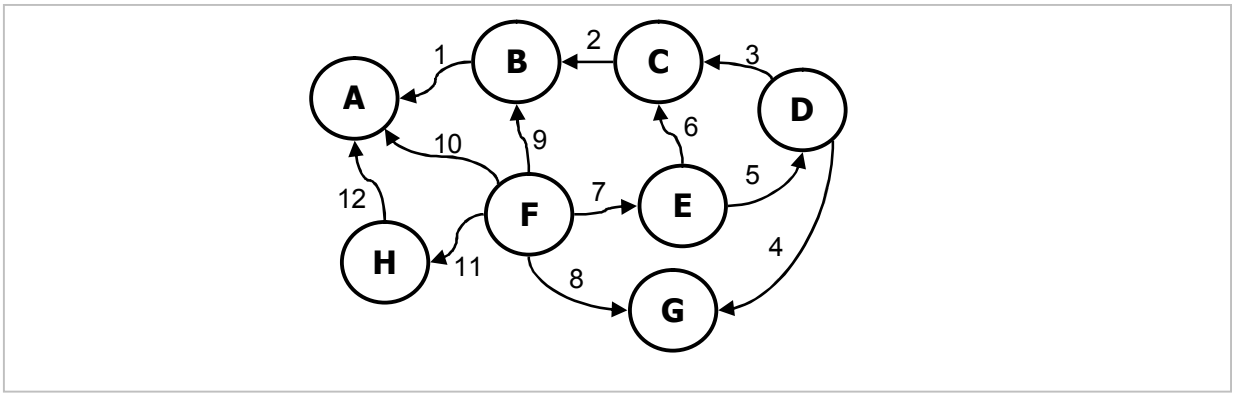


Рис. 12-1 — Пример ориентированного графа

В строках таблицы отмечены вершины, а в столбцах — дуги. Каждая непустая клетка таблицы содержит либо единицу, если дуга выходит из данной вершины, либо минус единицу, если дуга входит в неё. Пустые ячейки означают отсутствие дуг между вершинами. Вместо единиц можно указать числа, характеризующие, к примеру, расстояния между вершинами. Так или иначе, но каждый столбец содержит всего пару чисел, сумма которых равна нулю.

Табл. 12-1 — Матрица инциденций

Вершины	Дуги (номера дуг)											
	1	2	3	4	5	6	7	8	9	10	11	12
A	-1									-1		-1
B	1	-1							-1			
C		1	-1			-1						
D			1	1	-1							
E					1	1	-1					
F							1	1	1	1	1	
G				-1				-1				
H											-1	1

Очевидно, что для разреженных графов большая часть памяти, отведенной таблице, будет пустовать.

12.2.2. Матрица смежности

В левом столбце матрицы смежности (табл. 12-2) указаны вершины, из которых исходят дуги, а в шапке — вершины, в которые они входят. На

пересечениях строк и столбцов отмечены соседние вершины (единицами или числами, означающими вес дуги).

Табл. 12-2 — Матрица смежности

Исходящие вершины	Входящие вершины							
	A	B	C	D	E	F	G	H
A								
B	1							
C		1						
D			1				1	
E			1	1				
F	1	1			1		1	1
G								
H	1							

Из таблицы смежности легко представить граф зрительно. Так, например, по строке *A* видно, что из неё невозможно попасть в другие, то же относится к строке *G*. Зато вершина *F* соседствует сразу с пятью вершинами: *A*, *B*, *E*, *G*, *H*.

По столбцу можно найти вершины, из которых непосредственно достижима данная вершина. Например, вершина *A* достижима из вершин *B*, *F* и *H*.

Число, стоящее на главной диагонали, означает рефлексивную связь вершины с самой собой (в данной таблице таких связей нет). Общее число непустых клеток в матрице смежности равно количеству дуг графа.

Как и в матрице инциденций, вместо единиц можно указать числа, характеризующие длину (вес) дуги.

Матричные представления наглядны и удобны для математиков, но не всегда устраивают программистов. Причина — неэффективное использование памяти. Представим матрицей взаимные знакомства граждан некоторого населённого пункта. Круг знакомств людей обычно составляют не более сотни человек: родственники, соседи, сотрудники. Для жителей деревни почти все клетки в небольшой таблице смежности будут заполнены. Однако для населения крупного города или страны таблица будет огромна, и большинство её клеток окажутся пустыми.

Экономии памяти можно добиться на разреженных матрицах, но сам по себе этот механизм непрост. Здесь будет предложен иной путь: представим графы *множествами*, — такой приём сочетает эффективность с удобством формальных математических описаний.

12.3. Представление множествами

12.3.1. Основные идеи

Вернёмся к матрице смежности, и «выжмем» из неё лишь непустые клетки. Так для некоторой вершины *X* сформируем два множества соседних вершин: тех, что достижимы из неё (в строке), и тех, из которых она достижима (в столбце). На рис. 12-2 показан пример этих множеств в отношении вершины *C*.

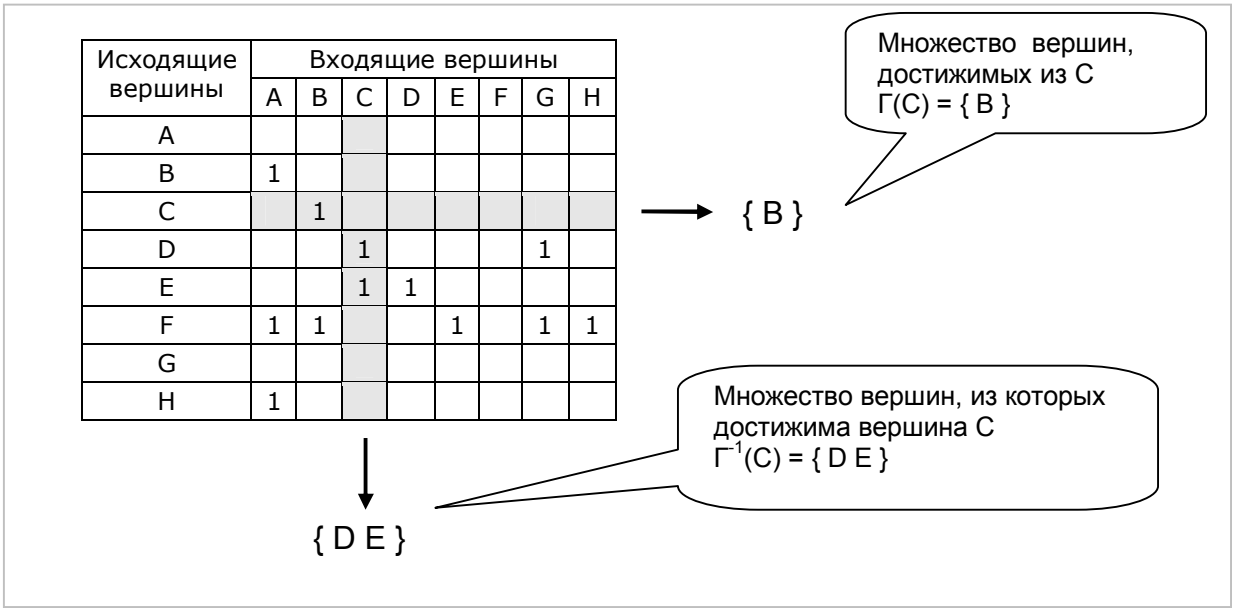


Рис. 12-2 — Множества достижимостей для вершины C

Отметим, что для полноценного представления графа достаточно одного из двух множеств, — другое находится алгоритмически. Но пара множеств значительно ускоряет многие алгоритмы на орграфе. В неориентированном графе достаточно одного множества, поскольку в нём оба множества совпадают.

В табл. 12-3 тот же орграф представлен совокупностью множеств. В левом столбце даны вершины, а в двух соседних — множества смежных им вершин.

Табл. 12-3 — Представление орграфа множествами

Множество вершин	Множество Г	Множество $Г^{-1}$
A	{ }	{ B F H }
B	{ A }	{ C F }
C	{ B }	{ D E }
D	{ C G }	{ E }
E	{ C D }	{ F }
F	{ A B E G H }	{ }
G	{ }	{ D F }
H	{ A }	{ F }

12.3.2. Терминология

Теперь условимся о терминах. Смежные вершины часто обозначают греческой буквой *Г* — «гамма». Будем называть *исходящей гаммой* вершины *X* , или просто *гаммой*, множество *достижимых из неё* соседей. Обозначать исходящую гамму вершины *X* будем так:

$Г(X)$ или $Г^{1}(X)$

Множество смежных вершин, *из которых* достижима данная вершина *X* назовём *входящей гаммой* и обозначим так:

$Г^{-1}(X)$ — гамма степени минус один (входящая).

Следуя далее этому принципу, введём обозначения, показанные в табл. 12-4.

Табл. 12-4 — Обозначение исходящих и входящих гамм

Гаммы		Пояснение
Исходящие	Входящие	
$Г^{0}(X)$	$Г^{0}(X)$	Сама исходная вершина *X*
$Г^{1}(X)$	$Г^{-1}(X)$	Соседи вершины *X*
$Г^{2}(X)$	$Г^{-2}(X)$	Соседи соседей вершины *X*
$Г^{3}(X)$	$Г^{-3}(X)$	Соседи соседей соседей вершины *X*

Для примера на рис. 12-3 показаны исходящие и входящие гаммы вершины *D*.

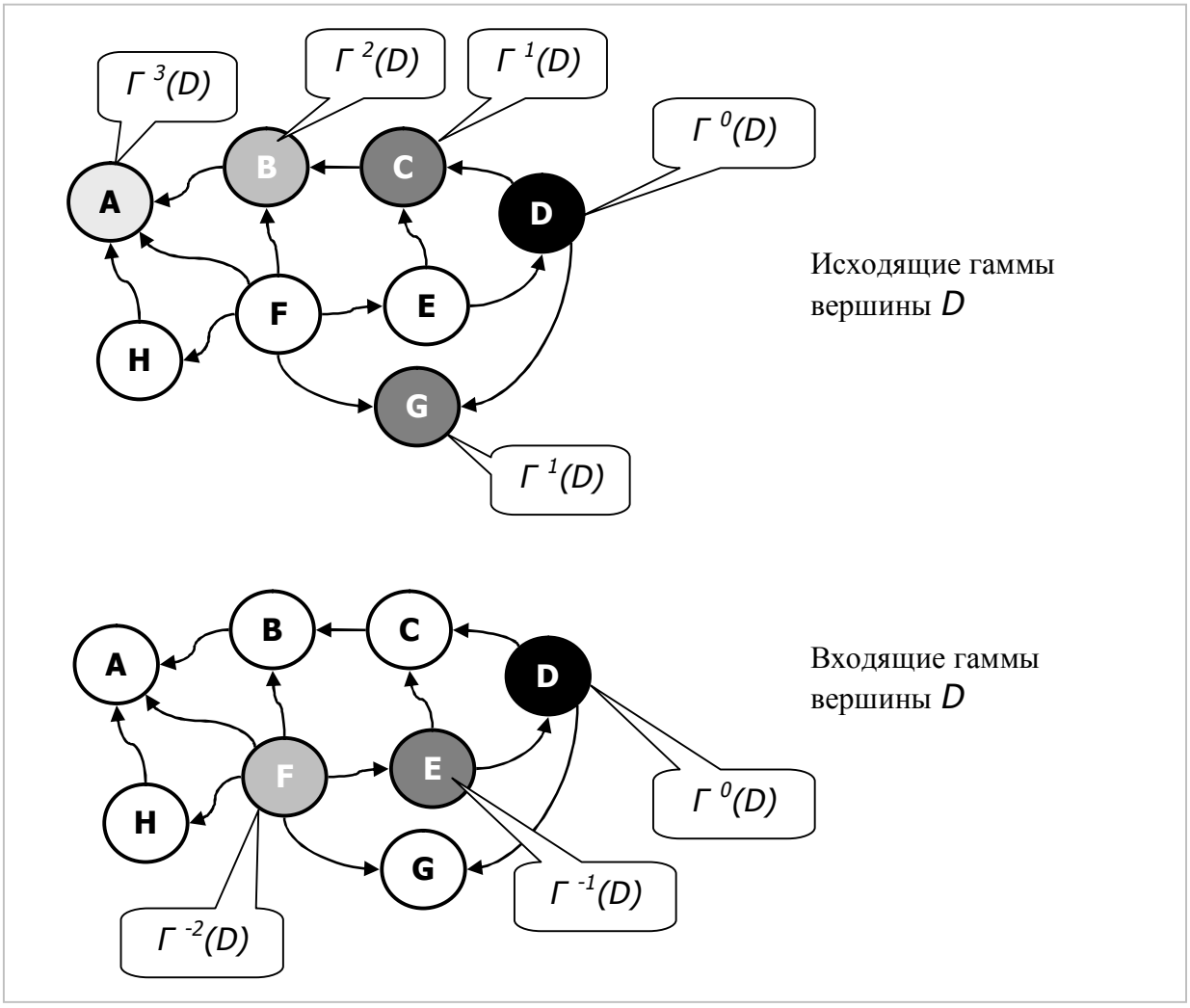


Рис. 12-3 — Исходящие и входящие гаммы вершины D

Иными словами, гамма степени ноль — это сама вершина, а все последующие *исходящие* гаммы степеней **N** доступны из неё за **N** шагов. По отношению к *входящим* гаммам степени **N** можно сказать, что эта вершина доступна из них за **N** шагов.

Объединив исходящие гаммы вершины *X* всех степеней, получим *интегральную* исходящую гамму — множество вершин, которые доступны из *X*. Аналогично объединением входящих гамм всех степеней получим *интегральную* входящую гамму — множество вершин, из которых достижима вершина *X*. Далее для краткости будем называть «гаммой» множества достижимых вершин; точный смысл этого термина будет ясен из контекста, либо дополнительно пояснён.

12.3.3. Граф и его составляющие

Для представления графа в памяти компьютера потребуются элементы трёх классов: собственно граф **TGraph** и две его составляющие: вершины **TNode** и связи **TLink** (линки). Все они будут потомками базового класса **TItem** (элемент).

Рассмотрим основные информационные поля этих объектов, а к прочим полям и методам обратимся по мере решения задач. Информационные поля графа таковы:

```
TGraph = class (TItem)     // Граф в целом
    protected
      mName : String;      // Произвольная строка, отображаемая при выводе
      mDirect : boolean;   // Признак ориентированного графа
      mNodes : TSet;       // Множество вершин графа
      . . .
    public
      mLoadNodes : boolean;  // Признак нагруженных вершин
      mLoadLinks : boolean;  // Признак нагруженных связей
      . . .
  end;
```

Этим набором полей можно представить любые разновидности графов: ориентированные и не ориентированные, с нагруженными и ненагруженными вершинами и связями.

Класс **TNode** описывает вершину графа:

```
  TNode = class (TItem)       // Вершина (узел) графа
    private
      mLnkOut : TSet;         // Множество исходящих связей
      mLnkIn  : TSet;         // Множество входящих связей
    protected
      mOwner  : TGraph;       // Ссылка на содержащий узел граф
      mColor  : integer;      // Цвет (используется в алгоритмах)
      mPred   : TNode;        // Предшествующий узел (Дейкстра)
      mDist   : integer;      // Расстояние (Дейкстра)
      . . .
    public
      mValue : integer;       // Нагрузка узла
      . . .
  end;
```

Поля **mLnkOut** и **mLnkIn** — это множества линков, ссылающихся на смежные вершины, иначе говоря, это исходящая и входящая гаммы первой степени. Для неориентированного графа используем только **mLnkOut**. Поле **mOwner** — это ссылка на граф, содержащего эту вершину, через это поле методы вершины получают доступ к полям графа. Вспомогательные поля **mColor**, **mPred**, **mDist** нужны методам, реализующим стандартные алгоритмы на графах. Поле **mValue** задаёт «вес» вершины в тех задачах, где он учитывается. Другие поля, содержащиеся в вершине, поясним по мере решения соответствующих задач.

Обратимся к связям (линкам) — это элементы множеств **mLnkOut** и **mLnkIn**, содержащихся в вершине **TNode**. Структура элемента связи **TLink** такова:

```
TLink = class (TItem)
    protected
      mLoadLinks : boolean;   // Признак нагруженных связей
      . . .
    public
```

```
      mValue: integer;  // вес дуги (стоимость, расстояние)
      mOwner: TNode;    // вершина-источник дуги (владелец)
      mDest : TNode;    // вершина-приёмник дуги
      . . .
  end;
```

Поле **mValue** задаёт «вес» дуги (длину или стоимость пути). Поля **mOwner** и **mDest** определяют две вершины: источник и приёмник дуги соответственно.

Для каждой связи *ориентированного* графа создаётся *один* линк, но вставляется он в двух местах: в множество **mLnkOut** узла-источника, и в множество **mLnkIn** узла-приёмника.

В *неориентированном* графе каждая связь представлена *двумя* линками: прямым и обратным; они вставляются в множества **mLnkOut** вершины-источника и вершины-приёмника. В этом типе графа поле **mLnkIn** не используется.

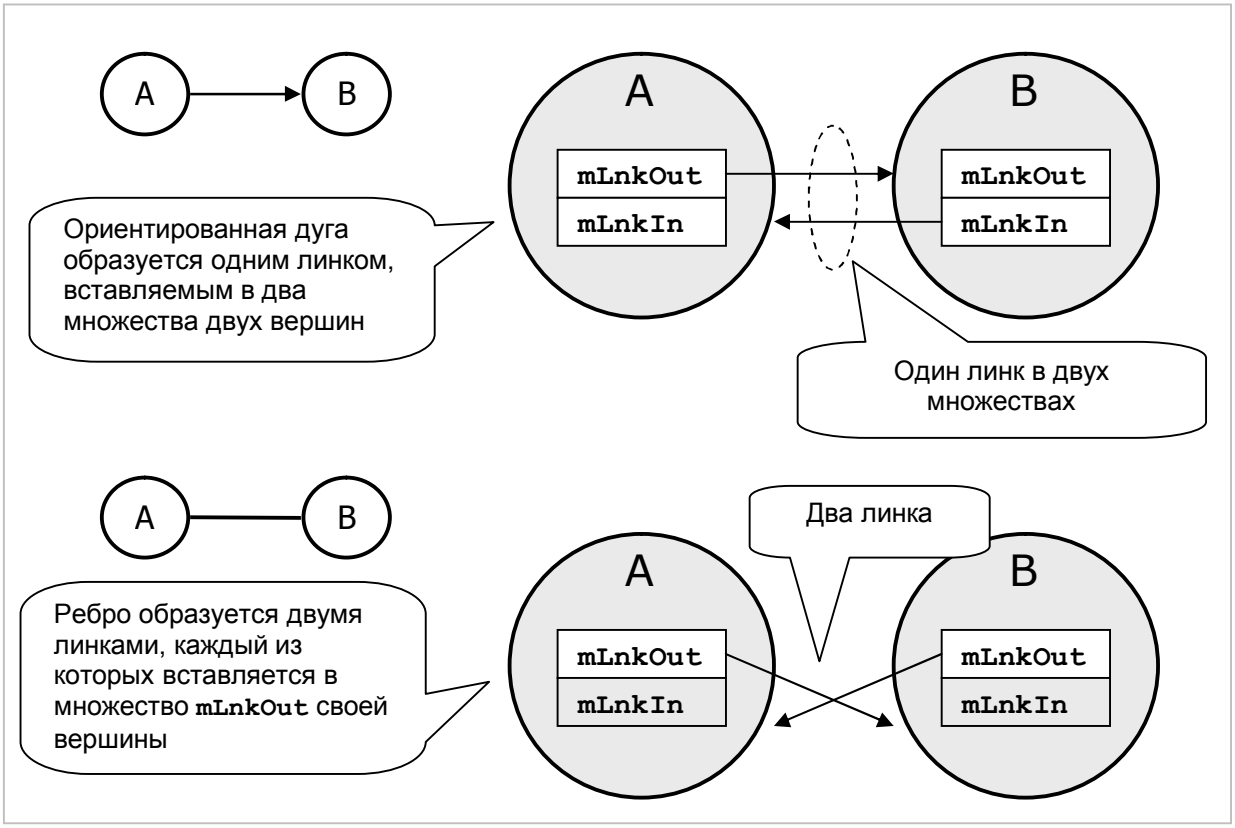


Рис. 12-4 — Организация связей в ориентированном (вверху) и неориентированном (внизу) графах

В следующей главе рассмотрим создание, уничтожение, ввод и вывод графов.

12.4. Итоги

- Табличные представления графа — матрица инциденций и матрица смежности — не вполне эффективны ввиду большого объёма требуемой памяти.

- Для представления графа множествами будут использованы три класса объектов: собственно граф (**TGraph**), вершина графа (**TNode**) и связь (**TLink**). Все они являются наследниками класса **TItem**.
- Вершины графа и связи снабжаются рядом полей, часть из которых выполняет служебные функции, а другие нужны для решения тех или иных задач на графах.
- Направленная дуга орграфа организуется одним объектом (линком), который вставляется в два множества двух вершин: исходящей и входящей.
- Ребро неориентированного графа организуется двумя объектами-линками, каждый из которых вставляется только в множество исходящих связей своей вершины.

12.5. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
√	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 13
Создание и ввод-вывод графов

Главной нашей заботой будет абстрактный класс **TGraph**, — предок графов иных типов, вершины которых могут содержать произвольную информацию: символы, числа, строки или сложные конгломераты данных (включая другие графы!). Класс **TGraph** оснастим методами решения типовых задач на графах, причём реализация этих методов не будет зависеть от типа вершин. Тем самым методы решения типовых задач на графах перейдут ко всем его наследникам.

Однако ввиду того, что объекты абстрактных классов создавать нельзя, наряду с базовыми классами **TGraph** и **TNode** будут сконструированы их наследники, содержащие какие-либо данные — например, символы. Символы удобно отображать на экране, вводить и выводить через текстовые файлы, а для отладки и демонстрации методов графа их вполне достаточно. Классы-наследники **TGraphChars** и **TNodeChar** выделены в отдельный модуль **GrChars**.

13.1. Конструкторы базовых классов

Рассмотрим конструкторы базовых классов, ниже показан конструктор графа (модуль **Graph**):

```
constructor TGraph.Create(const aName: string;
                          aDir, aLNodes, aLLinks: boolean);
begin
  inherited Create;          // вызов унаследованного конструктора
  mName:= aName;             // имя графа
  mDirect:= aDir;            // признак орграфа
  mLoadNodes:= aLNodes;      // признак нагруженных вершин
  mLoadLinks:= aLLinks;      // признак нагруженных связей
  mNodes:= CreateSet;        // пустое множество вершин
end;
```

Он выполняет лишь минимальную работу, заполняя часть полей и создавая пустое множество вершин. Наполнение пустого графа и связывание его вершин посредством линков происходит в других методах:

```
// Вставка узла графа, возвращает FALSE при попытке вставить дубликат

function TGraph.InsertNode(aNode: TNode): boolean;
begin
  Result:= mNodes.Insert(aNode)
end;

// Установка связи дугой или ребром
// aSource - вершина-источник
// aDest   - вершина-приёмник
// aVal    - "вес" дуги или ребра

procedure TGraph.SetLink(aSource, aDest: TNode; aVal: integer);
begin
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
```

```
  aSource.MakeLink(aDest, aVal);          // установка прямой связи
  if not mDirect                          // если не орграф,
    then aDest.MakeLink(aSource, aVal);   // то установка обратной связи
end;
```

Указанные выше методы будут вызываться из конструктора класса-наследника **TGraphChars**.

Обратимся к двум методам объекта-вершины **TNode**. Конструктор вершины будет вызываться, как и положено, из конструктора наследника, а метод **MakeLink** вызывается при установке связи из метода **TGraph.SetLink**.

```
constructor TNode.Create(aVal: integer; aOwner: TGraph);
begin
  inherited Create;             // унаследованный конструктор
  mOwner:= aOwner;              // владелец узла (граф)
  if mOwner.mLoadNodes          // если вершины нагружены
    then mValue:= aVal;         // то запомнить вес
  mLnkOut:= CreateSet;          // пустое множество исходящих связей
  if mOwner.mDirect             // если орграф
    then mLnkIn := CreateSet;   // то создать пустое множество входящих связей
end;

// Установка связи между данным узлом и заданным
// aDest - вершина-приёмник
// aVal  - "вес" связи

procedure TNode.MakeLink(aDest: TNode; aVal: integer);
var L: TLink;
begin
  if not Assigned(aDest) then Exit;
  L:= TLink.Create(Self, aDest, aVal, mOwner.mLoadLinks);
  // попытка вставить исходящую связь
  if not mLnkOut.Insert(L) then begin
    L.Free;  // если уже установлена, то удаляем ненужный линк
    Exit;
  end;
  if mOwner.mDirect               // если орграф
    then aDest.mLnkIn.Insert(L); // то вставляем в приёмнике как входящую
end;
```

Наконец, конструктор объекта-связи выглядит так:

```
constructor TLink.Create(aOwner, aDest: TNode; aVal: integer;
                         aLoadLinks: boolean);
begin
  inherited Create;          // унаследованный конструктор
  mOwner:= aOwner;           // узел-источник связи и её владелец
  mDest:= aDest;             // узел-приёмник связи
  mLoadLinks:= aLoadLinks;   // признак нагруженной связи
  if aLoadLinks              // если нагружена
    then mValue:= aVal       // запоминаем "вес"
    else mValue:= 1;         // а по умолчанию = 1
end;
```

13.2. Наследники

Носителем информации в графе-наследнике будет объект-вершина **TNodeChar**. Отметим, что его класс-предок **TNode** снабжен лишь служебными полями, поэтому в вершине-наследнике добавлено поле **mName**, хранящее один символ (см. модуль **GrChars**):

```
TNodeChar = class (TNode)
    private
      function GetLinkByName(aName : char): TLink;
    public
      mName  : char;  // хранимый символ
      constructor Create(aName: char; aVal: integer; aOwner: TGraph);
      function  Compare(arg: TItem): TCompare; override;
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;
```

Описание объекта-наследника **TGraphChars,** содержащего вершины-символы, выглядит так:

```
TGraphChars = class (TGraph)
    private
      function  GetLinkByName(aSource, aDest : char): TLink;
      procedure MakeLink(aSource, aDest : char; aVal : integer);
    public
      constructor Load(const aName: String);
      constructor GenRandom(aDir: boolean;
                            aLoadNodes, aLoadLinks,
                            aNodes, aLinks : integer);
      procedure Save(const aName: String);
      function  GetNode(aName : char): TNodeChar;
  end;
```

Наряду с тремя вспомогательными методами, здесь введены два конструктора и метод **Save**, сохраняющий граф в текстовом файле. Конструктор **Load** и метод **Save** взаимно согласованы так, что первый «умеет» вводить данные из файла, созданного вторым. Описание и образец такого файла для орграфа с взвешенными вершинами представлен ниже.

13.3. Формат файла для ввода и вывода графа

Первая строка файла с графом содержит произвольный текст — это комментарий к графу. Вторая — число 0 или 1, где 0 означает неориентированный граф, а 1 – орграф. Третья и четвёртая строки задают признаки ненагруженных (0) или нагруженных (1) вершин и рёбер. В пятой строке задано количество вершин, а имена вершин перечислены в шестой строке, причём для графа с взвешенными вершинами после знака «=» указывается вес вершины.

Последующие строки содержат связи каждой из вершин. Если дуга или ребро взвешены, то после каждой связи через знак «=» указывается вес связи, например:

```
A -> B=3 E=7 F=2
```

Вот пример файла для загрузки орграфа с взвешенными вершинами:

```
Орграф с взвешенными вершинами - комментарий
1 - граф(0), орграф(1)
1 - нагруженность вершин
0 - нагруженность рёбер (дуг)
13 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8 I=9 J=10 K=11 L=12 M=13
A -> B E F
B -> A
C -> B D E
D -> I
E -> A G
F -> E H J
G -> D
H -> G J
I -> G
J -> H
K -> J L
L -> H K M
M -> K
```

Такой файл удобен для ручной правки. Ознакомиться с методами загрузки и выгрузки файла можно по листингу модуля **GrChars**.

13.4. Генерация случайного файла

Другим средством создания графа является конструктор случайного графа, объявленный так:

```
constructor TGraphChars.GenRandom(
                            aDir: boolean;  // признак орграфа
                            aLoadNodes,     // нагруженность вершин
                            aLoadLinks,     // нагруженность связей
                            aNodes,         // предельное число вершин
                            aLinks:         // предельное число связей (%)
                            integer);
```

Здесь задаются основные параметры графа, а также предельное количество вершин и связей. Количество связей задаётся в процентах от теоретического максимума (**N*N-N**). При положительных значениях **aNodes** и **aLinks** их фактическое количество сформируется случайно, а при отрицательных — точно. Метод предназначен для тестовых программ.

13.5. Визуализация файла (метод Expo)

При отладке задач на графах полезно наблюдать граф в удобной для восприятия форме (или выводить в этой форме в текстовый файл). Напомню, что выполняющие эту работу методы **Print** (виртуальный) и **Expo** (статический) существуют во всех предках базового класса **TItem**.

Визуализация графа реализована в базовом классе (**TGraph.Print**), что избавляет нас от повторения этой работы в наследниках. Метод очень прост, поскольку применяет отработанную визуализацию множеств. Особенности наследника учтены через виртуальный метод **GetName** класса **TNode** где он объявлен абстрактным. А в классе-наследнике **TNodeChar** он реализован так:

```
function TNodeChar.GetName: string;
begin
  Result:= mName;  // возвращаем символ - название вершины
end;
```

13.6. Испытание

Для демонстрации методов конструирования и отображения графа предназначена следующая ниже программа.

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';

Const  CFile = 'Output.txt';
var Gr : TGraphChars;
    S : string;
    F : Text;
begin
  repeat
    Gr:= TGraphChars.GenRandom(   // создание случайного графа
              Boolean(Random(2)), // признак орграфа
              Random(2),          // признак нагруженных вершин
              Random(2),          // признак нагруженных дуг (рёбер)
              20,                 // вершин не более 20
              40{%}               // связей не более 40%
              );
    Gr.Expo;  // отображение на экране
    Write('Save? [Y] :'); Readln(S);
    if (S='y') or (S='Y') then begin
      // Сохранить в файле вместе с визуальным отображением
      Gr.Save(CFile);
      Assign(F, CFile);
      Append(F);
      Writeln(F,'============================');
      Gr.Print(F);
      Close(F);
      // Прочитать из файла и вывести на экран:
      Gr.Free;
      Gr:= TGraphChars.Load(CFile);
      Gr.Expo;
      Readln;
    end;
    Gr.Free;
  until S<>'';
end.
```

Результаты работы этой программы представлены ниже.

Файл, описывающий не взвешенный орграф:

```
Random: T : 0 : 0 : 14 : 20
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
14 - количество вершин
A B C D E F G H I J K L M N
A -> I J
B -> C I
C -> M
D ->
E -> G J
F ->
G ->
H -> I
I ->
J -> E H I N
K -> E G J N
L -> D
M -> J K L
N ->
```

Отображение этого орграфа на экране методом **Expo** таково:

```
Random: T : 0 : 0 : 14 : 20
{
A  ->   { I J } : 2
B  ->   { C I } : 2
C  ->   { M } : 1
D  ->   { } : 0
E  ->   { G J } : 2
F  ->   { } : 0
G  ->   { } : 0
H  ->   { I } : 1
I  ->   { } : 0
J  ->   { E H I N } : 4
K  ->   { E G J N } : 4
L  ->   { D } : 1
M  ->   { J K L } : 3
N  ->   { } : 0
} : 14
```

Файл, описывающий граф с взвешенными вершинами и рёбрами:

```
Random: F : 1 : 1 : 13 : 10
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
13 - количество вершин
A=1 B=1 C=1 D=1 E=2 F=2 G=2 H=1 I=2 J=1 K=1 L=1 M=1
A -> B=2
B -> A=2 M=1
C -> I=2 M=1
D ->
E ->
F -> G=1
G -> F=1
H ->
I -> C=2
J ->
K ->
L ->
M -> B=1 C=1
```

Тот же файл, отображённый на экране методом **Expo**:

```
Random: F : 1 : 1 : 13 : 10
{
A =  1 ->   { B= 2 } : 1
B =  1 ->   { A= 2 M= 1 } : 2
C =  1 ->   { I= 2 M= 1 } : 2
D =  1 ->   { } : 0
E =  2 ->   { } : 0
F =  2 ->   { G= 1 } : 1
G =  2 ->   { F= 1 } : 1
H =  1 ->   { } : 0
I =  2 ->   { C= 2 } : 1
J =  1 ->   { } : 0
K =  1 ->   { } : 0
L =  1 ->   { } : 0
M =  1 ->   { B= 1 C= 1 } : 2
} : 13
```

Обратите внимание, что вызовом **Gr.Save('')** можно отобразить исходный файл с описанием графа на экране.

13.7. Итоги

- Основой для реализации графов любых типов является класс **`TGraph`**, в нём содержатся все статические методы, решающие задачи на графах. Однако этот класс не привязан к конкретным данным. Для наполнения графа данными создаются наследники узла и графа.
- Наследники **`TGraphChars`** и **`TNodeChar`** оперируют с символами. В наследнике графа предусмотрены два конструктора: для загрузки из файла, и для случайной генерации графа. Реализован также методы сохранения графа в файле и отображения его на экране.

Глава 14
Достижимость

Отныне предметом наших размышлений будет Банановый архипелаг — группа островов (рис. 14-1), где по прихоти своей мы будем строить мосты, разрушать их, и возводить новые. Всё это будет подчинено решению задач на графах. Эти решения оформим как статические методы объекта класса **TGraph**.

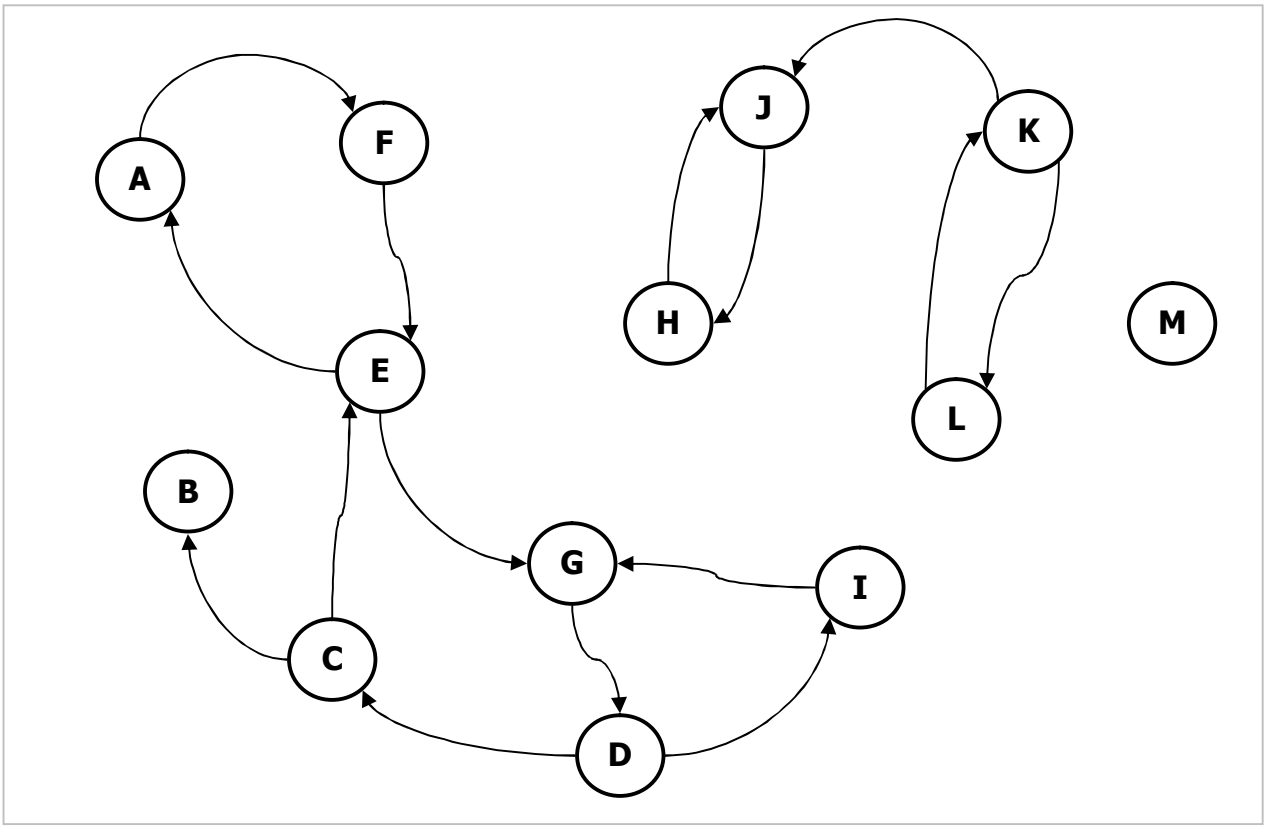


Рис. 14-1 — Мостовая сеть Банановых островов

14.1. Достижима ли вершина?

Простейшая задача о достижимости формулируется так: возможно ли, следуя по дугам или рёбрам, попасть из вершины X в вершину Y? Да или нет? Вопрос применим и к графам, и к орграфам. Из того, что некоторые острова или группы островов не связаны мостами, следует, что часть островов взаимно не достижима. Но даже когда все вершины орграфа соединены, это не гарантирует повсеместной взаимной достижимости. Например, с острова B недостижимы другие острова (см. рис. 14-1), хотя сам он достижим из части других островов.

Для выявления достижимости построим булеву функцию — метод **TGraph.TestLink**, принимающую два параметра: исходную и конечную вершины. Метод будет возвращать **TRUE**, если возможен путь из первой вершины во вторую:

Листинг 14-1 — Проверка достижимости вершины

```
// Возвращает TRUE при наличии пути из вершины 1 в вершину 2
// (граф и орграф)

function TGraph.TestLink(arg1, arg2: TNode): boolean;
var SN1 : TSet;     // предыдущее множество вершин
    SN2 : TSet;     // следующее  множество вершин (накопитель)
   node : TNode;    // очередная вершина из SN1
    old : integer; // мощность предыдущей гаммы
      i : integer;
begin
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    Result:= SN1.Exist(arg2);    // конечная вершина в гамме?
    if Result then Break;        // да, выход из цикла
    old:= SN2.GetCount;          // запоминаем мощность
    // расширяем гамму SN2 := Gamma(SN1)
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если гамма не расширилась
    SN1.CopyItems(SN2);    // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
end;
```

Решение состоит в постепенном расширении множества смежных вершин — накоплении исходящей интегральной гаммы, начиная от исходной вершины:

$$\Gamma = \Gamma^{0} + \Gamma^{1} + \Gamma^{2} + \ldots$$

Цикл расширения гаммы **repeat-until** прекращается, либо когда целевая вершина попала в гамму Γ, либо когда расширение гаммы стало невозможным, и тогда функция возвращает **FALSE**.

Текущая гамма — множество вершин **SN2** — расширяется циклическим вызовом метода **TNode.OutGammaAdd**, который добавляет к множеству **SN2** исходящую гамму для каждого узла из **SN1**. Если после очередной попытки расширения мощность **SN2** не увеличилась, значит, дальнейшее движение по мостам невозможно. Метод **TNode.OutGammaAdd** будет применяться ещё не раз, и потому покажем его здесь:

```
procedure TNode.OutGammaAdd(aRes : TSet);
var L: TLink;
begin
  L:= OutLinkFirst;             // первая исходящая связь
  while Assigned(L) do begin    // перебор связей
    aRes.Insert(L.mDest);       // вставить целевую вершину в результат
    L:= OutLinkNext;            // следующая исходящая связь
  end;
end;
```

14.2. Подсчёт промежуточных мостов

Слегка изменив функцию **TestLink**, можно получить количество дуг (мостов) на пути из одной вершины в другую. Если вершины не достижимы, функция **TGraph.CalcSteps** возвращает минус единицу:

Листинг 14-2 — Подсчёт количества дуг на пути из вершины в вершину

```
function TGraph.CalcSteps(arg1, arg2: TNode): integer;
var SN1 : TSet;      // предыдущее множество вершин
    SN2 : TSet;      // следующее  множество вершин (накопитель)
   node : TNode;     // очередная вершина из SN1
     ok : boolean;   // признак достижения целевой вершины
    old : integer;   // предыдущая мощность накопленной гаммы
      i : integer;
begin
  Result:= 0;
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    ok:= SN1.Exist(arg2);  // признак достижения целевой вершины
    if ok then Break;
    old:= SN2.GetCount;    // запоминаем мощность до расширения
    // Накопление гаммы:  SN2 := Gamma (SN1)
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы SN1
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если гамма не расширилась
    Inc(Result);
    SN1.CopyItems(SN2);   // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
  if not ok then Result:= -1;   // признак недостижимости
end;
```

Следующая программа служит для испытания рассмотренных выше методов:

Листинг 14-3 — Проверка наличия пути между двумя вершинами

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';

var Gr : TGraphChars;
    NodeA, NodeB : TNode;
    S : string;

procedure Save;
const CName = 'Out.txt';
var F: TextFile;
begin
  Gr.Save(CName);
  AssignFile(F, CName);
```

```
  if FileExists(CName) then Append(F) else Rewrite(F);
  Writeln(F,'- - - - - - - - - - - - - - - - - -');
  Writeln(F,'A -> B : ', Gr.TestLink(NodeA, NodeB));
  Writeln(F,'B -> A : ', Gr.TestLink(NodeB, NodeA));
  Writeln(F,'A => B : ', Gr.CalcSteps(NodeA, NodeB));
  Writeln(F,'B => A : ', Gr.CalcSteps(NodeB, NodeA));
  Writeln(F,'-----------------------------------');
  Close(F)
end;
//////////////////////////////////////////////////////////////
begin
  repeat
    Gr:= TGraphChars.GenRandom(false, 0, 0, 20, 40);
    Gr.Expo;
    NodeA:= Gr.GetNode('A'); NodeB:= Gr.GetNode('B');
    Writeln('A -> B : ', Gr.TestLink(NodeA, NodeB));
    Writeln('B -> A : ', Gr.TestLink(NodeB, NodeA));
    Writeln('A => B : ', Gr.CalcSteps(NodeA, NodeB));
    Writeln('B => A : ', Gr.CalcSteps(NodeB, NodeA));
    Write('s - Save, q - Quit : '); Readln(S);
    if S='s' then Save;
    Gr.Free;
  until S='q';
end.
```

Вот один из результатов, выданных программой:

```
Исходный неориентированный не взвешенный граф
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
13 - количество вершин
A B C D E F G H I J K L M
A -> C
B -> F G K
C -> A G
D -> L
E -> H
F -> B M
G -> B C H M
H -> E G
I ->
J -> M
K -> B L
L -> D K
M -> F G J
- - - - - - - - - - - - - - - - - -
A -> B : TRUE
B -> A : TRUE
A => B : 3
B => A : 3
```

14.3. Итоги

- Одна вершина графа достижима из другой, если существует путь из второй в первую.
- В неориентированном графе, где связи симметричны, достижимость вершин взаимна, в орграфе это не так.

- Достижимость целевой вершины из исходной вершины, а также количество промежуточных дуг (рёбер) ищется посредством расширения множества смежных вершин (построением исходящей интегральной гаммы).

14.4. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
√	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
√	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 15
Кратчайшие пути

В отношение достижимости вершин можно задать следующие вопросы: которые из вершин и дуг лежат на пути между исходной и конечной вершинами? И, если путей несколько, то который из них короче?

15.1. Кратчайший путь

Сначала рассмотрим решение задачи для единичных дуг или рёбер (иначе говоря, без учёта длины дуг). Пусть требуется найти кратчайший путь из вершины *A* к вершине *G* (рис. 15-1). Сначала построим вспомогательное дерево обратных связей. Для этого, начиная с исходной вершины *A*, станем обрабатывать соседей, помечая в них два поля:

- **`mColor`** — цвет вершины, это числа: 0 - белый, 1 - серый, 2 - чёрный;
- **`mPred`** — указатель обратной связи на предшествующую вершину.

Здесь предстоит своеобразное расширение гаммы, но с привлечением подсобного средства — очереди вершин. Вначале все вершины сбросим в исходное состояние: цвет белый, а обратный указатель пустой. Затем исходную вершину покрасим серым и поставим в очередь обработки (серость вершины — это признак пребывания её в очереди), — в этом состоит начальная подготовка (рис. 15-1).

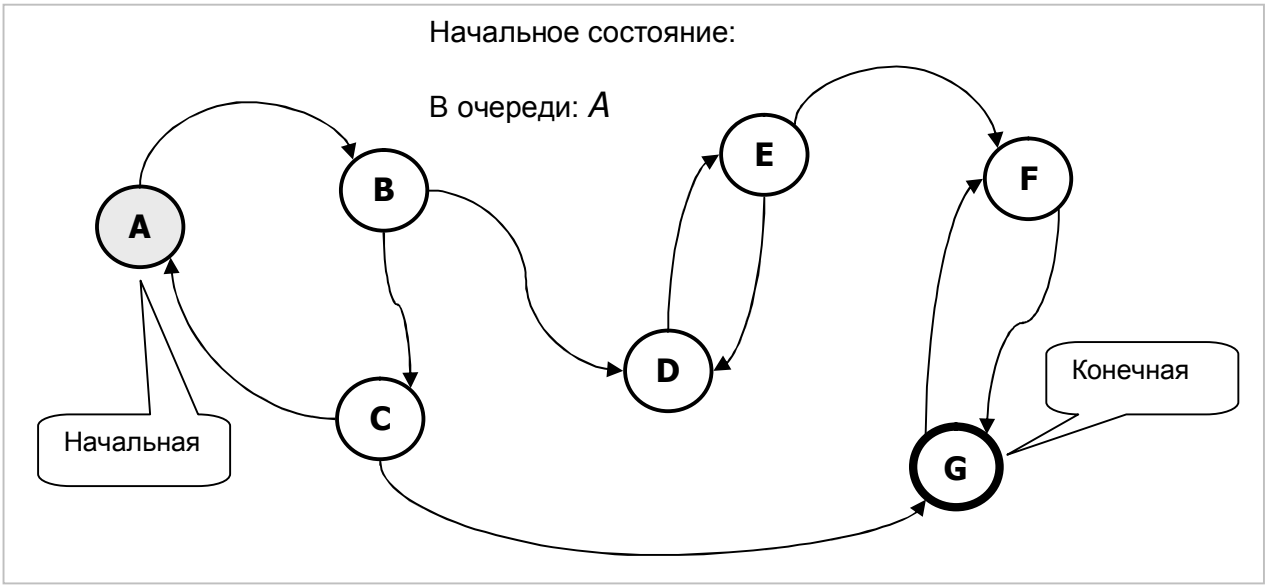


Рис. 15-1 — Начальное состояние процесса построения дерева

Далее запускаем цикл построения дерева обратных связей. Выбрав из очереди первую вершину, поступаем с её белыми соседями так: указатель обратной связи устанавливаем на текущую вершину, красим соседнюю вершину серым и ставим в очередь. Так продолжаем, пока соседней не окажется целевая вершина *G*, либо не будет исчерпана очередь вершин.

Два промежуточных состояния процесса показаны на следующих рисунках, где обратные связи изображены пунктиром.

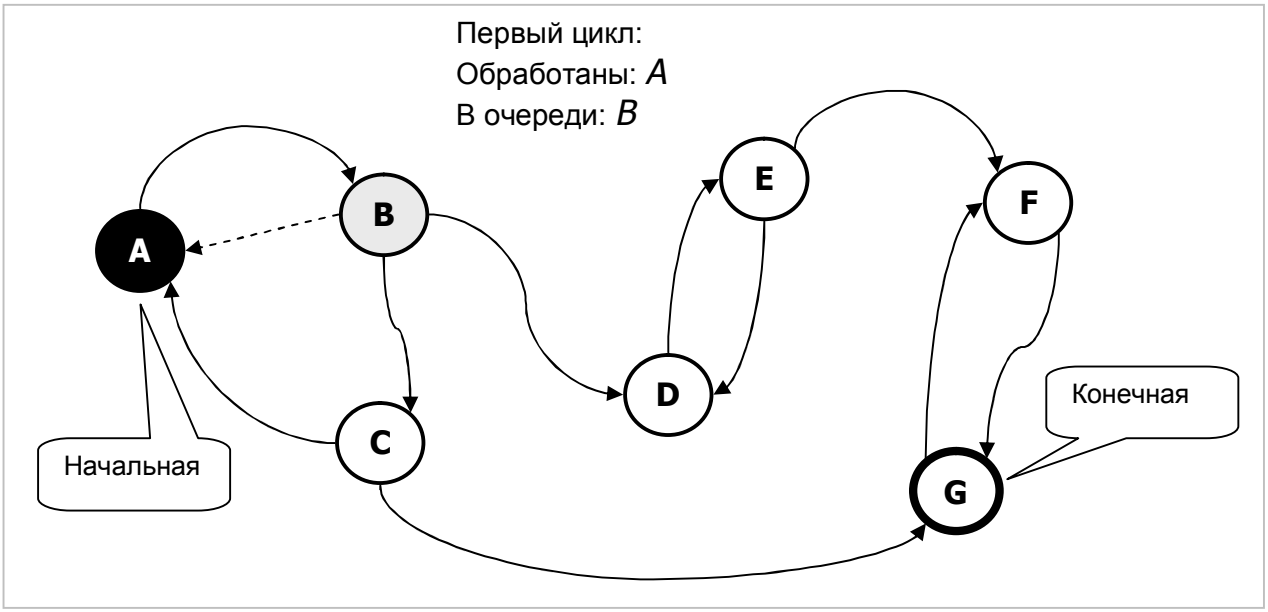


Рис. 15-2 — Состояние процесса после обработки A

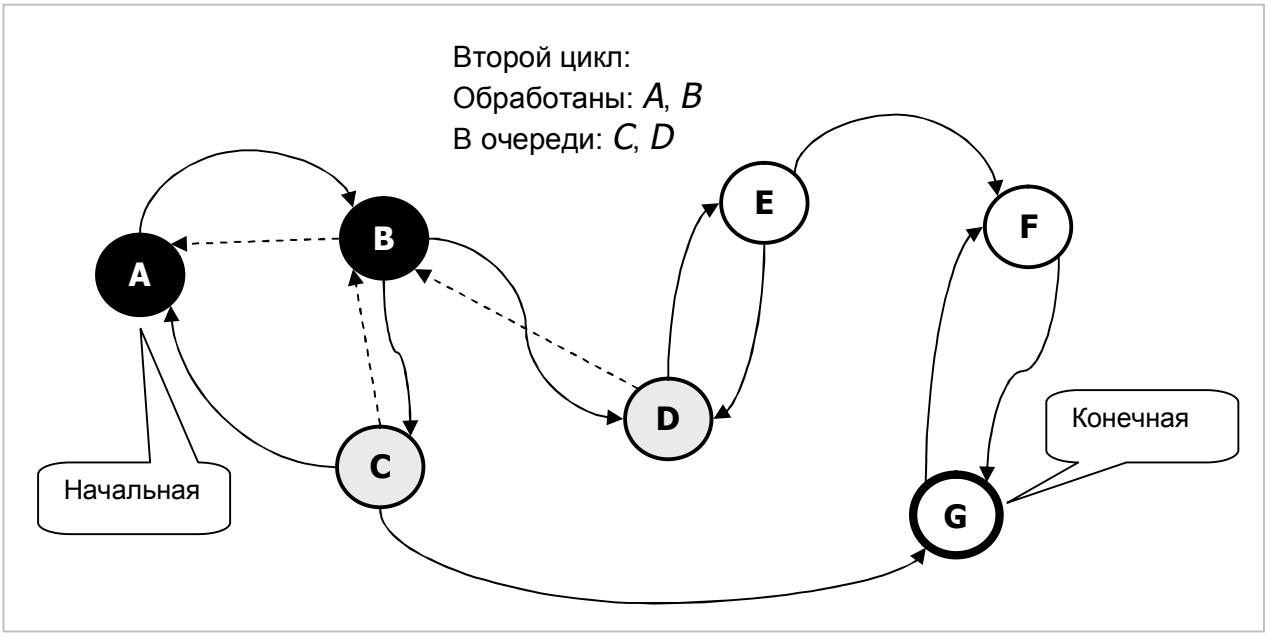


Рис. 15-3 — Состояние процесса после обработки A и B

В конце концов, когда при обработке соседей будет обнаружена конечная вершина, обратные связи будут такими, как показано на рис. 15-4. С этого момента выполняется финальная фаза: следуя от конечной вершины по обратным связям, строим путь от конечной вершины к начальной. Затем для получения пути от начальной вершины к конечной этот путь реверсируем. В неориентированном графе реверс не потребуется, если расширение гаммы начать с конечной вершины.

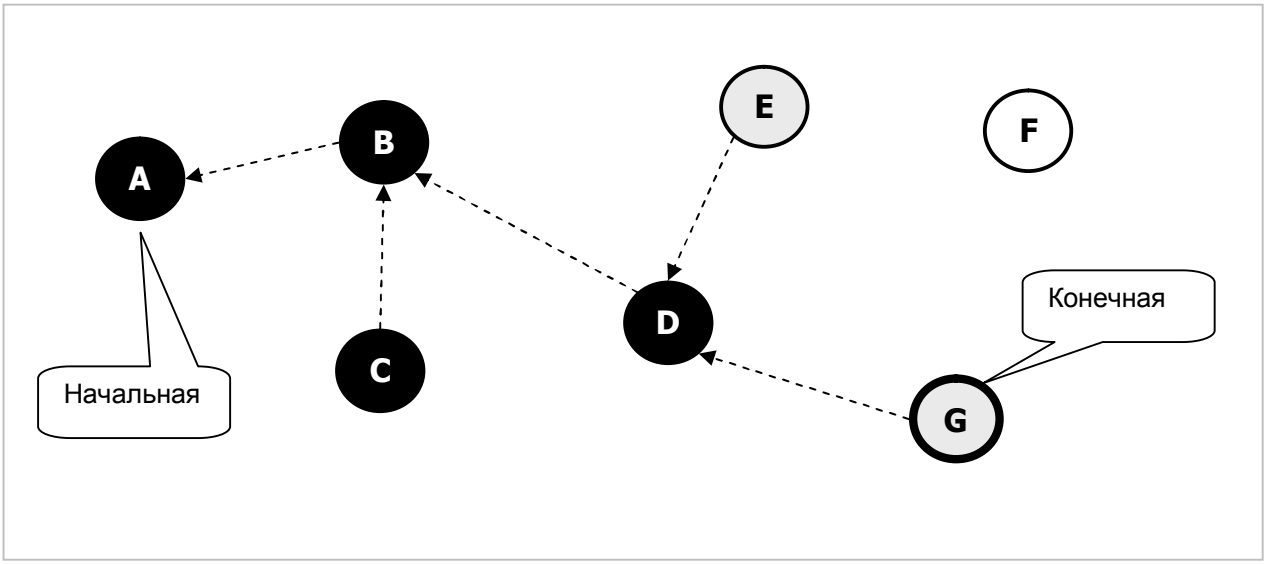


Рис. 15-4 — Дерево обратных связей при достижении конечной вершины G

Этот алгоритм реализован в методе **TGraph.GenPath**.

Листинг 15-1 — Построение кратчайшего пути в не взвешенном графе

```
function TGraph.GenPath(aSource, aDest: TNode): TBuffer;

var  Que : TBuffer;  // очередь вершин
    Node : TNode;      // текущая вершина
    Link : TLink;      // исходящая связь
      OK : boolean;    // признак достижения целевой вершины

begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)

  ResetNodes;               // сброс узлов в исходное состояние
  aSource.mColor:= CGray;   // исходная = серая
  Que.Push(aSource);        // исходную в очередь

  repeat // повторяем, пока очередь не пуста
    OK:= false;                    // признак достижения конечной вершины
    Node:= TNode(Que.Get);         // извлечь очередную вершину
    Link:= Node.OutLinkFirst;      // первая исходящая связь
    while Assigned(Link) do begin // перебор исходящих связей
      with Link do if mDest.mColor = CWhite then begin
        // если смежная не тронута (белая)
        mDest.mPred:= Node;        // указатель на предыдущую
        OK:= mDest = aDest;        // целевая?
        if OK then Break;          // выход из цикла while
        mDest.mColor:= CGray;      // иначе красим серым
        Que.Push(mDest);           // и ставим в очередь
      end;
      Link:= Node.OutLinkNext; // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;       // обработанную вершину красим чёрным
    // выход, если достигнута целевая либо исчерпана очередь
  until OK or (Que.GetCount = 0);
```

```
  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая
    Que.Clear;                        // очистить буфер
    Que.Push(aDest);                  // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Push(Node);                 // вершину --> в результат
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
    Que.Free;        // если целевая не достигнута, ликвидируем буфер
  end;
end;
```

Следующая программа испытывает метод **GenPath**.

Листинг 15-2 — Программа для испытания метода **GenPath**

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';

var Gr : TGraphChars;
    P : TBuffer;
    NodeA, NodeB : TNode;
    S : string;

//////////////////////////////////////////////////////////////
///  Сохранение избранных графов

procedure Save;
const CName = 'Out.txt';
var F: TextFile;
begin
  Gr.Save(CName);
  AssignFile(F, CName);
  Append(F);
  Writeln(F,'- - - - - - - - - - - - - - - - - -');

  P:= Gr.GenPath(NodeA, NodeB);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;

  Writeln(F,'- - - - - - - - - - - - - - - - - -');

  P:= Gr.GenPath(NodeB, NodeA);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;

  Writeln(F,'- - - - - - - - - - - - - - - - - -');
  Close(F)
end;
```

```
begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 5, 5, 20, 50);
    Gr.Expo;

    NodeA:= Gr.GetNode('A');
    NodeB:= Gr.GetNode('B');
    Writeln('- - - - - - - - - - - - - - - - - -');

    P:= Gr.GenPath(NodeA, NodeB);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - -');

    P:= Gr.GenPath(NodeB, NodeA);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - -');

    Write('s - Save, q - Quit : '); Readln(S);
    if S='s' then Save;
    Gr.Free;
  until S='q';
end.
```

15.2. Алгоритм Дейкстры

Теперь рассмотрим поиск кратчайшего пути для графов с взвешенными дугами и (или) вершинами. Пусть некий турист отправляется из города *A* в город *C*. География даёт ему три маршрута, но по неясным причинам стоимость проезда по дорогам оказалась такова, как показано на рис. 15-5.

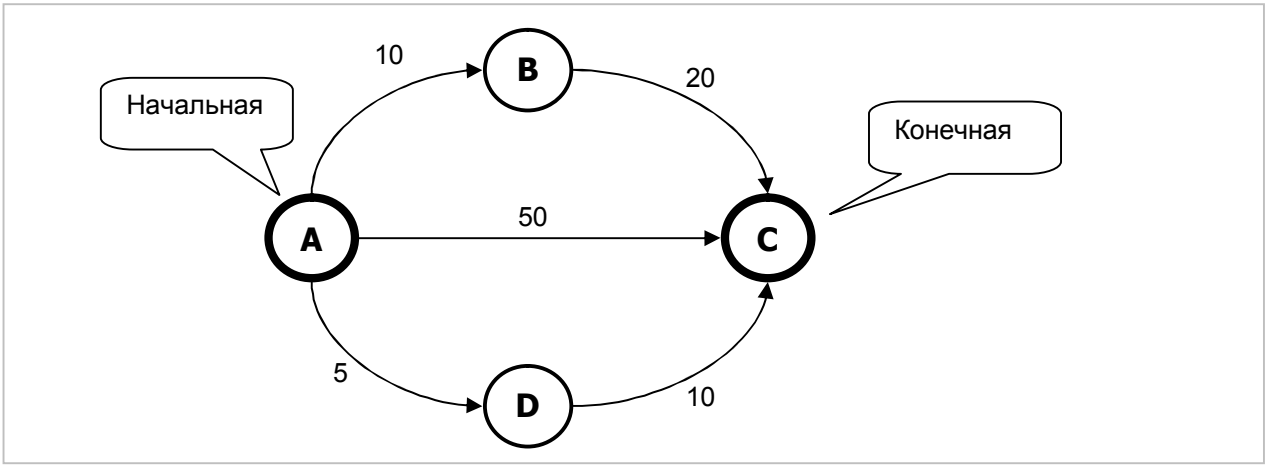


Рис. 15-5 — Возможные пути из A в C

Здесь самым дорогим является прямой путь из *A* в *C* (50 пиастров), а самым дешёвым — путь через город *D* (15 пиастров). Построим алгоритм, отыскивающий этот наиболее дешёвый путь (или один из путей, если существуют несколько равноценных). Вдобавок к стоимости дуг, учтём и стоимость промежуточных вершин, — тем самым можно моделировать стоимость ночлега в городах (в нашем примере вершины не взвешены и стоимость ночлега нулевая).

В основе решения лежат те же идеи, что в первой задаче, но ссылку на предшествующую вершину будем поправлять с учётом накопленной стоимости

пути как суммы стоимостей промежуточных дуг и вершин. Пусть после обработки и почернения исходной вершины *A* в очереди оказались серые вершины *B*, *C* и *D* (рис. 15-6).

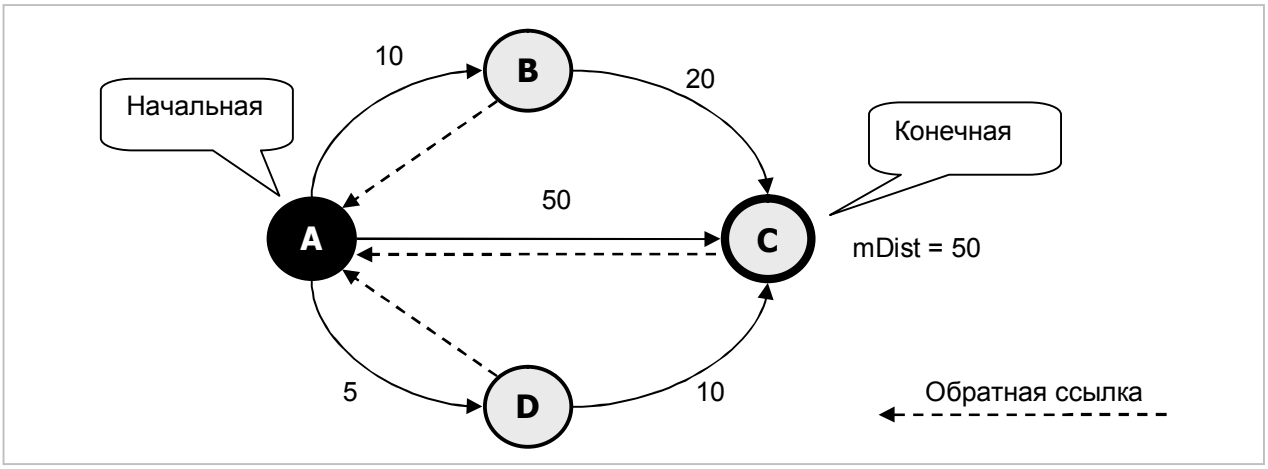


Рис. 15-6 — Состояние после почернения вершины A

Здесь пунктиром показаны обратные ссылки на предшествующие вершины. Значение поля **mDist** для вершины *C* составит 50 — это стоимость дуги *A-C*.

Продолжим выбирать вершины из очереди и обрабатывать их (в алфавитном порядке). При обработке вершины *B* исследуем расстояния от неё ко всем соседним вершинам. Тут выясняется, что сумма стоимостей дуг 10+20 меньше, чем отмеченная в вершине *C* стоимость 50, потому ссылку на предшествующую вершину перестраиваем с вершины *A* на вершину *B*, а в поле **mDist** заносим 30 (рис. 15-7).

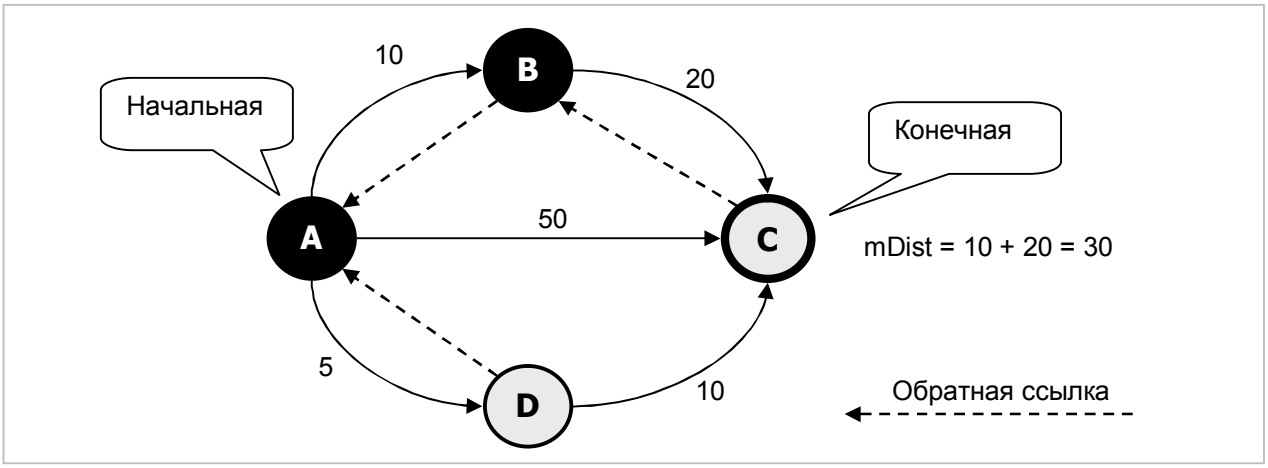


Рис. 15-7 — Состояние после обработки вершины B

При обработке вершины *C* ничего не изменится, поскольку у неё нет смежных исходящих вершин (рис. 15-8).

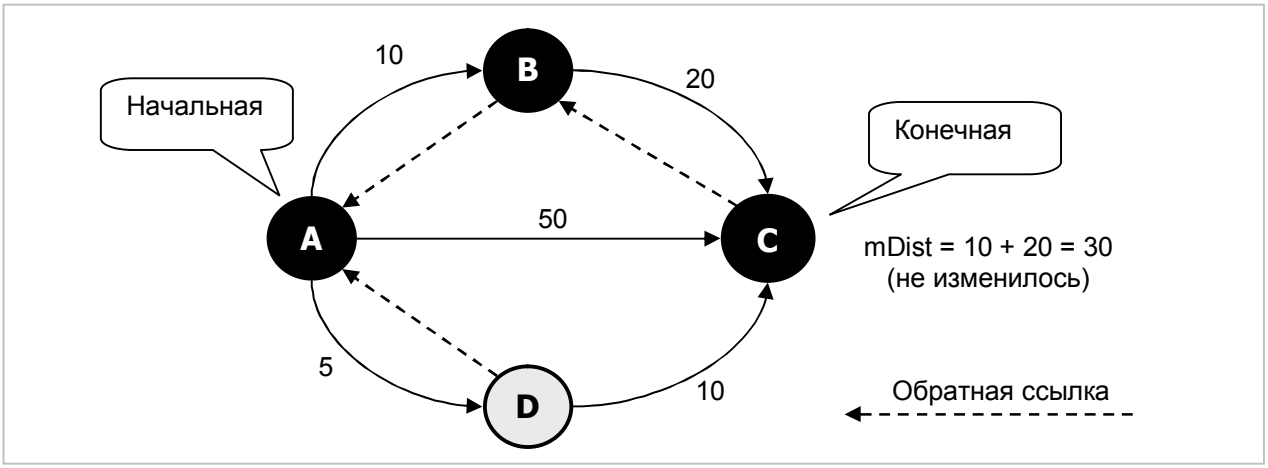


Рис. 15-8 — Состояние после обработки вершины C не изменилось

При обработке вершины *D* вновь оказывается, что смежная с нею вершина *C* может быть достигнута за меньшую цену, поэтому обратную ссылку в *C* и поле **mDist** вновь корректируем (рис. 15-9).

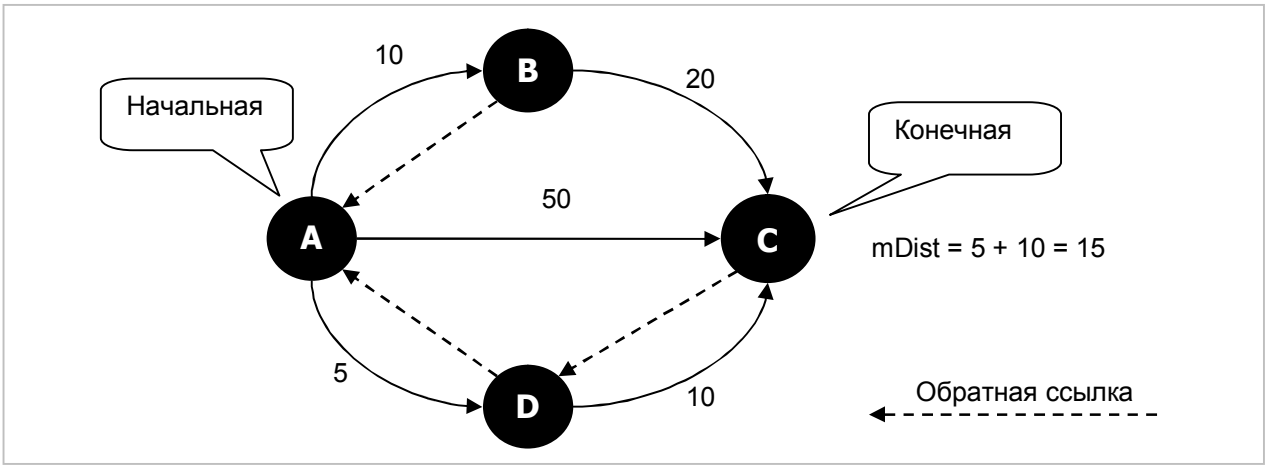


Рис. 15-9 — Состояние после обработки вершины D

В итоге кратчайшим путём из *A* в *C* оказался путь через вершину *D*. Детали алгоритма показаны в следующем листинге. Отметим, что здесь нельзя прерывать цикл при первом достижении конечной вершины. Кратчайший путь может быть окольным, и потому надо обработать все доступные вершины графа (вплоть до опустошения очереди).

Листинг 15-3 — Алгоритм Дейкстры

```
function TGraph.GenDijkstra(aSource, aDest: TNode): TBuffer;
var  Que : TBuffer;  // очередь вершин
    Node : TNode;      // текущая вершина
    Link : TLink;      // исходящая связь
begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)
  ResetNodes;              // сброс узлов в исходное состояние
  aSource.mColor:= CGray;  // исходная = серая
  aSource.mDist:= 0;       // расстояние к исходной = 0
  Que.Push(aSource);       // исходную в очередь
```

```
  repeat                    // повторяем, пока очередь не пуста
    Node:= TNode(Que.Get); // извлечь очередную вершину
    Link:= Node.OutLinkFirst; // первая исходящая связь
    while Assigned(Link) do with Link do begin  // обраб. текущего линка
      if mDest.mDist > Node.mDist + Node.mValue + mValue then begin
        // поскольку расстояние в смежной больше нового,
        // обновляем расстояние и обратную ссылку
        mDest.mDist:= Node.mDist + Node.mValue + mValue;
        mDest.mPred:= Node;
        // Если смежная не стоит в очереди, то красим её белым,
        // чтобы поставить в очередь (возможно, что повторно)
        if mDest.mColor <> CGray then mDest.mColor:= CWhite;
      end;
      if mDest.mColor = CWhite then begin
        // если смежная не стояла в очереди
        mDest.mColor:= CGray;    // то красим серым
        Que.Push(mDest);         // и ставим в очередь
      end;
      Link:= Node.OutLinkNext;  // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;    // признак, что вершина обработана
  until Que.GetCount = 0;    // выход, если исчерпана очередь

  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая вершина
    Que.Clear;                        // очистить буфер
    Que.Push(aDest);                  // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Push(Node);                 // вершину --> в результат
      if Node = aSource then Break;   // выход, если достигли исходную
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
    Que.Free;    // если целевая не достигнута, ликвидируем буфер
  end;
end;
```

Для проверки метода служит следующая программа.

Листинг 15-4 — Программа для проверки алгоритма Дейкстры

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';


var Gr : TGraphChars;
    P : TBuffer;
    NodeA, NodeB : TNode;
    S : string;
```

```
//  Сохранение путей в файле

procedure Save;
const CName = 'Out.txt';
var F: TextFile;
begin
  Gr.Save(CName);
  AssignFile(F, CName);
  Append(F);
  Writeln(F,'- - - - - - - - - - - - - - - - - -');
  P:= Gr. GenDijkstra (NodeA, NodeB);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;
  Writeln(F,'- - - - - - - - - - - - - - - - - -');
  P:= Gr. GenDijkstra (NodeB, NodeA);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;
  Writeln(F,'- - - - - - - - - - - - - - - - - -');
  Close(F)
end;
//////////////////////////////////////////////////////////////
begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 5, 5, 20, 50);
    Gr.Expo;

    NodeA:= Gr.GetNode('A'); // взять ссылку на A
    NodeB:= Gr.GetNode('B'); // взять ссылку на B
    Writeln('- - - - - - - - - - - - - - - - - -');
    P:= Gr. GenDijkstra (NodeA, NodeB);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - -');
    P:= Gr. GenDijkstra (NodeB, NodeA);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - -');
    Write('s - Save, q - Quit : '); Readln(S);
    if S='s' then Save;
    Gr.Free;
  until S='q';
end.
```

15.3. Навигационная карта

15.3.1. Основная идея

Любой, кто плутал в незнакомой местности, будет рад указателю на разыскиваемый объект: следуя по цепочке таких промежуточных указателей, легко отыскать дорогу.

Сходную проблему решают в компьютерных сетях, где отдельные узлы могут быть источниками информации, её получателями, а также промежуточными узлами: маршрутизаторами или роутерами (англ. *Router*). Путь сетевых пакетов от источника к получателю обычно пролегает через несколько промежуточных узлов. Получив пакет, узел решает, которому из соседей передать его с тем, чтобы пакет достиг конечного адресата по кратчайшему пути. Роутер не «беспокоит Дейкстру» по поводу каждого пакета, а пользуется заранее подготовленной «картой» — набором готовых указателей.

Организуем в графе нечто подобное дорожным указателям с тем, чтобы составлять кратчайшие маршруты между любыми парами вершин по возможности быстро и просто.

Первое что приходит в голову: найти маршруты между всеми парами вершин и хранить их в готовом виде. Избыточность такого решения в смысле занимаемой памяти очевидна. А время? В графе, имеющем **N** вершин, количество путей между всеми парами вершин составит **N•(N-1)**. Поиск каждого из них методом Дейкстры требует обработки **N** вершин, стало быть, предварительные затраты времени будут пропорциональны кубу от размера графа (N^3).

С другой стороны, путнику, следующему, к примеру, из города *E* в город *C* (рис. 15-10) в начальной точке не обязательно знать весь маршрут к цели, — достаточно выбрать лишь один из немногих исходящих из *E* путей. Подходящий указатель направит путника сначала в город *B*, где его ждёт следующий указатель на город *C*.

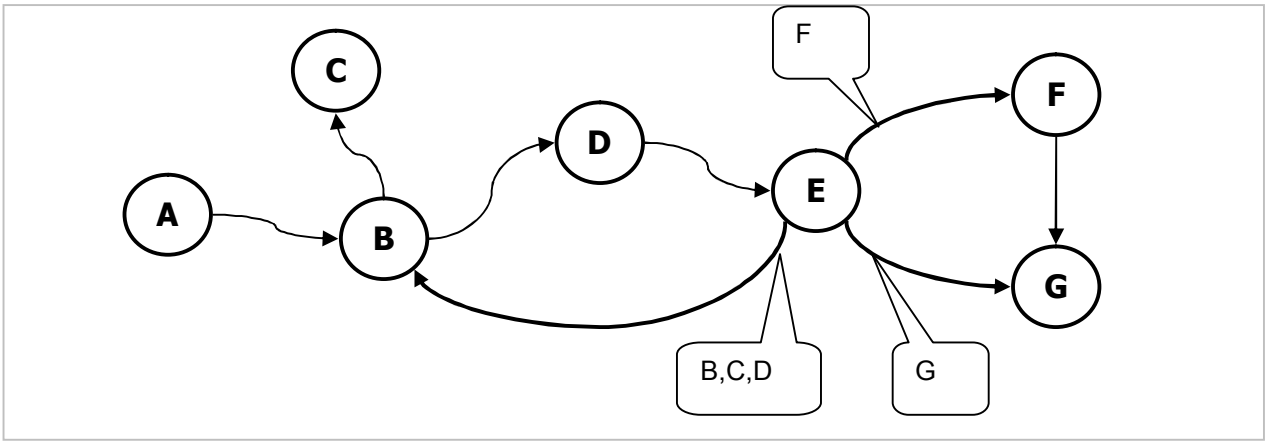


Рис. 15-10 — Дальние указатели кратчайших путей из вершины E

Принятое здесь название указателя — *дальний* — обусловлено тем, что он указывает путь к *дальней* вершине через ближайшую (смежную) вершину, лежащую на искомом пути. Структура этого указателя такова:

```
TFarLink = class(TItem)      // Дальний указатель
    mNodeFar : TNode;        // целевая вершина
    mNodeNear: TNode;        // ближайшая вершина на пути к целевой
    mDist    : integer;      // расстояние от текущей к целевой
    mStep    : integer;      // номер этапа, используется при построении карты
    constructor Create(aNear, aFar: TNode; aDist: integer);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile);  override;
  end;
```

Пусть удалось организовать в каждой вершине графа (в поле **TNode.mFarLinks**) множество таких указателей. Тогда путь между исходной вершиной **aSource** и конечной **aDest** можно построить за линейное время, следуя по дальним указателям, что реализовано в методе **TGraph.GenQuickPathStr**:

Листинг 15-5 — Формирование кратчайшего маршрута по карте маршрутизации

```
function TGraph.GetQuickPathStr(aSource, aDest: TNode): String;
var Next : TNode;
      FL : TFarLink;
       S : String;
begin
  Result:= aSource.GetName;
  // Если распределённая карта указателей ещё не построена, то строим её
  InitMap_Floyd;
  // Проверяем достижимость целевой вершины:
  Next:= aSource.GetNear(aDest);
  if not Assigned(Next) then Exit;   // выход, если не достижима
  // Следуем по маршруту:
  repeat
    Result:= Result + ' -> ' + Next.GetName;
    Next:= Next.GetNear(aDest);
  until Next = aDest;
  FL:= aSource.GetFarLink(aDest); // дальний указатель на конечную
  Str(FL.mDist, S);               // растояние до конечной
  Result:= Result + ' -> ' + Next.GetName + ' : ' + S;
end;
```

Здесь переход к следующей вершине выполняется методом **TNode.GetNear**, который ищет ближайшую вершину перебором дальних связей (см. листинг).

Обратите внимание на оператор:

```
  // Если карта ещё не построена, то строим её
  InitMap_Floyd;
```

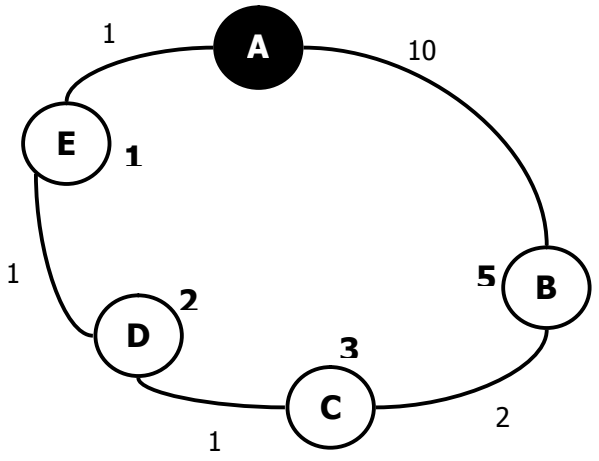
Здесь вызывается вспомогательный метод **InitMap_Floyd**, — этому методу уделим дальнейшее внимание.

15.3.2. Алгоритм Флойда-Уоршелла

Итак, для быстрого поиска кратчайших путей между двумя вершинами необходимо предварительно расставить в вершинах графа подходящие дальние указатели. Для построения такой распределённой карты можно было бы призвать алгоритм Дейкстры, но метод Флойда-Уоршелла быстрее, его мы и рассмотрим.

Суть алгоритма Флойда-Уоршелла состоит в следующем. Пусть на каком то этапе решения известен путь и соответствующее ему расстояние между парой вершин, причём на этом пути лежит **M** промежуточных вершин (**M=0...N-2**). Если на следующем этапе алгоритма будет обнаружен менее длинный путь между этими же вершинами, то прежний путь и расстояние заменяются новыми.

Рассмотрим алгоритм на примере графа, изображённого на рис. 15-11. Рядом с рёбрами указаны их длины, а рядом с вершинами — кратчайшие расстояния к вершине *A*. Мы видим, что хотя вершины *A* и *B* соединены ребром непосредственно, кратчайший путь между ними пролегает через ряд других вершин и длина его (стоимость) составляет 5.



Граф, в котором кратчайшие пути из вершины *A* ко всем остальным вершинам лежат через вершину *E*.

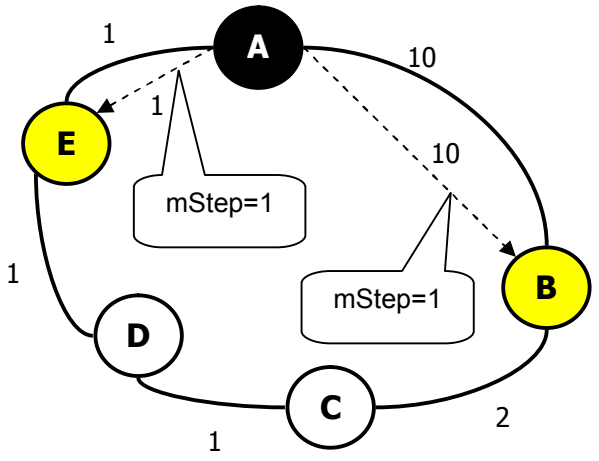
Рис. 15-11 — Пример циклического графа

Начнём строить распределённую карту данного графа по Флойду-Уоршеллу, формируя дальние указатели из вершины *A*. Параллельно будут формироваться аналогичные указатели из других вершин графа, хотя для упрощения рисунка они не показаны (формирование прочих указателей можно проследить в следующих далее таблицах).

В нулевом, подготовительном цикле алгоритма, разместим в каждой вершине множество дальних указателей на все вершины графа (см. описание класса **`TFarLink`**). Здесь один из них будет ссылаться на саму вершину *A*, два других — на ближайших соседей *B* и *E*, — на рис. 15-12 они показаны пунктиром. Дальние указатели на вершины *C* и *D* на этом этапе существуют, но ещё не определены, и на рисунке не показаны. Их поля **`mNodeNear`** будут пусты, а поля **`mDist`** равны условной бесконечности (**`MaxInt`**).

Точно так же готовятся дальние указатели из других вершин, все они показаны в табл. 15-1. Сформированные на этом этапе указатели ссылаются пока лишь на соседние вершины, их поля **`mStep`** установлены в единицу, а это значит, что они будут обработаны в первом цикле.

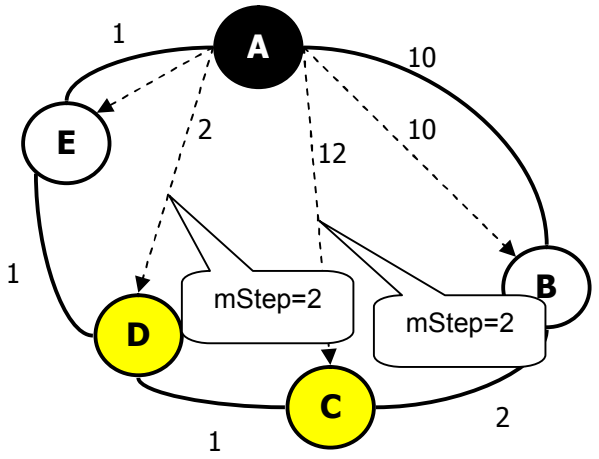
После начальной подготовки выполняем циклы обработки вершин. В первом цикле в каждой вершине обрабатываются дальние указатели, помеченные числом 1 (поле **`mStep=1`**). Через эти указатели становятся доступными вершины, отстоящие через одно ребро (соседи соседей). Если текущая длина дальнего указателя к новой вершине окажется больше, чем сумма расстояний от исходной к промежуточной и от промежуточной к конечной, то дальний указатель корректируется: изменяется расстояние и ссылка на ближайшую вершину.



Исходное состояние дальних указателей из вершины *A*.
Определены лишь два указателя на соседние вершины.
В следующем (первом) цикле будут обработаны *B* и *E*.

Рис. 15-12 — Исходное состояние перед первым циклом алгоритма

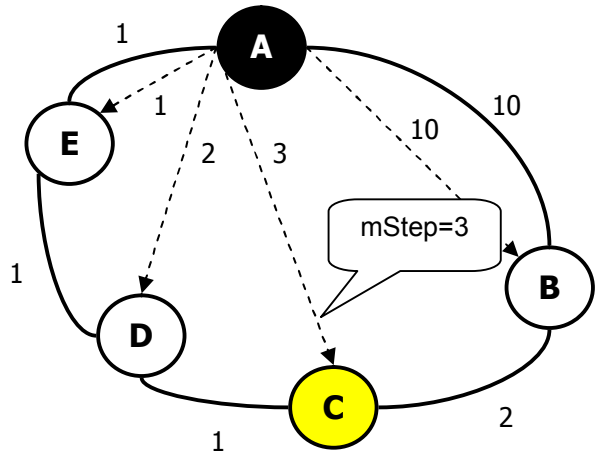
На рис. 15-13 показаны дальние указатели вершины *A* после первого цикла. Здесь из вершины *B* по цепочке *A*→*B*→*C* стала доступной вершина *C* на расстоянии 10+2=12. Из вершины *E* по цепочке *A*→*E*→*D* стала доступна вершина *D* на расстоянии 1+1=2. Обновлённые указатели к вершинам *C* и *D* теперь отмечены в поле **mStep** значением 2, стало быть, будут обработаны на втором цикле.



Состояние дальних указателей из вершины *A* после 1-го цикла.
Определены ещё два указателя на вершины *C* и *D*.
Указатели к вершинам *B* и *C* не оптимальны (расстояния 10 и 12 соответственно).
Во втором цикле будут обработаны *C* и *D*.

Рис. 15-13 — Состояние после 1-го цикла

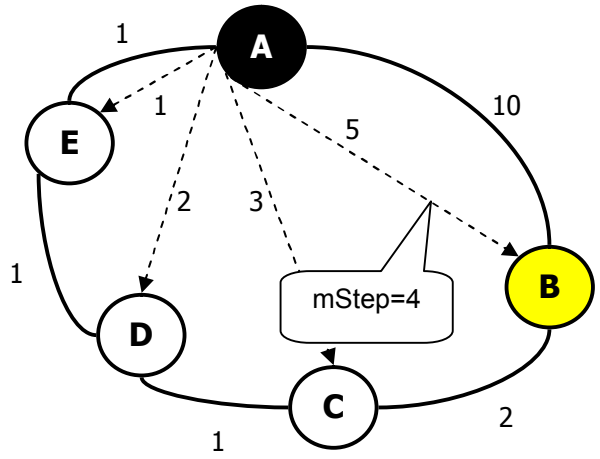
На втором цикле из вершины *C* по цепочке *A*→*B*→*C*→*D* становится доступной вершина *D*, но поскольку этот второй путь длиннее существующего (12+1), то дальний указатель на *D* не изменяется. Одновременно вершина *C* делается доступной из *D* по менее длинному пути *A*→*E*→*D*→*C* (2+1), потому указатель *A*→*C* перестраивается на путь через вершину *E*, что показано на рис. 15-14. Этот указатель помечается числом 3, чтобы быть обработанным в третьем цикле.



Состояние дальних указателей из вершины *A* после 2-го цикла. Указатель на вершину *C* изменился так, что путь к ней пролёг через вершину *E*. а расстояние уменьшилось до 3. В третьем цикле будет обработана вершина *C*.

Рис. 15-14 — Состояние после 2-го цикла

В третьем цикле обрабатывается только вершина *C*. Здесь вершина *B* стала доступна через неё по цепочке *A*→*E*→*D*→*C*→*B* с расстоянием 3+2, что меньше 10. Потому указатель на неё перестраиваем: теперь путь к *B* лежит через *E*, а расстояние составит 5 (рис. 15-15).



Состояние дальних указателей из вершины *A*. после 3-го цикла. Указатель на вершину *B* изменился так, что путь к ней стал пролегать через вершину *E*. а длина уменьшилась до 5.

Рис. 15-15 — Состояние после 3-го цикла

Сколько циклов потребуется на формирование всех дальних указателей? Из того, что длиннейший путь между вершинами содержит не более чем **N-2** промежуточных вершин, а в каждом цикле добавляется не более одной такой вершины, количество циклов составит не более **N-2**. Реально их может быть существенно меньше: когда в очередном цикле не обновится ни один дальний указатель, работа прекращается. Отметим также, что вместо пометок **mStep** здесь можно применить очередь, тогда алгоритм завершится по исчерпании этой очереди.

В следующих далее таблицах показаны этапы формирования дальних указателей для всех вершин графа. В левом столбце даны исходные вершины, в

верхней строке — конечные. На пересечениях даны ближайшие вершины на пути от исходной вершины к конечной и расстояния между ними. Пустые клетки соответствуют временно не определённым дальним указателям (**mNodeNear=nil**). Жёлтым цветом отмечены указатели, которые будут обрабатываться на данном этапе (то есть, стоят в очереди обработки).

Табл. 15-1 — Дальние указатели перед 1-м этапом (исходное состояние)

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	B = 10			E = 1
B --> A,C	A = 10	B = 0	C = 2		
C --> B,D		B = 2	C = 0	D = 1	
D --> C,E			C = 1	D = 0	E = 1
E --> A,D	A = 1			D = 1	E = 0

Табл. 15-2 — Дальние указатели перед 2-м этапом

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	B = 10	B = 12	E = 2	E = 1
B --> A,C	A = 10	B = 0	C = 2	C = 3	A = 11
C --> B,D	B = 12	B = 2	C = 0	D = 1	D = 2
D --> C,E	E = 2	C = 3	C = 1	D = 0	E = 1
E --> A,D	A = 1	A = 11	D = 2	D = 1	E = 0

Табл. 15-3 — Дальние указатели перед 3-м этапом

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	B = 10	E = 3	E = 2	E = 1
B --> A,C	A = 10	B = 0	C = 2	C = 3	C = 4
C --> B,D	D = 3	B = 2	C = 0	D = 1	D = 2
D --> C,E	E = 2	C = 3	C = 1	D = 0	E = 1
E --> A,D	A = 1	D = 4	D = 2	D = 1	E = 0

Табл. 15-4 — Дальние указатели перед 4-м этапом (итог)

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	E = 5	E = 3	E = 2	E = 1
B --> A,C	C = 5	B = 0	C = 2	C = 3	C = 4
C --> B,D	D = 3	B = 2	C = 0	D = 1	D = 2
D --> C,E	E = 2	C = 3	C = 1	D = 0	E = 1
E --> A,D	A = 1	D = 4	D = 2	D = 1	E = 0

Ниже представлен листинг метода **TGraph.InitMap_Floyd**.

Листинг 15-6 — Построение распределённой карты дальних указателей методом Флойда-Уоршелла

```
procedure TGraph.InitMap_Floyd;
  //- - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей
  procedure InitFarLinks(aNode: TNode);
  var     Node : TNode;      // текущая вершина
          Link : TLink;      // ближний указатель
            FL : TFarLink;   // дальний указатель
  begin
    PosPush;
    Node:= NodeFirst;
    while Assigned(Node) do begin   // цикл создания дальних связей
      if Node = aNode then begin
        FL:= TFarLink.Create(Node, Node, 0);   // связь на себя
      end else with aNode do begin
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        Link:= OutLinkFirst;            // первая исходящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mDest = Node then begin
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue + Ord(mLoadNodes)*Node.mValue;
            FL.mStep:= 1;      // указатель будет обработан на первом шаге
            Break;
          end;
          Link:= OutLinkNext;     // следующая исходящая связь
        end; // while
      end; // else
      aNode.mFarLinks.Insert(FL); // вставить в множество
      Node:= NodeNext;
    end;
    PosPop;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
         FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin  { Handle }
    Result:= false;
    FL1:= aNode.mFarLinks.GetFirst as TFarLink; // указатель из вершины aNode
```

```
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)            // если определён
             and (mDist<>0)                 // и не сам на себя
             and (mNodeFar<>aNode)          // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear) // то проверяем расстояние
              then Result:= true;           // признак того, что изменён
          FL2:= GetNext as TFarLink;        // следующая дальняя связь
        end; // while
        PositionPop;
      end;
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
var  Node : TNode;     // текущая вершина
     Step : integer;   // этап обработки (номер цикла)
     Flag : boolean;   // признак продолжения обработки

begin { InitMap_Floyd }

if mInitMap then Exit;  // Защита от повторной инициализации
  mInitMap:= true;         // Признак, что карта создана
  // Предварительная очистка карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin        // перебор вершин
    with Node do begin
      if Assigned(mFarLinks)           // если карта существует
        then mFarLinks.ClrAndDestroy   // очищаем
        else mFarLinks:= CreateSet;    // иначе создаём пустую
    end;
    InitFarLinks(Node);   // инициализация дальних указателей
    Node:= NodeNext;
  end;

  // Обработка вершин
  for Step:= 1 to mNodes.GetCount-2  do begin
    Flag:= false;
    Node:= NodeFirst;
    while Assigned(Node) do begin     // перебор вершин
      if Handle(Node, Step)           // если обновлялись дальние указатели
        then Flag:= true;             // то отметить
      Node:= NodeNext;
    end;
    if not Flag then Break;  // если указатели не обновлялись, то выйти
  end;
end;
```

15.3.3. Испытания

Следующая программа предназначена для испытания метода Флойда-Уоршелла, а также для сравнения его с методом Дейкстры по скорости.

Листинг 15-7 — Испытание метода Флойда-Уоршелла

```
{$APPTYPE CONSOLE}

uses
  SysUtils, DateUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  Assembly in '..\Common\Assembly.pas',
  SetUtils in '..\Common\SetUtils.pas';

var   Gr : TGraphChars;
       S : string;
       P : TBuffer;
    Node1, Node2 : TNode;
    Start, Time : TDateTime;
    i, N1, N2 : integer;

begin
  Gr:= TGraphChars.Load('Test_1.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - -');
  repeat
    Write('S:= '); Readln(S);    // два первых символа - имёна вершин
    if Length(S)<2 then break;   // признак завершения

    Node1:= Gr.GetNode(Upcase(S[1]));  // исходная вершина
    Node2:= Gr.GetNode(Upcase(S[2]));  // конечная вершина

    // Поиск кратчайшего пути методом Дейкстры
    P:= Gr.GenDijkstra(Node1, Node2);
    if Assigned(P) then P.Expo;
    P.Free;
    Writeln('- - - - - - - - - - - - - - - - - -');

    // Поиск кратчайшего пути по карте,
    // построенной методом Флойда-Уоршелла
    Writeln(Gr.GetQuickPathStr(Node1, Node2));
    Writeln('- - - - - - - - - - - - - - - - - -');

    // Сравнение быстродействия двух методов

    Start:= Now;  N1:=0;
    repeat
      for i:=1 to 1000 do begin
        P:= Gr.GenDijkstra(Node1, Node2);    // методом Дейкстры
        P.Free;
      end;
      Inc(N1);
      Time:= MilliSecondsBetween(Start, Now);
    until Time>1000;

    Start:= Now;  N2:=0;
    repeat
      for i:=1 to 1000 do begin
        P:= Gr.GenQuickPath(Node1, Node2);   // по маршрутной карте
        P.Free;
      end;
```

```
      Inc(N2);
      Time:= MilliSecondsBetween(Start, Now);
    until Time>1000;
    // Вывод соотношения быстродействий
    Writeln('QuickPath / Dijkstra = ', N2/N1:5:2);

  until false;
  Gr.Free;
end.
```

Результаты говорят о том, что даже на небольшом графе поиск маршрутов по распределённой карте даёт многократный выигрыш во времени.

15.4. Итоги

- Кратчайший путь между двумя вершинами — это последовательность лежащих на нём вершин и дуг.
- Кратчайший путь может определяться без учёта веса дуг и промежуточных вершин, либо с их учётом (Дейкстра).
- При поиске кратчайших путей используют вспомогательное средство — очередь обрабатываемых вершин.
- Для многократных вычислений кратчайших путей между произвольными парами вершин предварительно строят вспомогательную «карту» дальних указателей методом Флойда-Уоршелла.

15.5. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
√	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 29
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 16
Сильные связи

В предыдущих главах найдены способы определить: а) факт достижимости вершин, б) расстояние между вершинами, и в) кратчайшие пути между ними. Теперь зададимся следующими вопросами:

- Все ли вершины графа взаимно достижимы?
- Если взаимно достижимы не все вершины, то, сколько взаимно достижимых (связанных) областей содержится в графе? И как найти эти области?

В неориентированном графе действует закон: если из вершины *X* достижима вершина *Y*, то верно и обратное. В орграфах взаимная достижимость не гарантирована.

16.1. Неориентированный граф

Вернёмся на Банановые острова, где нередко свирепствуют тайфуны, разрушающие мосты. Вот карта островов после очередного тайфуна, когда часть мостов разрушена, но остальные допускают двустороннее движение (рис. 16-1).

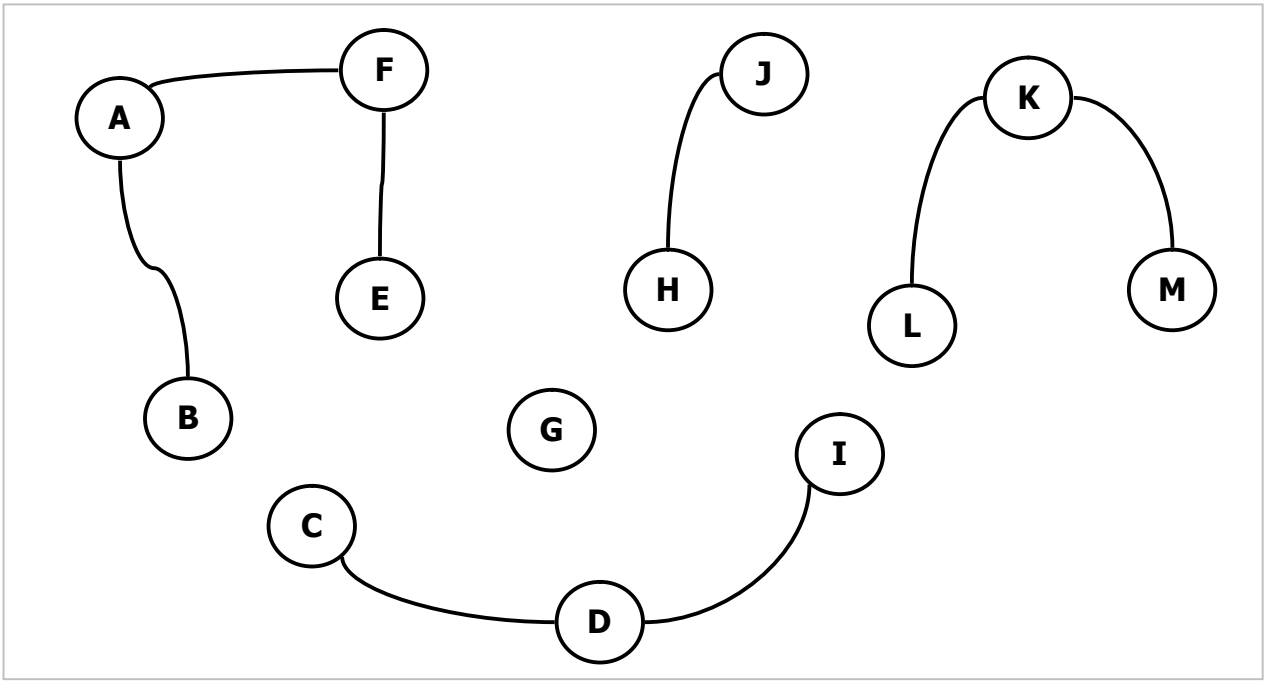


Рис. 16-1 — Соединение островов мостами с двусторонним движением

Легко догадаться, что жители островов *A-B-E-F* могут достичь друг друга по мостам — это *связная компонента* графа. Другие связные компоненты образуют подмножества вершин *C-D-I*, *G*, *H-J*, *K-L-M*, всего имеется 5 связных компонент. Поищем способ подсчёта связных компонент, пригодный для любого типа графов.

16.2. Ориентированный граф

Новый губернатор Банановых островов по имени Гордон Крик был неприятно удивлён жалобами туземцев на транспортные проблемы. Вступая в должность, Гордон изучил карту архипелага, и был уверен, что после очередного тайфуна все острова сохранили взаимную связь по уцелевшим мостам (Рис. 16-2).

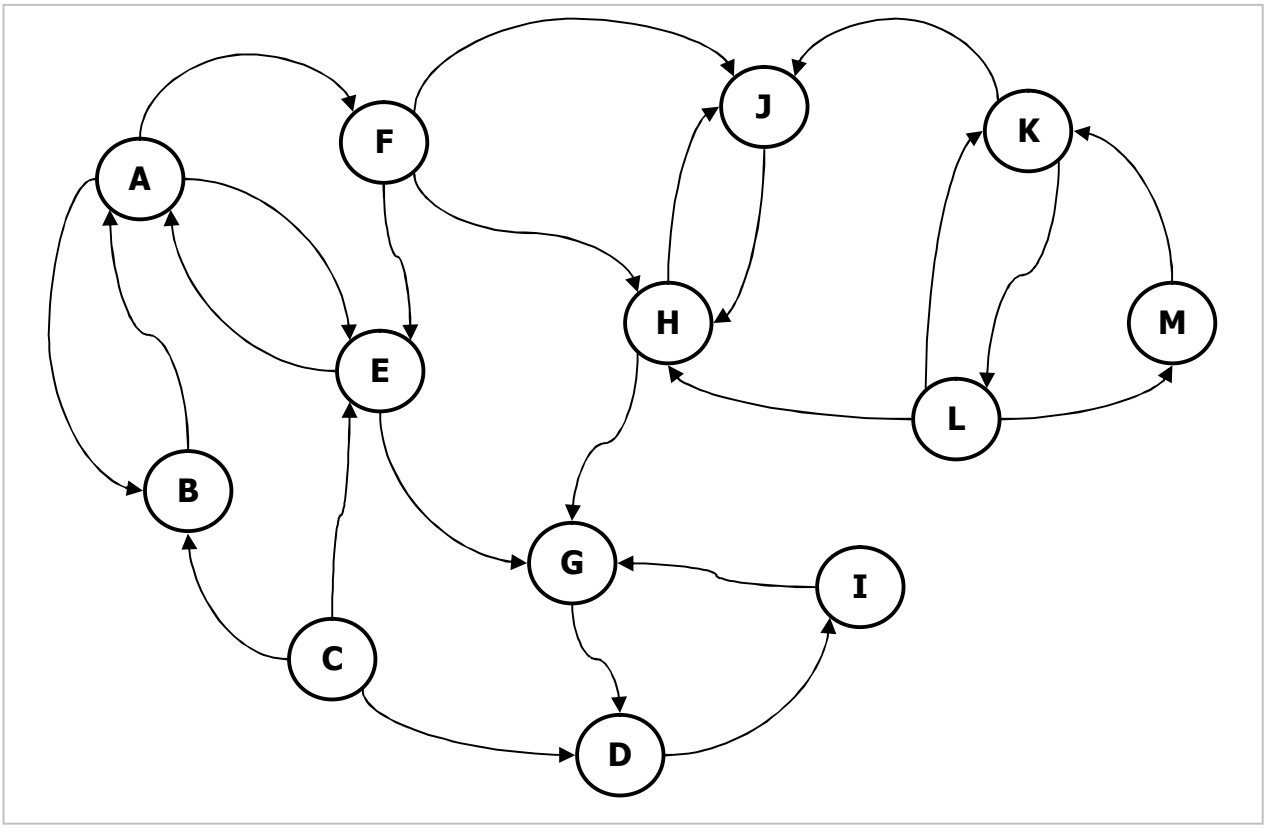


Рис. 16-2 — Мостовая сеть Банановых островов

Отчасти так оно и было, за исключением одной детали: узкие мосты допускали лишь одностороннее движение. И оказалось, к примеру, что хотя житель острова *A* и мог доехать к своему дяде на остров *H* посуху, но возвращаться приходилось морем. А к тёще, обитающей на острове *M*, он вообще не мог доехать по мостам. Впрочем, как и она к нему.

Присоединимся к группе «банановых» математиков, и займёмся решением мостовой проблемы. Предстоит выделить группы островов (подмножества вершин), допускающие взаимные поездки по мостам. Искомые подмножества вершин называют *сильными компонентами* графа. Выявив эти компоненты, можно определить места для постройки недостающих мостов.

16.3. Связная компонента неориентированного графа

Возьмём произвольную вершину графа и составим множество всех *достижимых из неё* вершин, включая и саму эту вершину — ранее это множество было названо *исходящей интегральной гаммой*. Ввиду взаимной достижимости вершин неориентированного графа это подмножество будет одной из связных

компонент. Алгоритм получения такой компоненты реализован в методе **TNode.GenGammaOut**.

Листинг 16-1 — Накопление исходящей интегральной гаммы

```
function TNode.GenGammaOut: TSet;
var Node: TNode;
    Temp: TSet;
begin
  Result:= CreateSet;
  Result.Insert(Self);       // расширение гаммы с текущей вершины
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);            // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;     // начало перебора вершин
    while Assigned(Node) do begin
      Node.OutGammaAdd(Result);        // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;
```

Построение всех таких компонент будет рассмотрено позднее.

16.4. Сильно связная компонента орграфа

Определив в орграфе *исходящую* интегральную гамму, как это было сделано выше, мы не можем гарантировать взаимной достижимости всех её вершин. Построим для той же исходной вершины *входящую* интегральную гамму — множество вершин, из которых она достижима.

Листинг 16-2 — Накопление входящей интегральной гаммы

```
function TNode.GenGammaIn: TSet;
var Node: TNode;
    Temp: TSet;
begin
  Result:= CreateSet;
  Result.Insert(Self);       // начианем расширение гаммы с текущей вершины
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);            // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;     // начало перебора вершин
    while Assigned(Node) do begin
      Node.InGammaAdd(Result);         // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;
```

Далее рассудим так: если какие-то вершины принадлежат одновременно обеим гаммам, то очевидно, что все такие вершины взаимно достижимы и образуют *сильную компоненту* графа. Формирование такой компоненты реализовано в методе класса **TNode**:

Листинг 16-3 — Формирование сильной компоненты графа

```
function TNode.GenStrongArea: TSet;
var Temp: TSet;
begin
  Result:= GenGammaOut;                 // множество достижимых вершин
  if not mOwner.mDirect then Exit;      // выход, если это не орграф
  Temp:= GenGammaIn;                    // множество, из которых достижима
  Result.Mul(Temp);                     // Результат = пересечение множеств
  Temp.Free;                            // освобождаем временное множество
end;
```

Стало быть, одна из сильных компонент графа находится на пересечении исходящей и входящей интегральной гамм любой из его вершин. Это иллюстрируется следующими рисунками, где исходной выбрана вершина *A*. Тот же результат получится при выборе в качестве исходной вершин *B*, *E* или *F*.

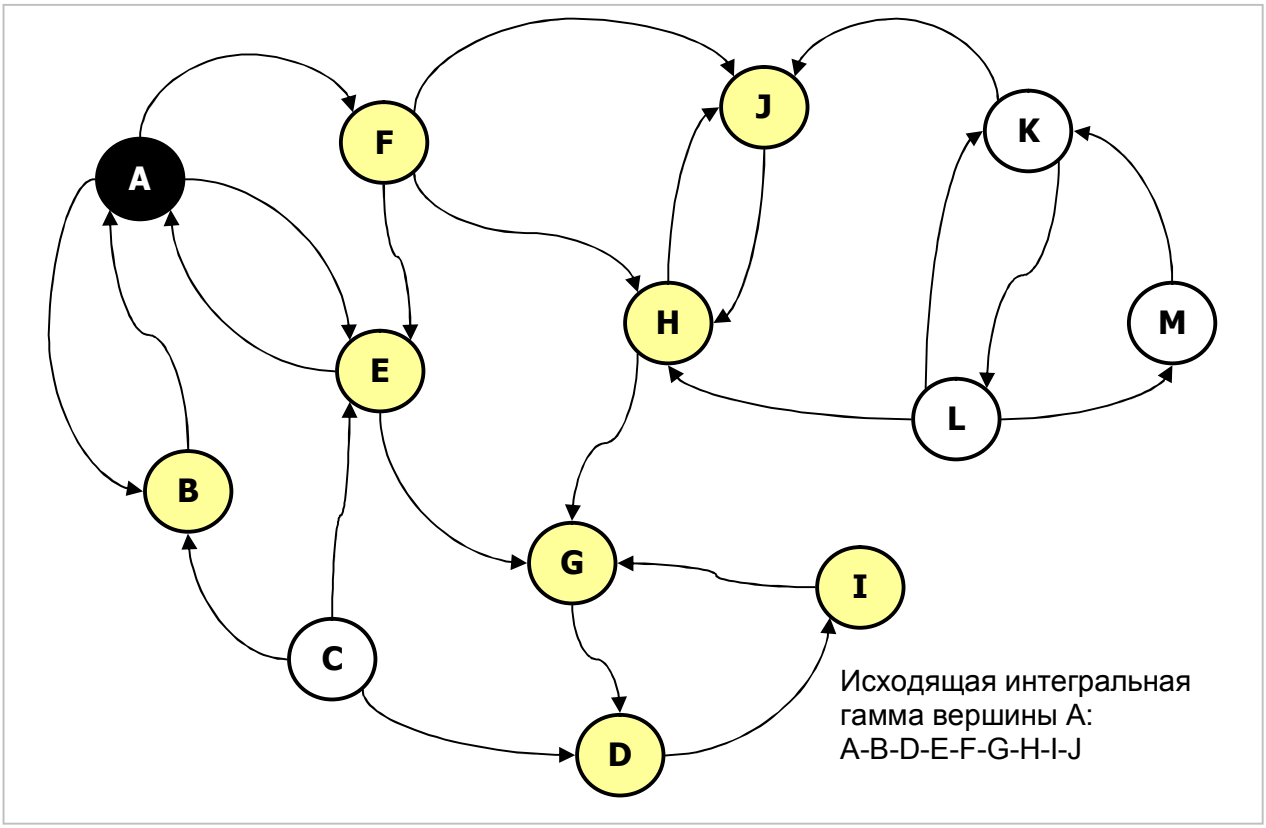


Рис. 16-3 — Вершины, достижимые из вершины A (исходящая гамма)

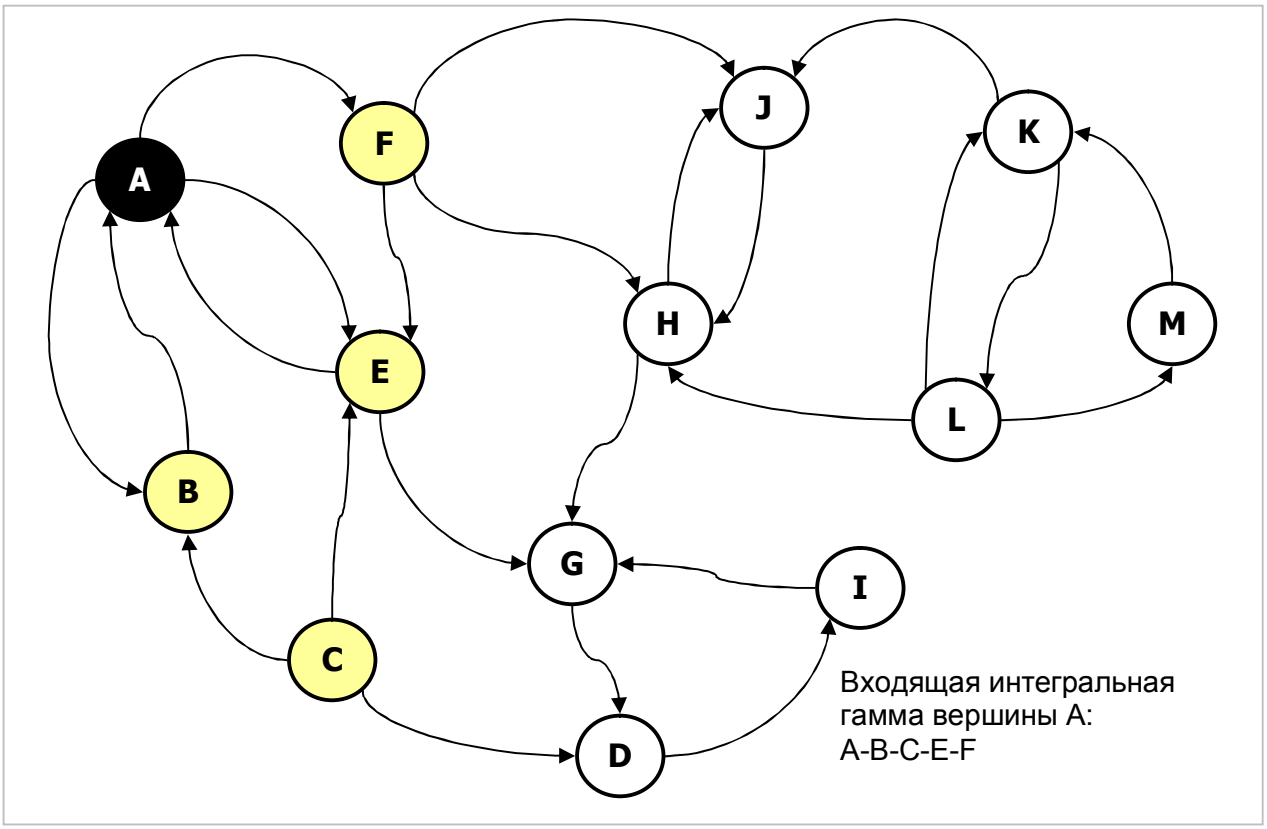


Рис. 16-4 — Вершины, из которых достижима вершина A (входящая гамма)

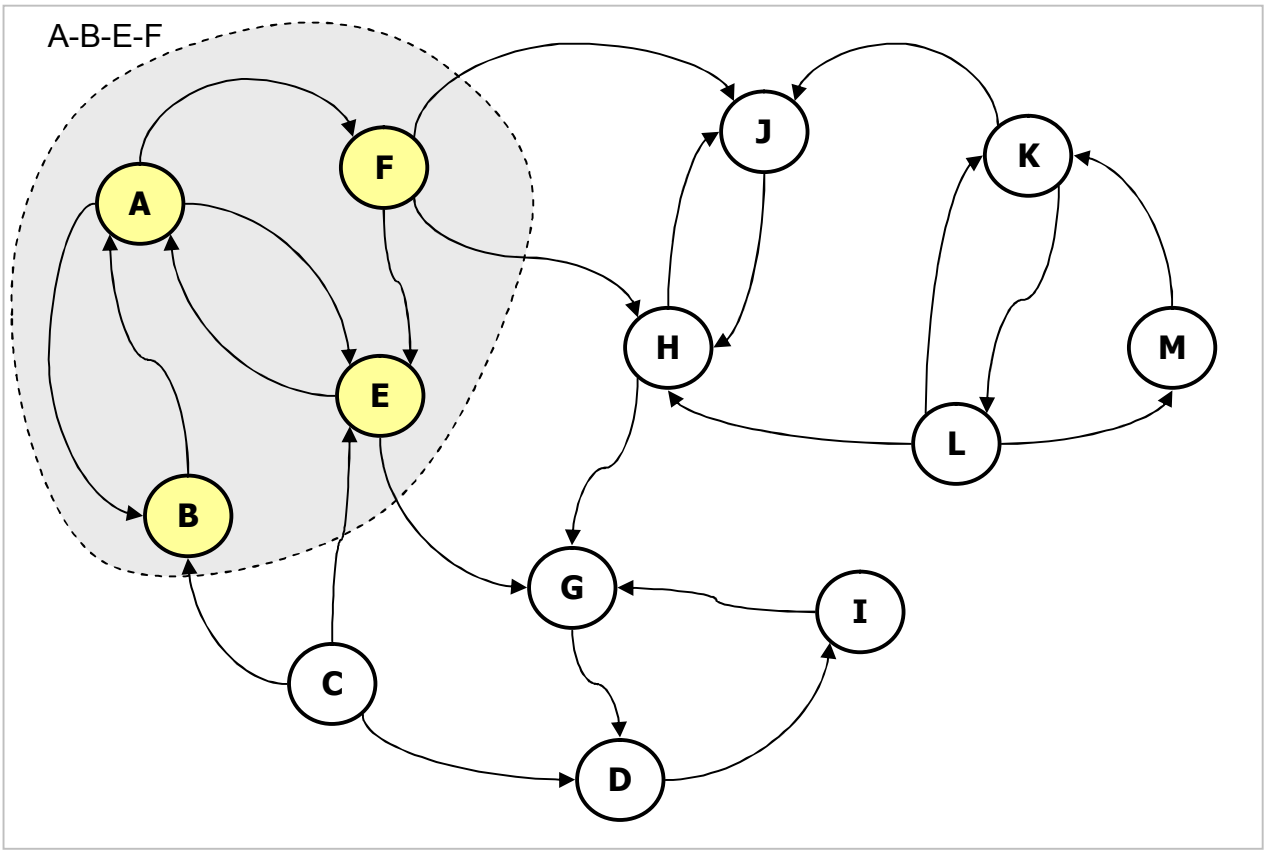


Рис. 16-5 — Одна из сильных компонент графа A-B-E-F

16.5. Поиск всех сильных компонент

Итак, найдя одну *сильную компоненту связности* для некоторой вершины орграфа или графа, попытаемся найти все такие компоненты. Создадим множество-копию всех вершин графа (эта копия будет постепенно очищаться). Выбрав в копии одну из вершин, найдём соответствующую ей сильную компоненту. Удалив эту компоненту из копии, вновь возьмём из копии одну из оставшихся вершин, и повторим те же действия, пока копия не опустеет. Альтернативой копии вершин может быть подходящая окраска вершин.

Листинг 16-4 — Подсчёт всех сильно связанных компонент

```
function TGraph.CalcAreas: integer;
var Copy: TSet;     // копия множества всех вершин графа
    Node: TNode;    // текущая вершина
    Strong: TSet;   // область сильной связи текущей вершины
begin
  Result:= 0;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Inc(Result);                      // счётчик областей связности
    Node:= Copy.GetFirst as TNode;    // первая вершина из копии
    Strong:= Node.GenStrongArea;      // область её связности
    Copy.Sub(Strong);                 // вычитаем из копии
    Strong.Free                       // освобождаем область
  until Copy.GetCount = 0;            // пока копия не пуста
  Copy.Free;
end;
```

Следующий метод возвращает множество всех связных областей в виде множества подграфов, порождённых на сильно связанных вершинах:

Листинг 16-5 — Порождение подграфов из сильно связанных компонент

```
function TGraph.GenAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    Node: TNode;    // текущая вершина
    Strong: TSet;   // область сильной связи текущей вершины
    Gr: TGraph;     // граф, порождённый на множестве вершин Strong
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Node:= Copy.GetFirst as TNode;    // первая вершина из копии
    Strong:= Node.GenStrongArea;      // область её связности
    Gr:= CopyByNodes(Strong);         // создание порождённого подграфа
    Result.Insert(Gr);                // вставка порождённого подграфа
    Copy.Sub(Strong);                 // вычитаем область из копии
    Strong.Free                       // освобождаем область
  until Copy.GetCount = 0;            // пока копия не пуста
  Copy.Free;
end;
```

Здесь применён метод порождения подграфа из подмножества вершин:

Листинг 16-6 — Метод порождения подграфа из подмножества вершин

```
function TGraph.CopyByNodes(aNodes: TSet): TGraph;
var  N1, N2 : TNode;     // исходный узел и его копия
         ND : TNode;     // приёмник в копии
          L : TLink;
begin
  Result:= TGraph.Create('CopyByNodes', mDirect, mLoadNodes, mLoadLinks);
  // Копирование вершин:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin  // если существует в множестве аргумента
      N2:= N1.Copy as TNode;         // то создаём копию
      Result.InsertNode(N2);         // и вставляем в граф
    end;
    N1:= NodeNext;
  end;
  // Копирование связей:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin  // если существует в множестве аргумента
      N2:= Result.GetNode(N1);       // узел копии
      L:= N1.OutLinkFirst;           // связь в исходном
      while Assigned(L) do begin     // перебор связей исходного
        ND:= Result.GetNode(L.mDest);      // приёмник в копии
        Result.SetLink(N2, ND, L.mValue); // связать узлы
        L:= N1.OutLinkNext;          // следующая связь
      end;
    end;
    N1:= NodeNext;    // следующий узел исходного
  end;
end;
```

Для проверки описанных выше методов служит следующая программа:

Листинг 16-7 — Подсчёт и формирование областей связности

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas';

var Gr : TGraphChars;
    N: integer;
    Areas: TSet;
    S: string;

begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 1, 1, 20, 40);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - -');
    N:= Gr.CalcAreas;
    Writeln('Areas count= ', N);
    Writeln('- - - - - - - - - - - - - - - - - -');
    Areas:= Gr.GenAreas;
    if Assigned(Areas) then begin
      Areas.Expo;
```

```
      Areas.ClrAndDestroy;
      Areas.Free;
    end;
    Writeln('- - - - - - - - - - - - - - - - - -');
    Gr.Free;
    Readln(S);
  until S<>'';
end.
```

16.6. Конденсация сильных компонент

Итак, найдя все сильные компоненты графа (рис. 16-6), то есть группы взаимно достижимых островов, попробуем найти такие острова, соединение которых мостами с односторонним движением полностью свяжет архипелаг. Разумеется, что для экономии средств количество новых мостов должно быть минимально.

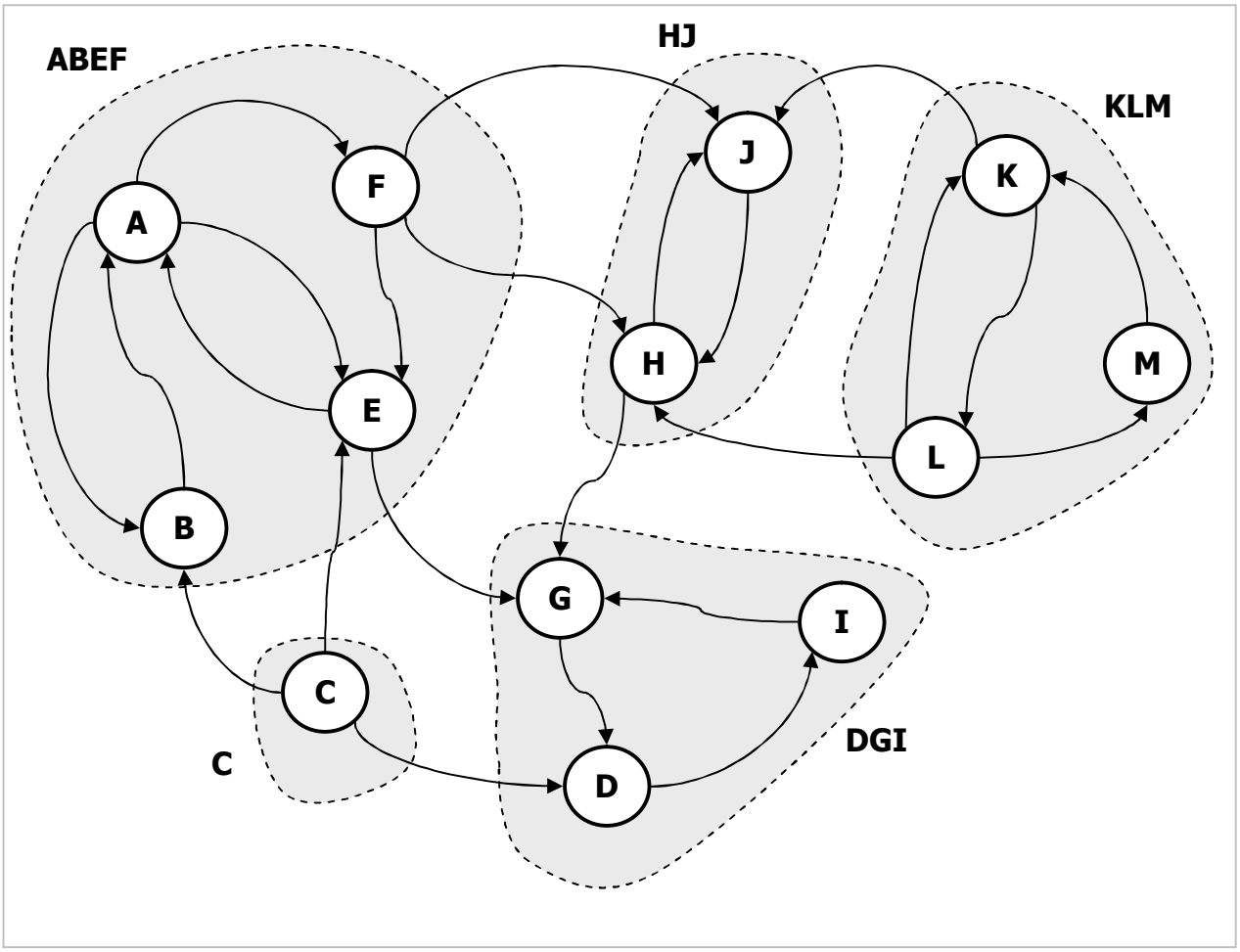


Рис. 16-6 — Сильные компоненты графа

Взяв полученное ранее множество сильных компонент, создадим из него новый граф. В этом графе сильные компоненты станут вершинами. Этот новый тип вершин, содержащих сильную компоненту, описан через класс **TCondensNode**. Здесь ещё раз проявляется универсальность принятого объектного подхода к построению графов.

```
TCondensNode = class (TNode)
    public
      mName  : string;// имя = сумма входящих в компоненту имён вершин
      mNodes : TSet;  // собственные вершины
      mOut   : TSet;  // ссылки за пределы сильной компоненты (вершины)
      constructor Create(aNodes: TSet; aOwner: TGraph);
      destructor Destroy;  override;
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;
```

Связи между вершинами нового графа установим на основе связей исходного графа, но так, чтобы между каждой парой компонент их было не более одной (без дублирующих связей). На рис. 16-7 они показаны пунктирными стрелками.

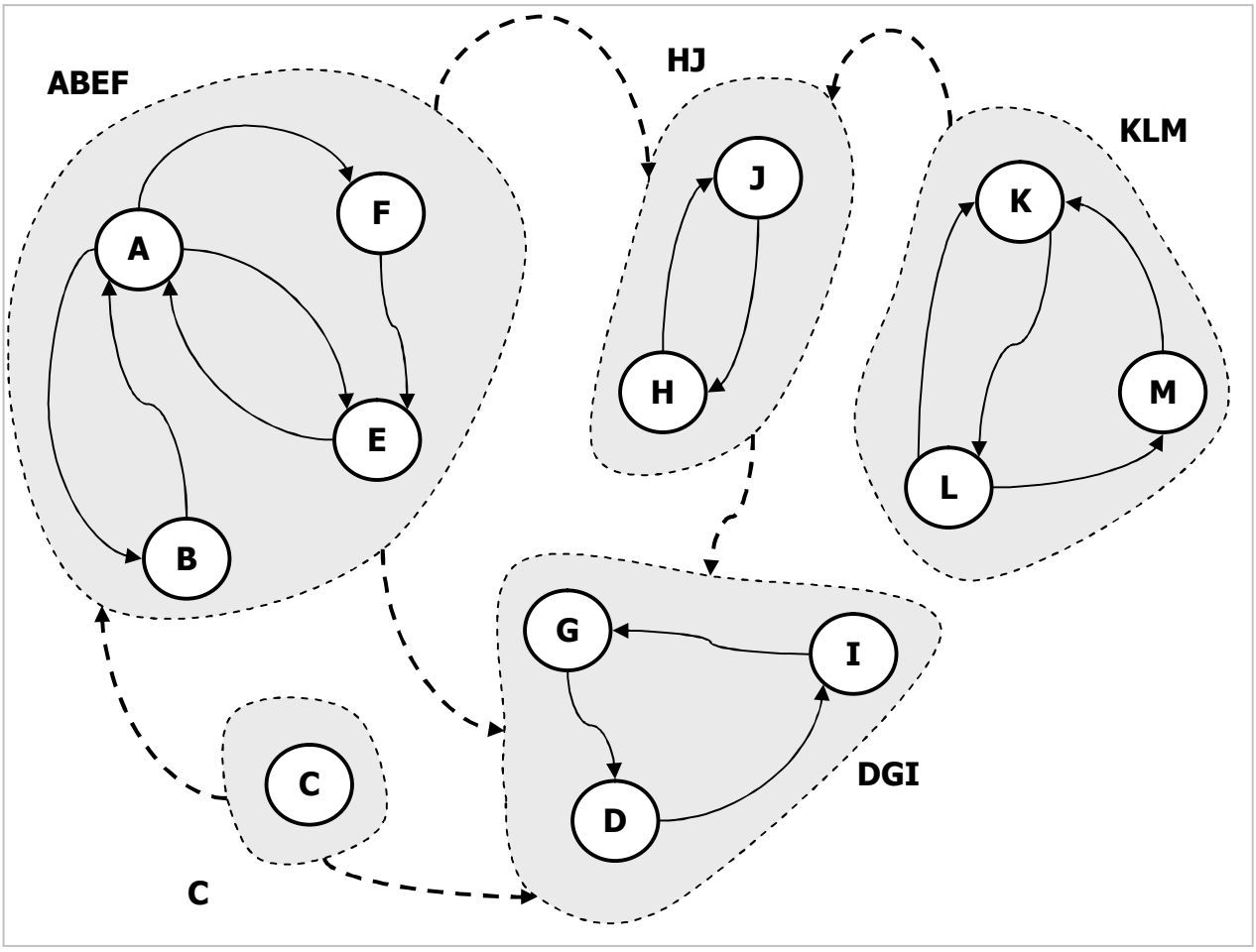


Рис. 16-7 — Подграфы, порождённые сильными компонентами

В результате построений получен граф-*конденсат*, изображённый на рис. 16-8.

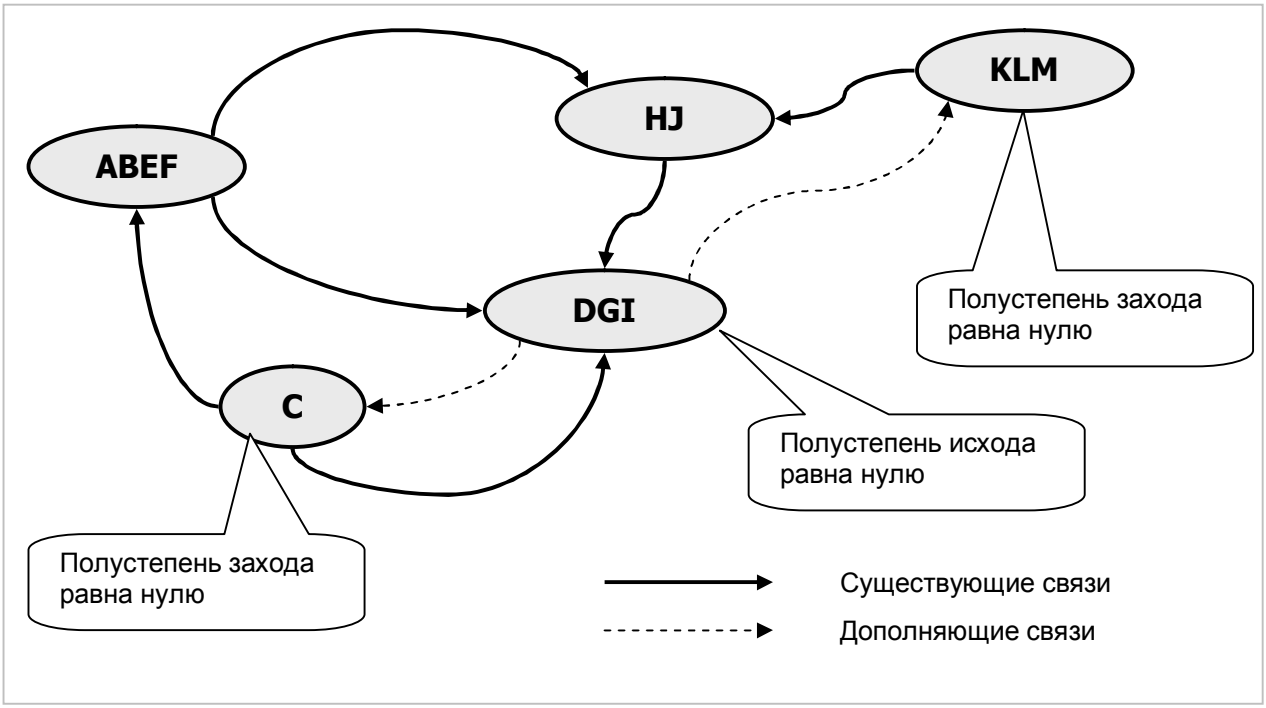


Рис. 16-8 — Конденсация сильных компонент

Обратите внимание на полу-степени исхода и захода вершин конденсата. Вершины *C* и *KLM* имеют нулевые степени захода (нет входящих дуг), — значит попасть сюда из других вершин невозможно. Наоборот, вершина *DGI* имеет нулевую степень исхода (здесь нет входящих дуг) — это ловушка, из которой нет выхода. Как добиться полной связности архипелага минимальными средствами? Очевидно, надо построить исходящие связи (мосты), ведущие из ловушки *DGI* в недостижимые пока «медвежьи углы» *C* и *KLM*. Для этого достаточно построить не более двух мостов: между любыми островами из множества *DGI* к острову *C*, и между теми же островами и любыми из множества *KLM*. Эти дополняющие связи обозначены пунктиром. Вот методы, реализующие описанные выше алгоритмы.

Листинг 16-8 — Формирование множества-конденсата

```
function TGraph.GenStrongAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    N : TNode;
    Strong: TSet;   // сильная компонента
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Пока копия не пуста:
  while Copy.GetCount <> 0 do begin
    N:= Copy.GetFirst as TNode;       // первая вершина в копии
    Strong:= N.GenStrongArea;         // находим сильную компоненту
    Result.Insert(Strong);            // и вставляем в результат
    Copy.Sub(Strong);                 // удаляем её из копии
  end;
  Copy.Free;  // освобождаем копию
end;
```

Следующий метод пользуется предыдущим и строит новый граф — конденсат исходного графа.

Листинг 16-9 — Создание графа-конденсата

```
function TGraph.GenCondens: TGraph;
var Strongs : TSet;          // множество сильных компонент
          S : TSet;          // вспомогательное множество
         CN : TCondensNode; // текущий узел-конденсат
     Target : TCondensNode; // целевой узел-конденсат
begin
  Result:= TGraph.Create('Condens', true, false, false);  // пустой граф
  Strongs:= GenStrongAreas;                 // формируем множество-конденсат
  S:= Strongs.GetFirst as TSet;             // подмнож. вершин в узле конденс.
  while Assigned(S) do begin                // перебор множеств конденсата
    CN:= TCondensNode.Create(S, Result);   // создаём вершину конденсата
    Result.InsertNode(CN);                  // и вставляем в граф-конденсат
    S:= Strongs.GetNext as TSet;            // подмнож. вершин в узле конденс.
  end;
  // формирование связей между вершинами конденсата:
  S:= CreateSet;                // вспомогат. множество вершин исходного графа
  CN:= Result.NodeFirst as TCondensNode;        // вершина конденсата
  while Assigned(CN) do begin                   // перебор вершин конденсата
    if CN.mOut.GetCount <> 0 then begin         // если степень исхода <> 0
      Result.PosPush;                           // сохр. позицию перебора
      Target:= Result.NodeFirst as TCondensNode;// вершина конденсата
      while Assigned(Target) do begin           // внутр. цикл перебора
        if CN <> Target then begin              // если не текущая вершина
          S.CopyItems(CN.mOut);                 // исходящие связи текущей
          S.Mul(Target.mNodes);                 // пересекаем с целевыми
          if S.GetCount<>0                      // если связаны в исх. графе
            then Result.SetLink(CN, Target, 1); // то уст. связь в конденсате
        end;
        Target:= Result.NodeNext as TCondensNode; // след. узел конденсата
      end;
      Result.PosPop;                        // восст. позицию перебора
    end;
    CN:= Result.NodeNext as TCondensNode;  // след. узел конденсата
  end;
  S.Free;
  Strongs.ClrAndDestroy;  // очищаем множество-конденсат
  Strongs.Free;
end;
```

Дополнительная информация об узле-конденсате **TCondensNode** содержится в листинге модуля **Graph**. Следующая программа служит для проверки метода постройки конденсата.

Листинг 16-10 — Тестирующая программа для проверки формирования конденсата

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas';
```

```
var Gr : TGraphChars;
    Condens: TGraph;  // граф-конденсат
    S: string;

begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 1, 1, 20, 40);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - -');
    Condens:= Gr.GenCondens;
    Condens.Expo;
    Condens.Free;
    Writeln('- - - - - - - - - - - - - - - - - -');
    Gr.Free;
    Readln(S);
  until S<>'';
end.
```

16.7. Анализ информационных потоков

В книге [7] дан пример анализа информационных потоков посредством конденсата. Пусть карта изображает не мосты, а пути распространения информации между островами. В конденсированном виде этот граф представлен на рис. 16-9.

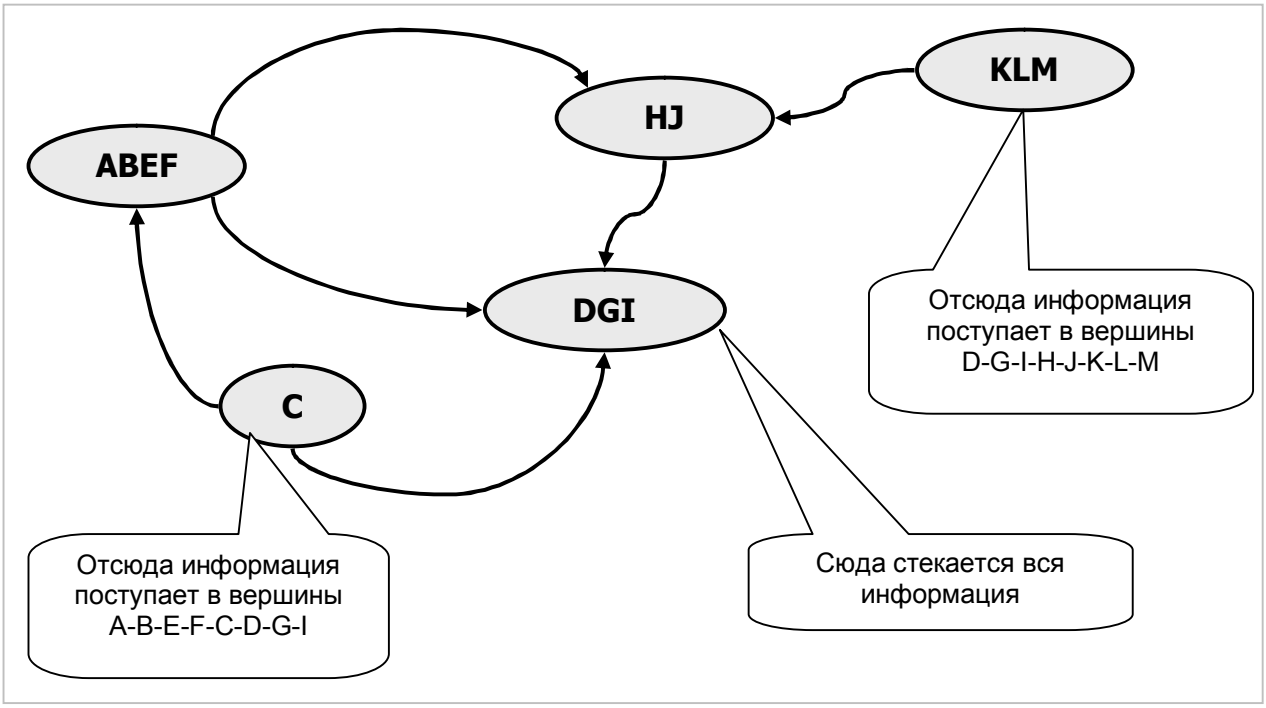


Рис. 16-9 — Конденсированный граф путей распространения информации

Внутри конденсатов острова обмениваются информацией без ограничений, а за их пределами — по указанным на рисунке дугам. Пусть информацией будут некие управляющие команды, и надо выбрать минимальное подмножество островов, из которых такие команды могут попасть ко всем остальным. Выберем вершины конденсата с нулевыми степенями захода: это *C* и *KLM*. Из первой достижимы восемь вершин: **A-B-E-F-C-D-G-I**, из второй другие восемь: **D-G-I-H-J-K-L-M**. Это даёт совокупность всех вершины графа, стало быть, для

управления архипелагом (или рассылки спама) достаточно взять остров *C* и любой из островов *K*, *L* или *M*. Полученные подмножества называют *базами* графа, в данном графе имеется три базы: **C-K**, **C-L, C-M**.

Теперь положим, что некие данные добываются в вершинах графа и распространяются теми же путями. Вершина *DGI* с нулевой степенью исхода отличается тем, что сюда стекается вся информация, вырабатываемая в вершинах графа. Таким образом, центр сбора и обработки информации надо разместить на одном из островов *D*, *G*, или *I*. Минимальные множества вершин, достижимых из любой вершины графа называют *антибазой* графа. Стало быть, в данном графе существуют три антибазы, каждая из которых состоит из одной вершины.

Пусть теперь граф изображает потоки информации и команд в некой организации, а вершины представляют собою людей. Тогда любая из *баз* даст минимальный круг *могущественных* лиц, способных управлять всеми другими. Одна из *антибаз* будет являть собой минимальный круг вполне *осведомлённых* лиц. Пусть требуется составить управляющий *комитет* такой, чтобы в него входили в минимальном числе и *осведомлённые* и *могущественные* лица. Очевидно, что для этого следует объединить одну из *баз* и одну из *антибаз*. В данном примере можно сформировать девять вариантов таких комитетов (3x3).

16.8. Итоги

- Подмножество частично достижимых вершин орграфа называется *связной* компонентой.
- Подмножество *взаимно* достижимых вершин называется *сильной* компонентой орграфа.
- В неориентированном графе вершины одной связной компоненты взаимно достижимы, для таких графов *связная* компонента и *сильная* представляют одно и то же.
- Сильная компонента орграфа находится на пересечении подмножеств исходящей и входящей интегральных гамм некоторой вершины. Все такие компоненты находятся последовательным исключением из полного множества вершин найденных сильных компонент.
- Граф, вершинами которого являются сильные компоненты, называется *конденсатом*. Конденсат — это мощное средство исследования орграфов.

16.9. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
√	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 17
Независимые вершины и клики

Здесь будут рассмотрены *независимые вершины* и *клики* — два взаимно противоположных, но тесно связанных понятия, имеющих отношение к вершинам *неориентированного* графа.

17.1. Знакомство

Кликой (*Clique* — фр.) называют тесно сплочённую группу людей. Вот в вагоне электрички оживлённо беседует небольшая группа людей, заметно, что все они хорошо знакомы, — это *клика*. Другие пассажиры этого вагона молча смотрят по сторонам, не пытаясь общаться с незнакомыми соседями, — они подобны подмножеству *независимых вершин*, о которых тоже пойдёт речь. Подобием *клики* является забрасываемая в тыл противника группа фронтовых разведчиков. Командир такой группы набирает только хорошо знакомых между собой бойцов.

17.1.1. О кликах

Изобразим схему взаимных знакомств неориентированным графом (рис. 17-1). Здесь два лица знакомы друг с другом, если вершины связаны ребром. Так, к примеру, у гражданина *A* нет знакомых, а гражданин *E* знаком с гражданами *C* и *D*, но не знаком с остальными.

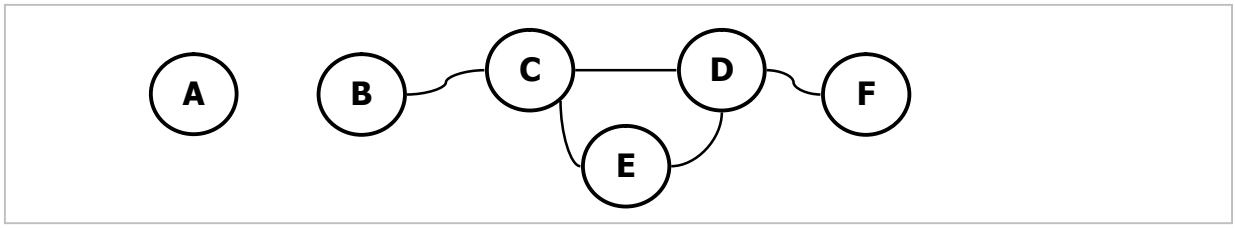


Рис. 17-1— Схема взаимных знакомств

Окажись в одном вагоне лица *C*, *D* и *E*, они создали бы клику — подмножество вершин, непосредственно связанных друг с другом (рис. 17-2).

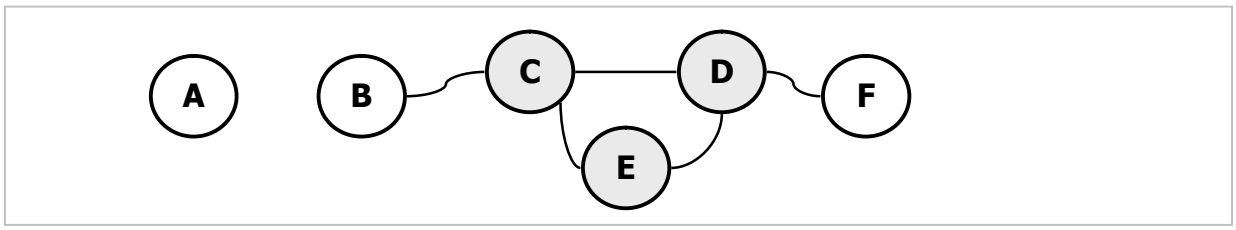


Рис. 17-2 — Клика C-D-E

В отсутствие лица *E*, могли бы образоваться менее мощные клики: *B-C*, *C-D* или *D-F*. Отметим, что тройки *B-C-D* или *C-D-F* не могут быть кликами, поскольку в первой не связаны между собой *B* и *D*, а во второй — *C* и *F*.

Далее будем искать все возможные клики графа, а точнее, все возможные *максимальные* клики. Максимальными они будут в том смысле, что такие клики уже нельзя расширить другими вершинами. При постройке всех максимальных клик следует перепробовать в качестве начальных вершин («командиров») все вершины графа.

Первую клику строим с вершины *A*. Поскольку она изолирована, то останется единственной в этой клике. Вторую клику строим от вершины *B*, к ней можно присоединить *C*, и на этом расширение клики *B-C* закончится. Сборку третьей клики начнём с *C*, очевидно, что далее к ней присоединятся *D* и *E*, образуя клику *C-D-E*. Наконец, начав постройку с вершины *D*, получим клику *D-G*. В итоге получаем четыре максимальных клики:

- *A*
- *B-C*
- *C-D-E*
- *D-F*

Все они максимальны, поскольку не могут быть расширены. В то же время клика *C-D-E* является среди них максимальной по мощности. Отметим, что ни одно из этих множеств вершин не является подмножеством другого (иначе оно не было бы максимальным).

17.1.2. О независимых вершинах

Подмножеству независимых вершин (*независимых множеств*, как их называют), так же, как и кликам, есть аналогия в разведке. Для создания глубоко законспирированной агентурной сети в целях безопасности вербуют взаимно незнакомых людей. Тогда, в случае провала одного, он не выдаст остальных. Проделаем подобную «вербовку» с тем же множеством вершин (рис. 17-1).

Для начала включим в нашу сеть вершину *A*. Поскольку она «не знакома» с другими вершинами, сеть можно расширить за счёт любой из оставшихся вершин, например, *B*. Теперь вершина *C* — сосед *B* — не годится для расширения сети, однако можно «завербовать» *D*, *E* или *G*. Продолжив «вербовку» в алфавитном порядке, присоединим вершину *D*. С этого момента дальнейшее расширение сети стало невозможным, поскольку оставшиеся вершины (*C*, *E*, *F*) «знакомы» хотя бы с одним участником сети. Так сформировалось одно из максимальных подмножеств независимых вершин: *A-B-D* (рис. 17-3).

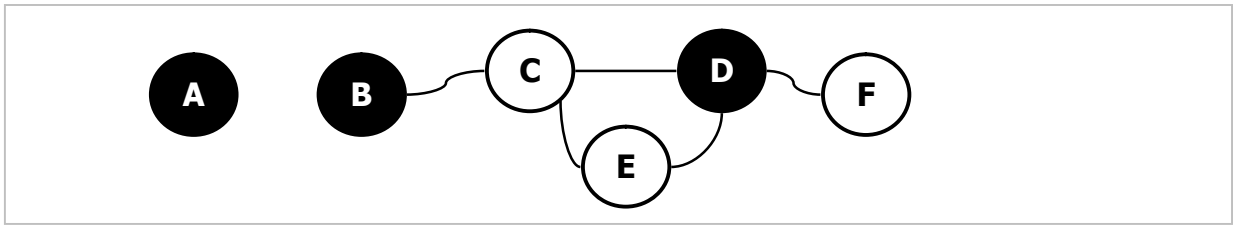


Рис. 17-3 — Независимые вершины A-B-D

Другие максимальные подмножества независимых вершин строятся аналогично. Например, после *A* можно присоединить не *B*, а *C* и далее *F*, в результате получится другое независимое подмножество: *A-C-F*. На этом небольшом графе легко найти все такие подмножества:

- *A-B-D*
- *A-C-F*
- *A-B-E-F*

Поскольку вершина *A* изолирована, она входит в каждое из этих максимальных независимых подмножеств. Подобно максимальным кликам, каждое из этих подмножеств не является подмножеством другого.

В заключение рассмотрим ещё два примера приложений клик и независимых множеств. Пусть графы на рис. 17-4 изображают группу островов, соединённых мостами с двусторонним движением.

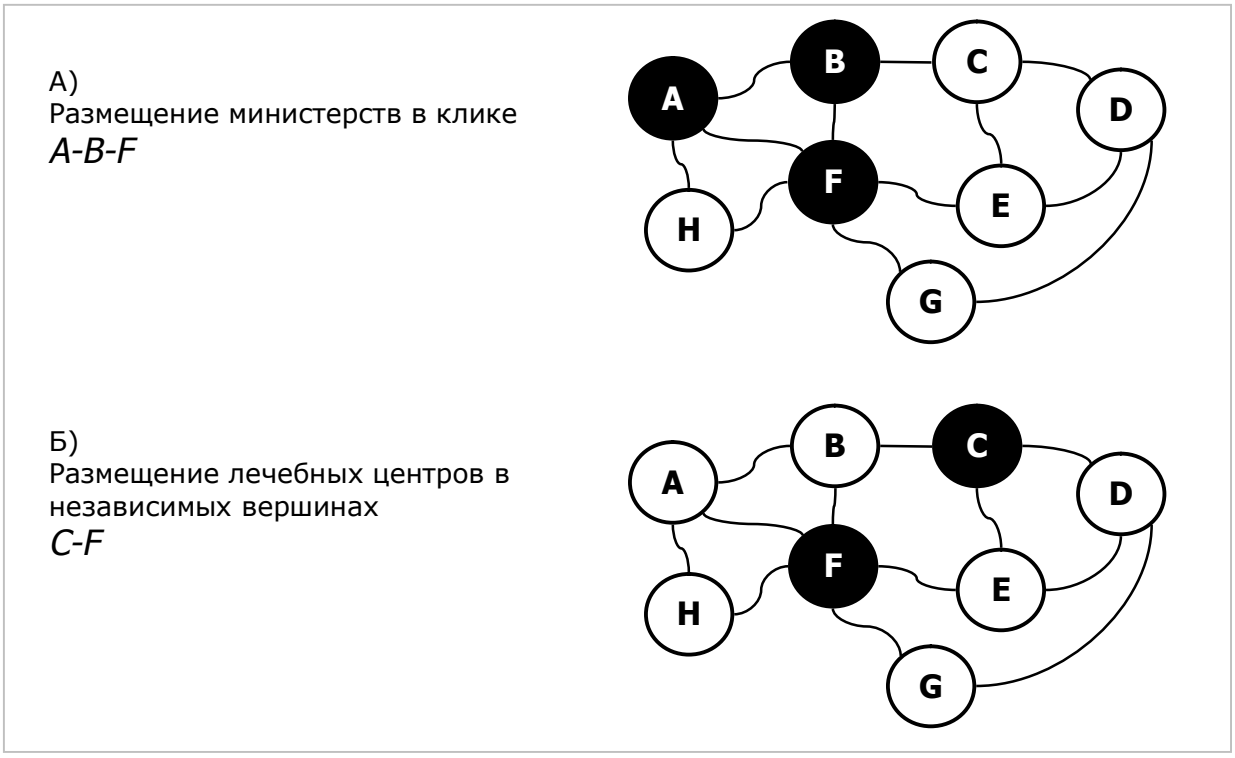


Рис. 17-4 — Размещение министерств (А) и лечебных центров (Б)

Испытывая недостаток места, правительство размещает свои министерства на нескольких островах так, чтобы из любого из них можно было добраться до любого другого, преодолев не более одного моста. Очевидно, что для этого подходит одна из двух максимальных клик: *A-B-F* или *C-D-E* (рис. 17-4 А).

Для быстрого доступа к лечебницам желательно, чтобы пациенты преодолевали не более одного моста. Тут подходит одно из независимых множеств, имеющее наименьшую мощность, например *C-F*. Так можно обойтись минимальным количеством лечебниц (рис. 17-4 Б).

17.1.3. Связь между кликами и независимыми вершинами

Говорят, что крайности сходятся. Независимые множества и клики — тот самый случай; они связаны между собой через понятие *обратного графа*. Напомню, что граф, обратный данному, строится на тех же вершинах путём добавления связей, отсутствующих в данном графе, и удаления имеющихся связей. Пример такого перестроения дан на рис. 17-5.

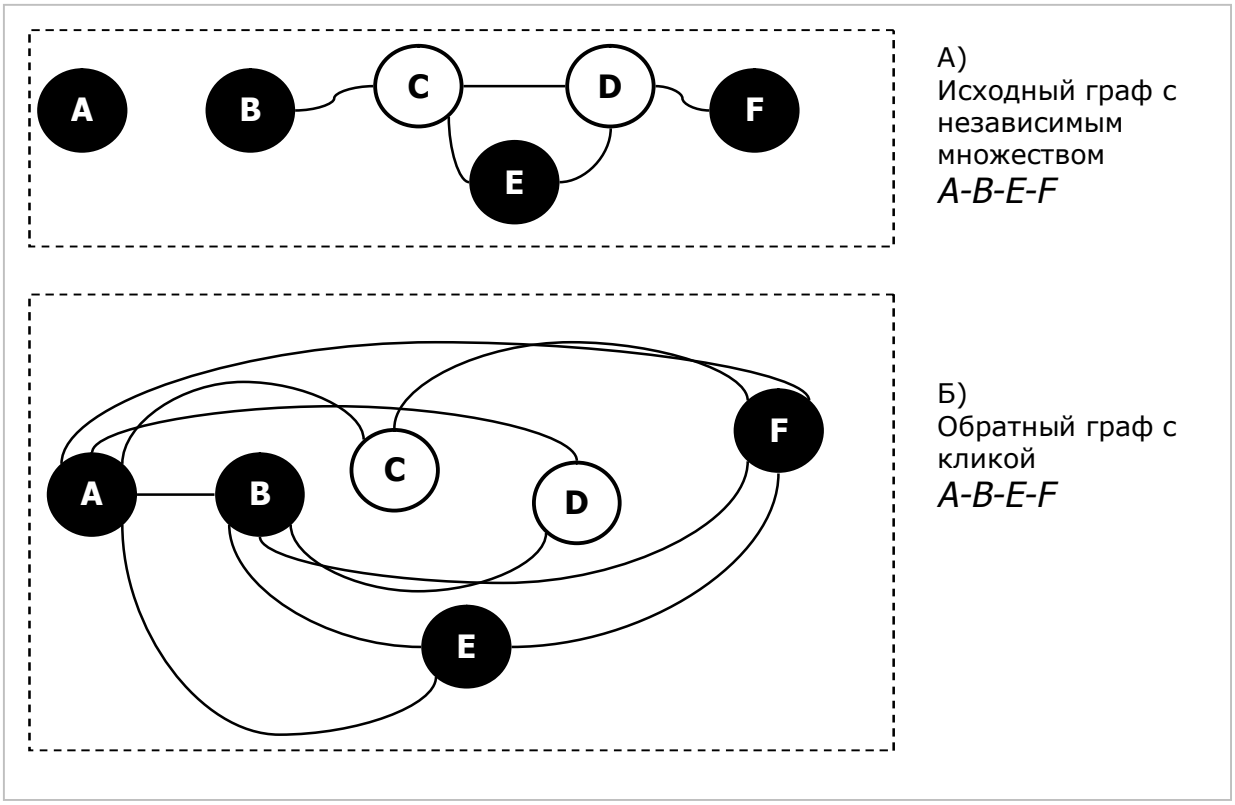


Рис. 17-5 — Исходный и обратный графы

Показано, что клики исходного графа становятся независимыми множествами вершин в обратном графе, и наоборот. Эта связь проявляется и в алгоритмах поиска всех максимальных *клик* и всех *независимых множеств*. По сути это две модификации одного алгоритма от голландцев Брона и Кербоши (*Bron* и *Kerbosh*).

17.2. Алгоритм Брона-Кербоша для поиска независимых вершин

Назначение алгоритма — сформировать множество максимальных подмножеств независимых вершин. Этот результат может быть использован впоследствии для выбора нужного подмножества (минимального или максимального).

17.2.1. Основная идея

Ранее было показано, что все максимальные подмножества ищутся прямым перебором начальных вершин. Возьмём первую вершину, и вставим её в подмножество-результат, после чего удалим вершину и её непосредственных

соседей из исходного множества. К оставшимся вершинам рекурсивно применим те же операции. Когда исходное множество опустеет, будет получено очередное максимальное подмножество независимых вершин. Выбрав на очередном уровне перебора другую вершину-кандидат, получим другую ветвь перебора и другое максимальное подмножество.

Очевидно, что для не сильно плотного графа количество ветвлений будет очень велико, и этот перебор будет чудовищно трудоёмок (NP-сложен).

17.2.2. Улучшение перебора

Брон и Кербош применили дополнительную проверку при переходе к следующему уровню перебора, отсекая не ведущие к цели тупики. В этом алгоритме на каждом уровне перебора используются три множества, которые здесь будут обозначены так:

- **Res** — текущий результат;
- **Cand** — вершины-кандидаты на включение в результат;
- **Tested** — уже обработанные вершины.

Первоначально множество **Cand** содержит все вершины графа, прочие множества будут пусты. Затем из множества кандидатов берётся одна вершина, включается в результат **Res**, и вместе со своими соседями вычёркивается из множества кандидатов. Вдобавок её соседей вычёркивают из множества обработанных вершин **Tested** — смысл этого действия будет объяснен позже. На следующих уровнях перебора (при соблюдении двух условий, обсуждаемых ниже), повторяются те же действия, пока списки **Cand** и **Tested** не опустеют. Тогда в множестве **Res** окажется очередное независимое подмножество. Если же, по исчерпании кандидатов, в **Tested** всё ещё останутся не вычеркнутые вершины, то это значит, что подмножество **Res** не максимально: в нём должно быть что-то из **Tested**, но не попало, — это тупик. В этом случае следует выход на предыдущий уровень перебора, удаление обработанной вершины из **Res** и **Cand**, и перенос её в **Tested**.

Для раннего отсечения тупиковых ветвей при каждом рекурсивном входе проверяются два условия: 1) исчерпаны ли все кандидаты? и 2) граничат ли все обработанные вершины в **Tested** с вершинами в **Cand**? Если хотя бы одна вершина не граничит, то **Tested** никогда не будет очищен, а это тупик. В таком случае данная ветвь отбрасывается, и происходит возврат на предыдущий уровень перебора.

17.2.3. Пример

В табл. 17-1 даны несколько последовательностей формирования независимых подмножеств для графа на рис. 17-1. После обработки вершины *A* и попадания её в множество **Tested** (шаг 2), выбор любой из оставшихся пяти

вершин ведёт в конечном счёте в тупик. Так, на шаге 2-1-1 множество **Res=** *B-D* оказалось подмножеством *A-B-D*, поэтому шаги, отмеченные серым цветом, в действительности выполняться не будут.

Табл. 17-1 — Пример формирования независимых подмножеств

Шаг	Cand	Выбрано	Tested	Res	Примечание
1	A B C D E F	A	-	-	
1-1	B C D E F	B	-	A	
1-1-1	D E F	D	-	A B	
1-1-1-1	-	-	-	A B D	Готово
1-1-2	D E F	E	-	A B	
1-1-2-1	F	F	-	A B E	
1-1-2-1-1	-	-	-	A B E F	Готово
1-2	B C D E F	C	-	A	
1-2-1	F	F	-	A C	
1-2-1-1	-	-	-	A C F	Готово
2	B C D E F	B	A	-	
2-1	D E F	D	A	B	
2-1-1	-	-	A	B D	Тупик

17.2.4. Листинг

Ниже представлен листинг функции **TGraph.GenUndepend**, создающей все максимальные подмножества независимых вершин. Основу метода составляют рекурсивная локальная процедура **Extend** и локальная функция **Test**, проверяющая наличие независимых вершин среди уже проверенных.

Листинг 17-1 — Формирование максимальных подмножеств независимых вершин

```
function TGraph.GenUndepend: TSet;
var
  Res: TSet;     // очередное максимальное независимое множество
  Gamma: TSet;   // для доступа к соседним вершинам
  //-------------------------------------------------------------
  // Возвращает TRUE, если среди обработанных вершин (aTested)
  // есть хоть одна, не соседствующая ни с одним кандидатом (aCand)

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);   // первый из обработанных
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи обработанного узла
      Gamma.Mul(aCand);        // Gamma = пересечение соседей с кандидатами
      if Gamma.GetCount=0 then begin // если пересечение пусто,
        Result:= true;  // значит этот обработанный узел
        Break;          // не соседствует ни с одним кандидатом
      end;
      Node:= TNode(aTested.GetNext);  // следующий из обработанных
    end;
  end;
  //-------------------------------------------------------------
  // Рекурсивная процедура расширения независимого множества
  // aCand, aTested - множества с предыдущего уровня

  procedure Extend(aCand, aTested : TSet);
  var
    newCand : TSet;  // новые вершины-кандидаты
    newTested: TSet; // новые обработанные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и
    // нет независимых от них обработанных вершин
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst);     // взять первого кандидата
      Res.Insert(Node);                 // вставить в результат
      Node.OutGammaGet(Gamma);          // Gamma= его соседи
      // Создать копии множеств кандидатов и проверенных
      newCand:= TSet(aCand.Copy);
      newTested:= TSet(aTested.Copy);

      newCand.Delete(Node);   // удалить текущий узел из кандидатов
      newCand.Sub(Gamma);     // и удалить его соседей
      newTested.Sub(Gamma);   // удалить соседей из копии проверенных

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
             // то найдено очередное макс. независимое множество
             Result.Insert(Res.Copy)
        else if newCand.GetCount<>0 then
               // если остались кандидаты, то рекурсивный вызов
               // с новыми наборами кандидатов и проверенных
               Extend(newCand, newTested);
      // Возвращение
      newTested.Free;       // удалить копию проверенных
      newCand.Free;         // удалить копию кандидатов
```

```
      Res.Delete(Node);      // удалить текущий из результата
      aCand.Delete(Node);    // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end; // while
  end;
  //-------------------------------------------------------------
var
  Cand : TSet;    // вершины-кандидаты
  Tested: TSet;   // проверенные вершины

begin
  // Инициализация
  Result:= CreateSet;       // пустое множество-результат
  Res:= CreateSet;          // очередное макс. независимое множество
  Cand:= CreateSet;         // кандидаты
  Cand.CopyItems(mNodes);   // Cand = все вершины графа
  Tested:= CreateSet;       // проверенные вершины
  Gamma:= CreateSet;        // соседние вершины

  Extend(Cand, Tested); // рекурс. процедура построения независимых множеств
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
  Res.Free;
end;
```

17.2.5. Тестовый пример

Метод **GenUndepend** проверен на графе, входной файл которого показан ниже.

```
KRISTOFIDES Page 44
0 - граф(0), орграф(1)
0 - нагруженность вершин
0 - нагруженность рёбер (дуг)
8 - количество вершин
A B C D E F G H
A -> B F H
B -> A C F
C -> B D E
D -> C E G
E -> C D F
F -> A B E G H
G -> D F
H -> A F
```

Испытания дали следующие подмножества независимых вершин рис. 17-6):

```
A-C-G,  A-D,  A-E-G,  B-D-H,  B-E-G-H,  C-F,  C-G-H,  D-F
```

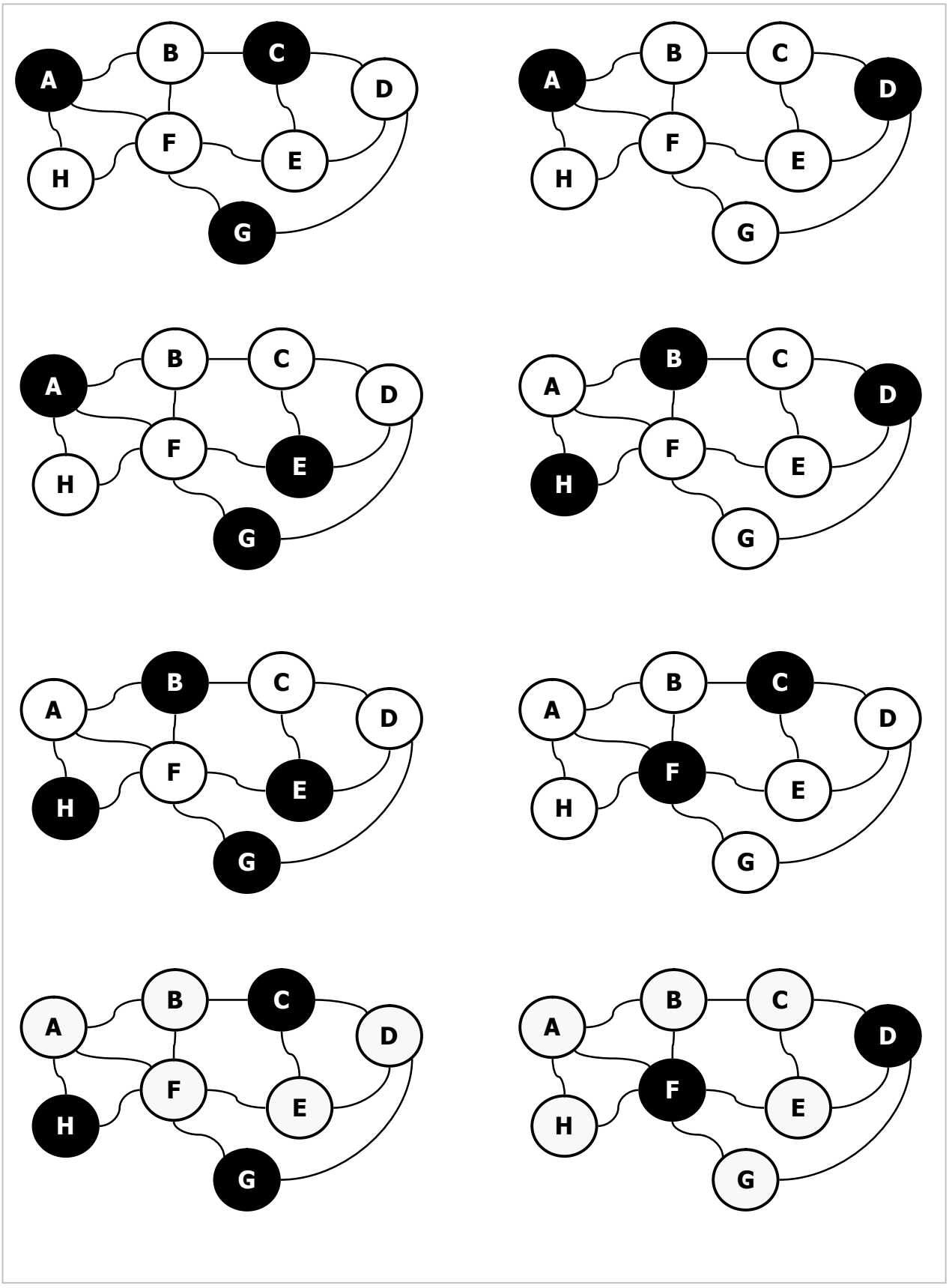


Рис. 17-6 — Независимые вершины графа

17.3. Алгоритм Брона-Кербоша для поиска максимальных клик

Назначение алгоритма — сформировать множество максимальных клик.

17.3.1. Основная идея

Здесь пригоден слегка модифицированный алгоритм поиска независимых множеств, в котором некоторые операции заменены зеркально противоположными. В табл. 17-2 показаны изменения, которые преобразуют один алгоритм в другой.

Табл. 17-2 — Отличия алгоритмов Брона-Кербоша

Процедура	Действия	
	Для независимых вершин	Для клик
Формирование новых кандидатов newCand	Из кандидатов удаляются соседние вершины	В кандидатах остаются только соседние вершины
Формирование новых проверенных newTested	Удаляются соседние вершины	Остаются только соседние вершины
Тестирование обработанных вершин на входе в процедуру расширения (функция Test)	Функция Test проверяет наличие хотя бы одной вершины, не соседствующей ни с одним кандидатом	Функция Test проверяет наличие хотя бы одной вершины, соседствующей со всеми кандидатами

Листинг 17-2 — Формирование множества максимальных клик

```
//      Формирование множества максимальных  клик
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Неориентированый граф

function TGraph.GenClique: TSet;
var
  Res: TSet;      // очередная клика
  Gamma : TSet;   // соседние вершины
  //-------------------------------------------------------------
  // Возвращает TRUE, если среди проверенных вершин в aTested
  // есть хоть одна, примыкающая ко всем кандидатами в aCand

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи Node
      Gamma.Mul(aCand);        // Gamma = соседи * кандидаты
      if Gamma.GetCount = aCand.GetCount then begin
        // если примыкает ко всем кандидатам
        Result:= true;
        Break;
      end;
      Node:= TNode(aTested.GetNext);
    end;
  end;
  //-------------------------------------------------------------
  // Рекурсивная процедура расширения клики

  procedure Extend(aCand, aTested : TSet);
```

```
  var
    newCand : TSet;  // вершины-кандидаты
    newTested: TSet; // проверенные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и в проверенных aTested нет таких,
    // что примыкают ко всем кандидатами в aCand
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst); // очередной кандидат
      Res.Insert(Node);             // вставляем в клику
      Node.OutGammaGet(Gamma);      // Gamma:= соседи Node

      newCand:= TSet(aCand.Copy);   // копия кандидатов
      newCand.Delete(Node);         // удаляем текущую вершину
      newCand.Mul(Gamma);           // и оставляем только соседние

      newTested:= TSet(aTested.Copy); // копия проверенных
      newTested.Mul(Gamma);           // оставляем только соседние

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
             // то найдена очередная клика
             Result.Insert(Res.Copy)

        else if newCand.GetCount<>0 then
             // если остались кандидаты, то рекурсивный вызов
             // с новыми наборами кандидатов и проверенных
             Extend(newCand, newTested);
      // Возвращение
      newTested.Free;        // удалить копию проверенных
      newCand.Free;          // удалить копию кандидатов
      Res.Delete(Node);      // удалить текущий из результата
      aCand.Delete(Node);    // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end
  end;
  //-------------------------------------------------------------
var
  Cand : TSet;   // вершины-кандидаты
  Tested: TSet;  // проверенные вершины
begin
  // Инициализация
  Result:= CreateSet;       // создать пустое множество-результат
  Res:= CreateSet;          // очередная клика
  Cand := CreateSet;        // кандидаты
  Cand.CopyItems(mNodes);   // Cand = все вершины графа
  Tested:= CreateSet;       // проверенные вершины
  Gamma:= CreateSet;        // соседние вершины

  Extend(Cand, Tested);  // рекурсивная процедура
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
  Res.Free;
end;
```

17.3.2. Контрольный пример

Метод **GenClique** проверен на графе, входной файл которого показан ниже.

```
KRISTOFIDES Page 44
0 - граф(0), орграф(1)
0 - нагруженность вершин
0 - нагруженность рёбер (дуг)
8 - количество вершин
A B C D E F G H
A -> B F H
B -> A C F
C -> B D E
D -> C E G
E -> C D F
F -> A B E G H
G -> D F
H -> A F
```

Получены следующие подмножества максимальных клик (см. рис. 17-7):

```
A-B-F,  A-F-H,  B-C,  C-D-E,  D-G,  E-F,  F-G
```

17.4. О трудоёмкости

Интуиция подсказывает (а опыт подтверждает), что в разреженном графе клики должны формироваться быстро, поскольку соседей у вершин немного, и круг кандидатов в клику быстро редеет. Наоборот, поиск независимых множеств в таком графе будет затруднён, и тут сильно выручает отсечение тупиковых ветвей функцией **`Test`**, что убыстряет перебор на много порядков. В плотном графе всё наоборот: клики ищутся долго, а независимые вершины быстро.

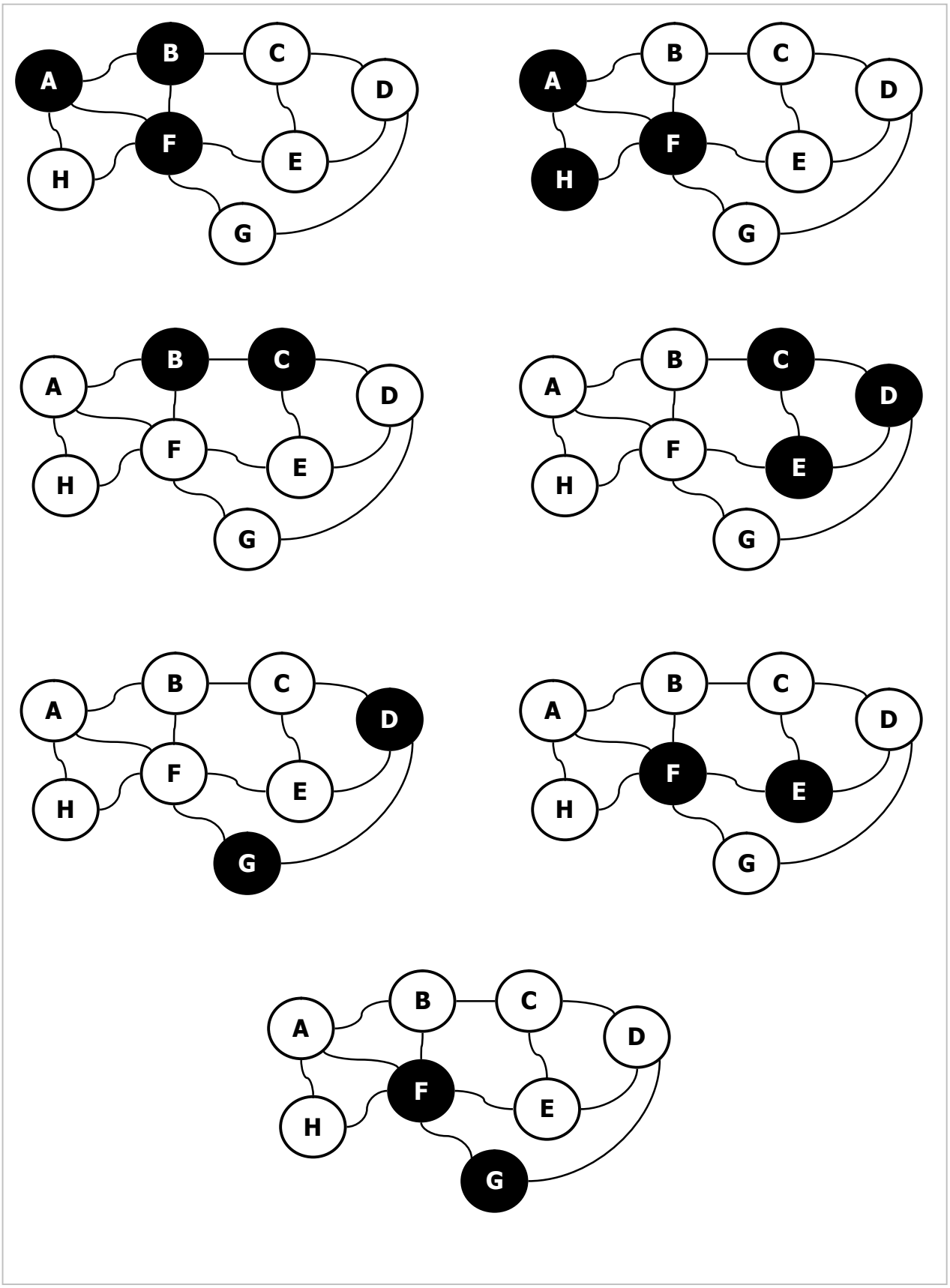


Рис. 17-7 — Клики графа

17.5. Итоги

- Клики и независимые вершины — это два взаимно противоположных, но тесно связанных понятия.
- Клики — это подмножества вершин, где каждая вершина связана с каждой в этом же подмножестве.
- Независимые множества — это такие подмножества вершин, где никакая вершина не связана с другой из этого же подмножества.
- Клики обратного графа совпадают с независимыми множествами исходного графа, и наоборот.
- Для поиска клик и независимых вершин используют две модификации алгоритма Брона-Кербоша.

17.6. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 43
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 18
Минимальные доминирующие множества

В качестве примера использования *множества независимых вершин* графа, в предыдущей главе приведено оптимальное размещение лечебниц. Рассмотрим более общую задачу такого рода: поиск оптимального размещения экстренных служб на любых графах, в том числе с взвешенными вершинами и дугами.

18.1. Постановка задачи

Рассмотрим архипелаг, где острова соединены мостами с односторонним движением так, как показано на рис. 18-1.

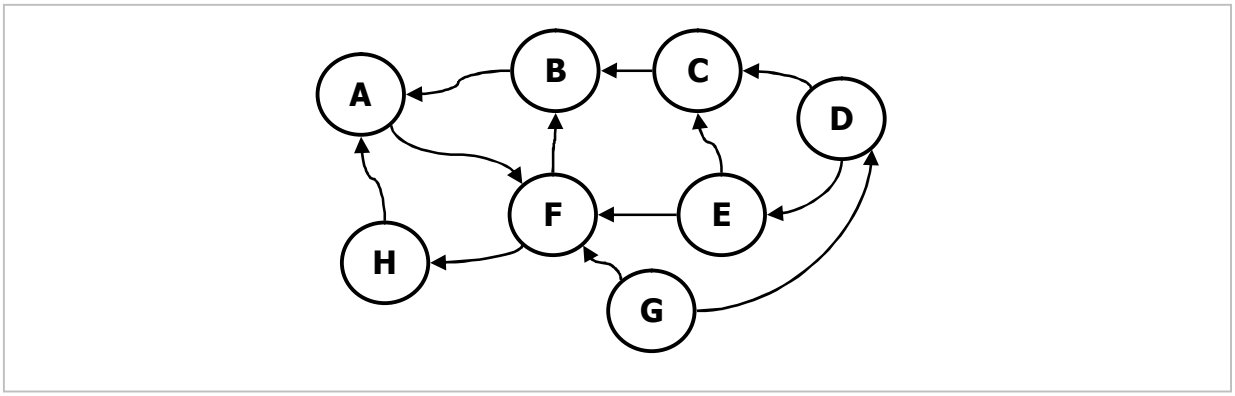


Рис. 18-1 — Островной архипелаг с односторонними мостами

Будем искать острова для размещения пожарных команд с тем условием, чтобы от места базирования команды до места пожара их автомобили преодолевали не более одного моста. То есть, команды должны тушить свой остров и ближайших соседей. Обратите внимание, что возвращаться пожарные могут окружными путями, либо морем — обратный маршрут нас не интересует.

У этой задачи есть вариации. Например, когда стоимость постройки пожарной части и её содержание на разных островах различна, и тогда ищут вариант минимальной стоимости проекта (на графе с взвешенными вершинами). В другом варианте учитывается длина мостов (веса дуг), и тогда минимизируют время их преодоления. Возможна и комбинация этих вариантов. Подмножество искомых здесь вершин называют *минимальным доминирующим множеством*.

18.2. Доминирующие множества

В графе может быть несколько доминирующих множеств, все они совпадают в том, что прочие вершины достигается из этого множества не более чем за шаг. На практике интересно одно из этих доминирующих множеств, имеющее либо минимальную мощность (содержащее наименьшее количество вершин), либо минимальную стоимость (**`Cost`**) с учётом весов вершин и/или дуг.

Результат предстоящего поиска будет зависеть от того, ориентирован ли граф, взвешены ли его вершины и дуги, однако все модификации задачи решаются одной общей процедурой.

18.3. Принцип решения

Рассмотрим следующую таблицу.

Табл. 18-1 — Подмножества покрывающих граф вершин

Корневая вершина (откуда)	Подмножества достигаемых вершин (куда)							
A	A	-	-	-	-	F	-	-
B	A	B	-	-	-	-	-	-
C	-	B	C	-	-	-	-	-
D	-	-	C	D	E	-	-	-
E	-	-	-	-	E	F	-	-
F	-	-	-	-	-	F	-	H
G	-	-	-	D	-	F	G	-
H	A	-	-	-	-	-	-	H
Сумма подмножеств	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*

Табл. 18-1 построена путём перебора вершин графа следующим образом. Взяв очередную вершину (назовём её корневой), поместим её в левую колонку, а в ячейках справа отметим вершины, доступные из корневой не более чем за шаг (сюда попадает и корневая). В итоге справа оказались подмножества достигаемых вершин, в сумме покрывающие все вершины графа. Остаётся собрать из этих подмножеств достигаемых вершин минимальное покрытие, — это стандартная задача на множествах (ЗНП), она решена в главе 11.

18.4. Листинг

Поиск одного из минимальных доминирующих множеств выполняется представленным ниже методом **`GenDominating`**. Своей простотой он обязан функции **`CollectMinCover`** из модуля **`Assembly`**. Остаётся лишь сформировать исходные данные для функции — промежуточный буфер, в который помещаются оценённые подмножества достигаемых вершин (табл. 18-1).

Листинг 18-1 — Формирование доминирующего множества (Граф + орграф)

```
function TGraph.GenDominating: TCostSet;
var  Buf: TBuffer;    // промежуточный буфер
     Node: TNode;     // Вершина графа
     Gamma: TSet;     // для множества соседей
     CS: TCostNodes; // оценённое подмножество
begin
  Buf:= TBuffer.Create;
  // В промежуточный буфер заносим оценённые подмножества,
  // каждое из которых соответствует одной вершине
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Gamma:= CreateSet;         // создаём пустое множество
    Node.OutGammaGet(Gamma);   // Gamma= соседние вершины
    Gamma.Insert(Node);        // + текущая
    // Node.CalcCost вычисляет цену узла как сумму веса узла и веса дуг
    CS:= TCostNodes.Create(Node.CalcCost, Gamma, Node);
    Buf.Push(CS);          // заносим в буфер
    Node:= NodeNext;
  end;
  // Поиск минимального покрытия
  Result:= Assembly.CollectMinCover(mNodes, Buf);
  // Удаляем результат из буфера
  SetUtils.RemoveItemsFromBuf(Buf, Result.mSet);
  // а буфер очищаем и уничтожаем
  Buf.ClrAndDestroy;
  Buf.Free;
end;
```

В промежуточный буфер заносятся элементы класса **TCostNodes** — это потомок класса **TCostSet** с добавленным полем — корневой вершиной, здесь также переопределены конструктор и метод вывода в текстовый файл.

```
// Элемент для представления доминирующих множеств
type
  TCostNodes = class (TCostSet)
    mNode: TNode;  // вершина-элемент доминирующего множества
    constructor Create(aCost: integer; aSet: TSet; aNode: TNode);
    procedure   Print(var aFile: TextFile); override;
  end;

constructor TCostNodes.Create(aCost: integer; aSet: TSet; aNode: TNode);
begin
  inherited Create(aCost, aSet, true);
  mNode:= aNode;
end;

procedure TCostNodes.Print(var aFile: TextFile);
var Node: TNode;
begin
  Write(aFile, mNode.GetName, ' = ', mCost, ' -> { ');
  Node:= TNode(mSet.GetFirst);
  while Assigned(Node) do begin
    Write(aFile, Node.GetName + ' ');
    Node:= TNode(mSet.GetNext);
  end;
  Writeln(aFile, '}');
end;
```

«Ценовую политику» при создании подмножеств определяет метод **Node.CalcCost**, вычисляющий цену соответствующего подмножества на основе полей **mValue** в объектах-вершинах и объектах-связях. Если же вершины и дуги графа не взвешены, то каждому подмножеству назначается единичная цена, и тогда алгоритм ищет подмножество вершин минимальной мощности.

```
// Подсчёт стоимости узла для поиска минимальных покрытий

function TNode.CalcCost: integer;
var L: TLink;
begin
  Result:=1;
  if mLoadNodes then Result:= mValue;
  if not mLoadLinks then Exit;
  // добавляем суммарную стоимость исходящих дуг (рёбер)
  L:= OutLinkFirst;
  while Assigned(L) do begin
    Inc(Result, L.mValue);
    L:= OutLinkNext;
  end;
end;
```

Наконец, перед выходом из метода **GenDominating** очищается и удаляется промежуточный буфер, но так, чтобы не потерять подмножества, попавшие в результат. Для этого вызывается процедура **RemoveItemsFromBuf** из модуля **SetUtils**.

18.5. Контрольные примеры

Ниже представлены результаты испытаний метода на ориентированном графе (**рис. 17-1**) и на том же графе с взвешенными вершинами (рис. 18-3). Во втором примере вес вершины *F* задан равным 9, в результате чего она потеряла статус доминирующей, передав «полномочия» двум соседним.

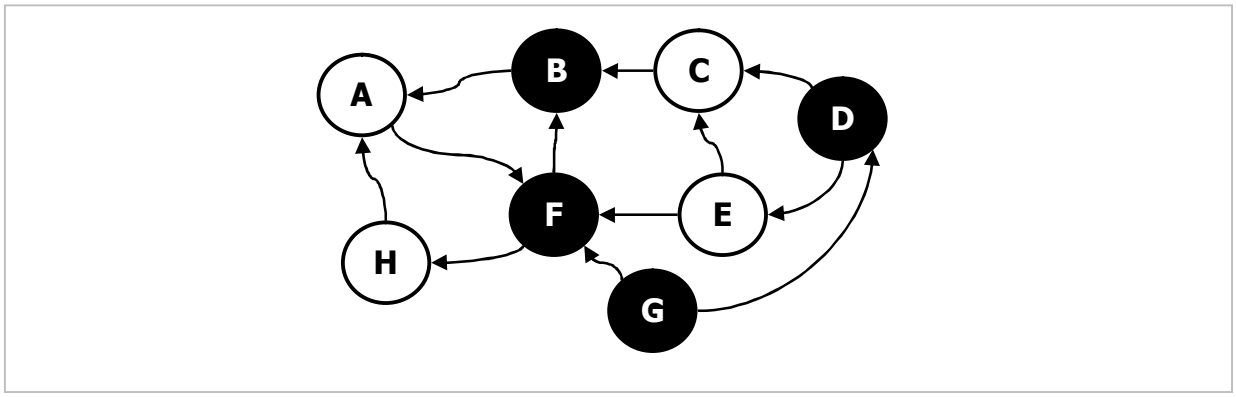


Рис. 18-2— Доминирующее множество не взвешенного графа

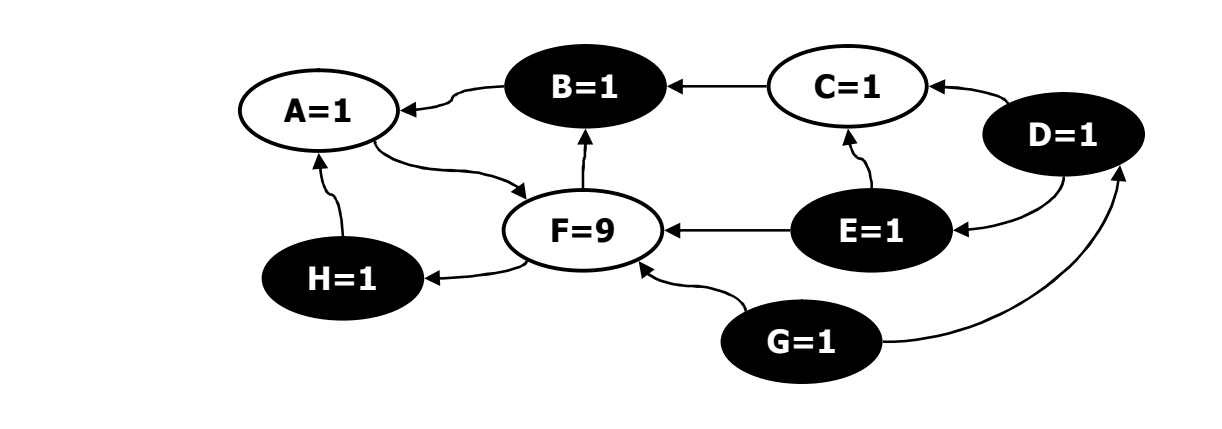


Рис. 18-3 — Доминирующее множество графа с взвешенными вершинами

18.6. Итоги

- Путь из доминирующего множества вершин в любую другую пролегает не более чем через одну дугу (ребро).
- На практике интерес представляют минимальные доминирующие множества, они отыскиваются через решение задачи о минимальном покрытии (ЗНП).
- Минимальные доминирующие множества могут формироваться либо с учётом, либо без учёта весов вершин и дуг.

18.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 43
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 19
Раскраски

19.1. Постановка задачи

Обратимся к «детской забаве», и раскрасим вершины неориентированного графа минимальным набором красок так, чтобы ни одна вершина не соседствовала с вершиной того же цвета. Пример такой раскраски цвета показан на рис. 19-1.

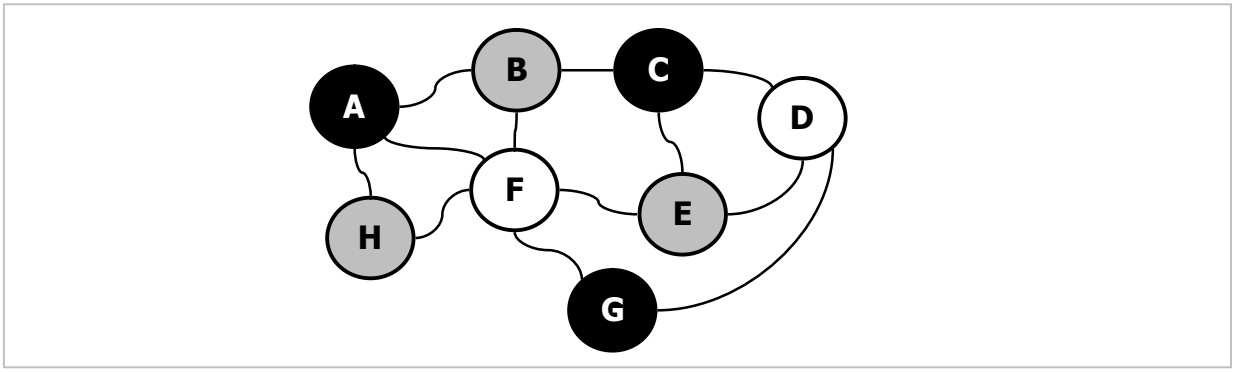


Рис. 19-1 — Трёхцветная раскраска графа

Суть, разумеется, не в красоте, а в практической необходимости. Пусть требуется перевезти по тряской дороге расфасованные вещества, способные вступать между собой в нежелательные химические реакции. Тогда следует уложить их в контейнеры так, чтобы при нарушении упаковки, лежащие вместе вещества не вступили в реакцию. При этом количество контейнеров должно быть минимально.

Сведём это к задаче раскраски. Представим каждое вещество вершиной графа и соединим рёбрами взаимно реагирующие вещества. Тогда соответствующие инцидентным вершинам вещества нельзя класть в один контейнер. Если все контейнеры окрашены в разные цвета, то можно вообразить окрашенными в те же цвета и помещённые в них упаковки с веществами.

В задачах раскраски преследуют две цели: а) определить минимально необходимое количество контейнеров (красок) и б) найти распределение упаковок по контейнерам, то есть раскраску вершин конкретного графа.

19.2. Точное решение

Начнём с множества независимых вершин (глава 17). В каждом таком множестве (рис. 19-2) все вершины попарно не инцидентны, и могут быть окрашены одним цветом — уложены в один контейнер. Но как быть с оставшимися вершинами?

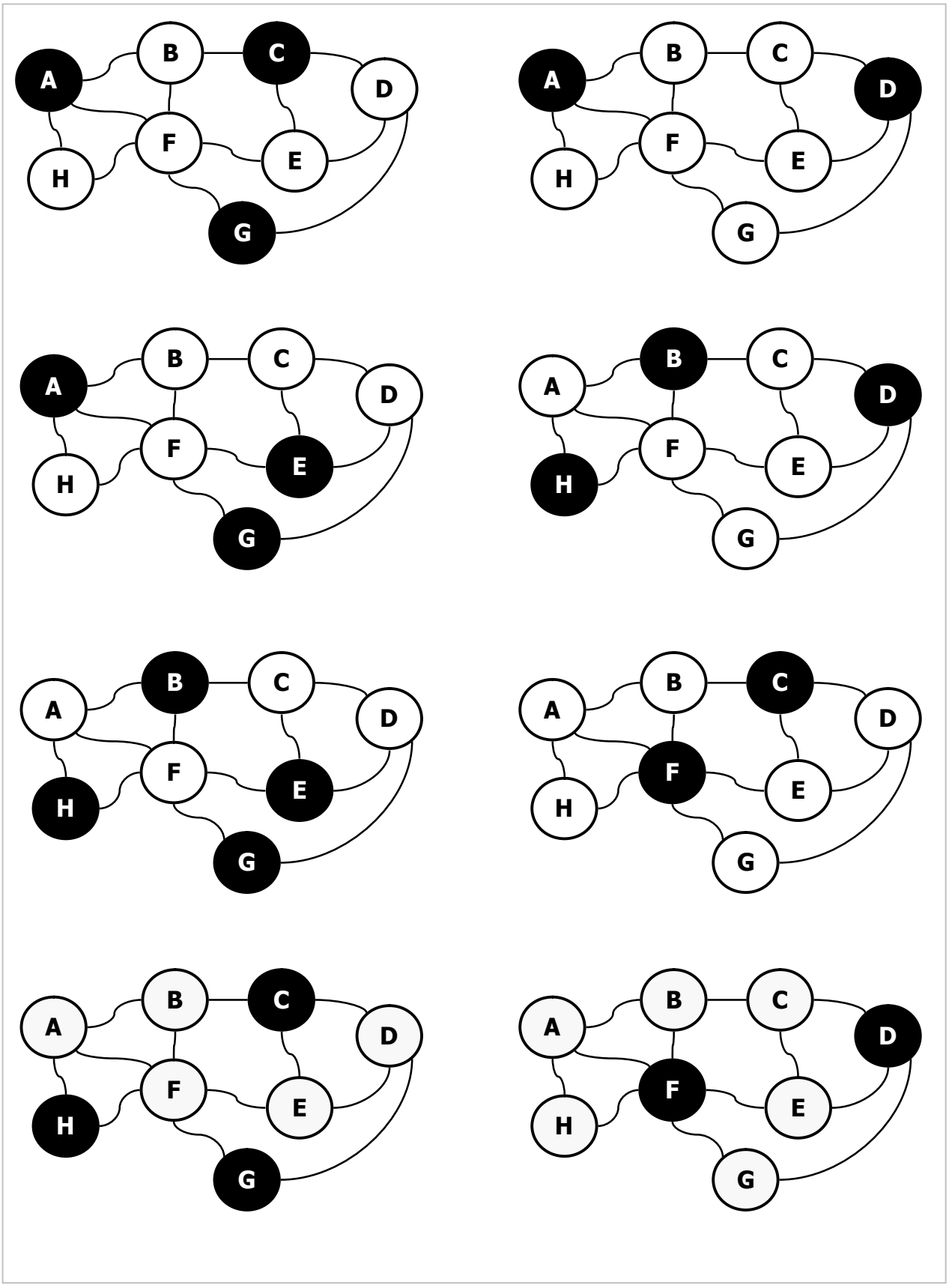


Рис. 19-2 — Все подмножества независимых вершин графа

Начнём с менее сложной задачи: определим минимальное количество требуемых красок — *хроматическое число* графа (то есть, минимальное число контейнеров). Для этого найдём и поместим в табл. 19-1 все максимальные независимые множества вершин графа.

Табл. 19-1 — Все независимые множества вершин

Условный цвет	Независимые подмножества вершин							
1	A	-	C	-	-	-	G	-
2	A	-	-	D	-	-	-	-
3	A	-	-	-	E	-	G	-
4	-	B	-	D	-	-	-	H
5	-	B	-	-	E	-	G	H
6	-	-	C	-	-	F	-	-
7	-	-	C	-	-	-	G	H
8	-	-	-	D	-	F	-	-
Сумма подмножеств	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*

Каждое из этих подмножеств можно без опаски окрасить своим цветом (цвета условно пронумерованы в левой колонке). Если найти минимальный набор подмножеств, полностью покрывающий граф, то их количество и будет хроматическим числом. В данном примере для покрытия годны три подмножества с «цветами» **1**, **5** и **8**: **A–C–G**, **B–E–G–H** и **D–F**, они выделены серым. Значит, хроматическое число данного графа составляет 3. Такой минимальный набор подмножеств ищется как минимальное покрытие (ЗНП), — этот же приёмом использовался для поиска доминирующих вершин в главе 18.

Отсюда рукой подать до оптимальной раскраски графа (в данном примере — тремя красками). Если бы найденные подмножества составили *разбиение*, то стали бы решением задачи. Напомню, что в разбиении подмножества взаимно *не пересекаются*. Однако здесь вершина *G* входит в два подмножества и претендует на запрещённую двойную окраску. Если оставить её только в одном из подмножеств, то проблема исчезнет, — далее мы так и поступим, превратив *покрытие* в *разбиение*.

19.3. Приближённое решение

Итак, путь к точному решению задачи о раскраске лежит через решение двух промежуточных задач:

- формирование всех подмножеств независимых вершин:
- решение задачи о наименьшем покрытии (ЗНП).

Вторая задача, как известно, NP-сложна, и даже для умеренного объёма данных может затребовать неприемлемое время. Рассмотрим несколько быстрых и «почти оптимальных» алгоритмов раскраски. «Почти» потому, что количество

истраченных при этом красок иногда может превышать хроматическое число. Один из таких приёмов — быстрое градиентное решение задачи о наименьшем покрытии, рассмотренное в главе 11. Как правило, оно даёт результат не более чем на 25% хуже оптимального.

19.4. Листинг

Алгоритм поиска одной из оптимальных раскрасок реализован в методе **TGraph.GenPaints**. Он генерирует множество подмножеств вершин, каждое из которых может быть окрашено своим цветом. Отмечу, что поле **TNode.mColor** здесь не затрагивается, и никакой реальной окраски вершин не происходит.

Листинг 19-1 — Генерация оптимальной или субоптимальной раскраски

```
function TGraph.GenPaints(aQuick: boolean): TSet;
var  Undep : TSet;        // множество независимых множеств
       Buf : TBuffer;   // рабочий буфер
         S : TSet;        // текущее подмножество вершин
        CS : TCostSet;    // оценённое подмножество вершин
     Cover : TCostSet;    // минимальное покрытие
begin
  Undep:= GenUndepend;    // генерируем все независимые подмножества вершин
  // Переносим независимые множества в буфер оценённых подмножеств
  // для последующего поиска минимального покрытия
  Buf:= TBuffer.Create;       // создаём рабочий буфер
  S:= Undep.GetFirst as TSet;  // первое независимое множество
  while Assigned(S) do begin   // перебор независимых множеств
    // Создаём оценённые подмножества (цена Cost=1)
    // и помещаем в буфер:
    CS:= TCostSet.Create(1, S.Copy as TSet, true);
    Buf.Push(CS);
    S:= Undep.GetNext as TSet;
  end;
  // Уничтожаем множество максимальных независимых множеств
  Undep.ClrAndDestroy;
  Undep.Free;
  // Формируем минимальное покрытие вершин (unit Assembly):
  if aQuick
    then Cover:= CollectGradCover(mNodes, Buf)  // быстро
    else Cover:= CollectMinCover(mNodes, Buf);  // точно
  RemoveItemsFromBuf(Buf, Cover.mSet);  // удаляем результат из буфера
  Buf.ClrAndDestroy;                    // а буфер очищаем
  Buf.Free;                             // и удаляем
  // Формируем результат:
  Result:= Cover.mSet.Copy as TSet;  // копия множества-покрытия
  Cover.Free;                        // удаляем покрытие
  Result.CoverToDissect;             // превращаем покрытие в разбиение
end;
```

Метод принимает булев параметр, влияющий на ускорение при поиске наименьшего покрытия: **false** — требуется точное решение; **true** — необходимо быстрое решение, но количество красок может превысить хроматическое число.

После генерации в переменной **Undep** множества всех максимальных независимых вершин, формируется буфер оценённых подмножеств **Buf**. Далее он «скармливается» функциям **CollectMinCover** или **CollectGradCover**. Найденные через них покрытия копируются в результат, удаляются из временного буфера, а буфер уничтожается. Напоследок вызывается специфический метод **TSet.CoverToDissect**, превращающий покрытие в одно из разбиений (листинг 19-2).

Листинг 19-2 — Преобразование наименьшего покрытия в разбиение

```
// Преобразование покрытия в разбиение
// Элементами множества должны быть TCostSet!

procedure TSet.CoverToDissect;
var   i, j : integer;  // индексы
    Si, Sj : TCostSet; // текущие элементы
begin
  for i:= 1 to GetCount-1 do begin
    Si:= GetItem(i) as TCostSet;
    // Вычитаем из текущего подмножества все последующие
    for j:= i+1 to GetCount do begin
      Sj:= GetItem(j) as TCostSet;
      Si.mSet.Sub(Sj.mSet);          // Si:= Si - Sj
    end;
  end;
end;
```

19.5. Испытание

Ниже показана программа для испытания раскрасок графа:

Листинг 19-3 — Программа для испытания раскрасок

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr: TGraphChars;
    Paints: TSet;
    S : string;

begin
  repeat
    Gr:= TGraphChars.GenRandom(false, 1, 1, 20, 50);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - - -');
    Paints:= Gr.GenPaints(true);  // быстро
    Paints.Expo;
    Paints.ClrAndDestroy;
    Paints.Free;
```

```
    Writeln('- - - - - - - -');
    Paints:= Gr.GenPaints(false); // точно
    Paints.Expo;
    Paints.ClrAndDestroy;
    Paints.Free;
    Writeln('- - - - - - - - - - - - - - - - -');
    Write('Quit ? ');
    Readln(S);
    Gr.Free; // Освобождение графа
  until S<>'';
end.
```

19.6. Итоги

- Суть задачи раскраски: окрасить вершины графа минимальным набором красок так, чтобы вершины в любых смежных парах оказались разноцветными.
- Минимальное количество красок, необходимое для раскраски, называется хроматическим числом графа.
- В ходе поиска оптимальной раскраски решаются две промежуточные задачи: a) формирование всех независимых подмножеств вершин и b) поиск наименьшего покрытия с последующим преобразованием его в разбиение.
- Точное решение задачи о раскраске экспоненциально сложно, поэтому иногда прибегают к менее точным, но быстрым методам.

19.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 75
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

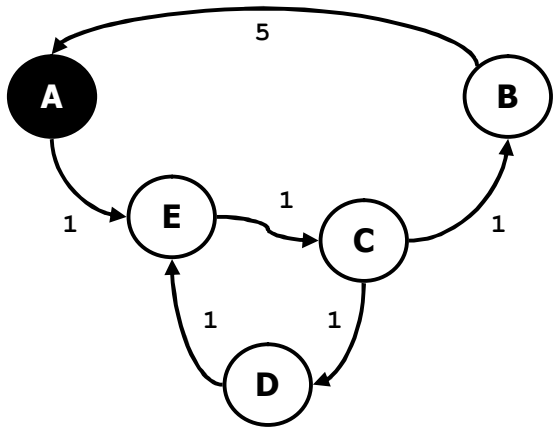
Глава 20
Центры графа

Продолжим тему оптимального размещения служб и рассмотрим в этой связи несколько примеров.

20.1. Центры: внешние, внутренние и внешне-внутренние

Пусть мосты с островами образуют либо связанный граф, либо сильно связанный орграф. Напомню, что в таких графах все вершины взаимно достижимы, хотя в орграфе прямой и обратный пути между вершинами могут отличаться. Поищем место для базирования единственной пожарной части. Разумно будет найти остров, путь от которого до самого удалённого от него острова был бы минимален. Задачи такого рода называют *минимаксными*. Из этих же соображений выбирают место размещения скорой помощи по вызову. Вершину, или подмножество вершин, отвечающих этому требованию, называют *внешним центром* графа. Путь из центра к самой удалённой от него вершине называют внешним радиусом ρ.

Для графа на рис. 20-1 внешним центром является вершина *A*, путь из которой к любой другой вершине (внешний радиус ρ) не превышает 3-х единиц.



Вершина *A* — внешний центр

Внешний радиус $\rho = 3$

Рис. 20-1 — Внешний центр орграфа

Теперь подыщем место для медпункта, к которому пациент добирается сам. Здесь надо минимизировать расстояние от самого удалённого острова к центру, который называют *внутренним*. Для графа на рис. 20-2 внутренним центром является вершина *B*, расстояние к которой из любой другой вершины (внутренний радиус ρ) не превышает 3-х единиц.

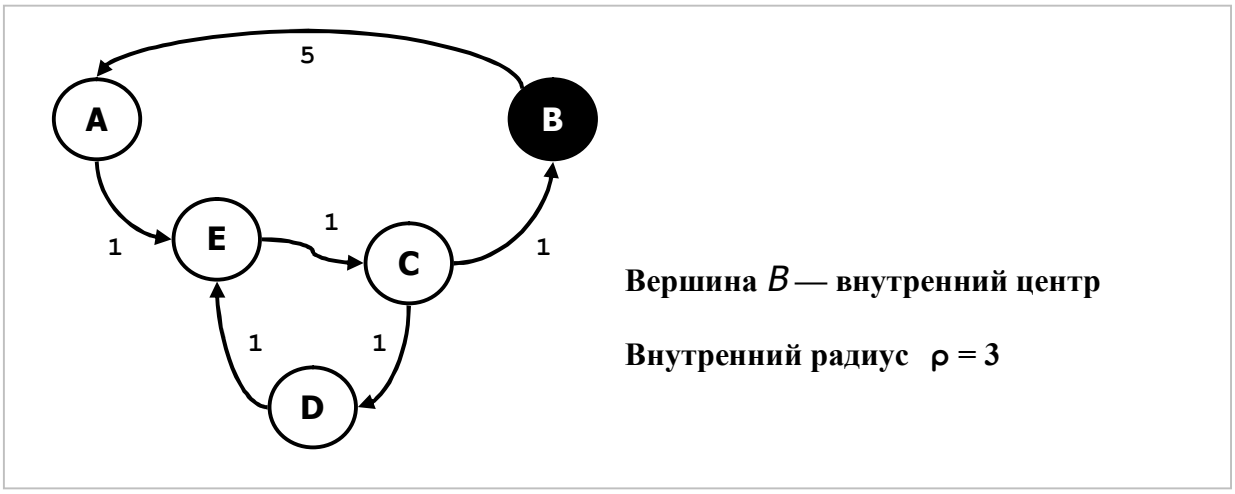


Рис. 20-2 — Внутренний центр орграфа

Наконец, подумаем о размещении больницы вкупе со скорой помощью. Здесь карета должна сначала доехать от больницы до пациента, а затем вновь доставить его в больницу. Тут следует минимизировать сумму расстояний от больницы до пациента и обратно, такой центр называют *внешне-внутренним*.

Понятно, что в *неориентированном* графе все три упомянутых центра совпадают. Центры неориентированного графа, показанного на рис. 20-3, представлены множеством вершин *C-D-E*. Здесь расстояние между любой из них и другими вершинами графа (радиус ρ) не превышает 2-х единиц. В данном случае экстренную службу можно разместить на любом из трёх островов.

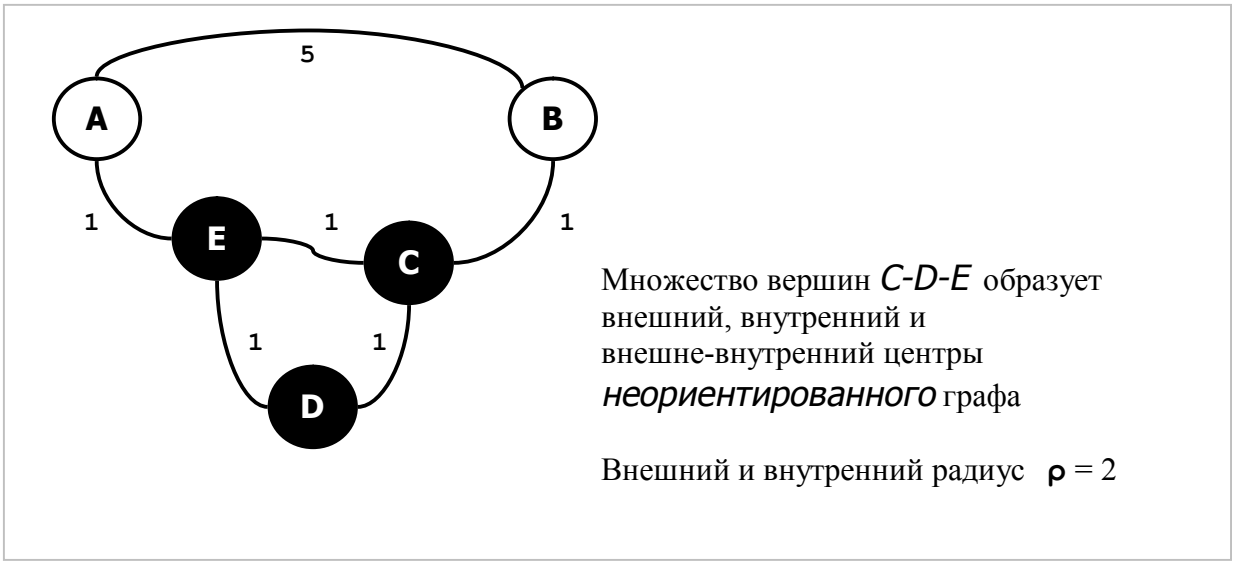


Рис. 20-3 — Центр неориентированного графа

Далее будут построены алгоритмы для поиска *внутренних*, *внешних* и *внешне-внутренних* центров графов и орграфов.

20.2. Взвешенное расстояние

В ходе этой работы введём расширенное понятие расстояния — *взвешенное расстояние*. Его определяют как произведение расстояния между вершинами на вес одной из этих вершин. Подобное существует в механике: это момент силы, определяемый как произведение силы **F** на плечо **L**. При равенстве моментов рычаг сохраняет равновесие (рис. 20-4).

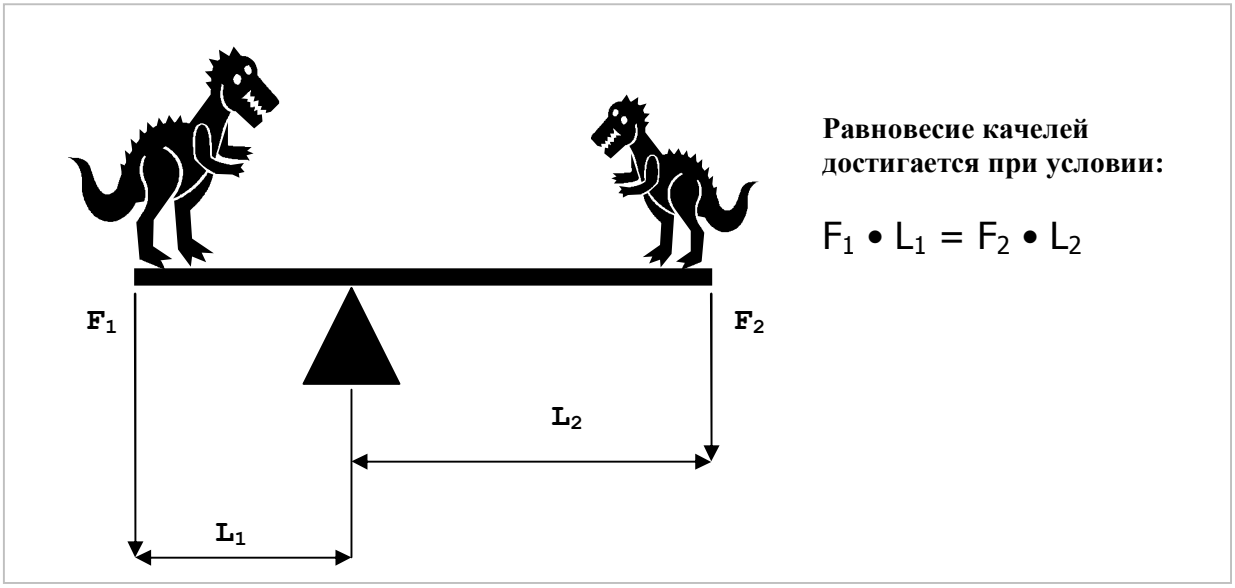


Рис. 20-4 — Пример равенства взвешенных расстояний (моментов)

Рассмотрим пример учёта взвешенного расстояния. Пусть почтовая связь между островами налажена через центр, куда стекается вся почта. Здесь она сортируется и вновь развозится по островам. Объём почты для каждого острова пропорционален его населённости (весу вершины графа). Положим, что почта доставляется самолётами, в которых часть полезного груза можно замещать дополнительным топливом, увеличивая тем дальность полёта. Стало быть, самолёт может доставить, либо большой груз на малое расстояние, либо малый груз — на большое. Здесь произведение дальности самолёта на его грузоподъёмность оценим константой **λ** — *взвешенной дальностью*. Какова должна быть эта константа **λ** для самолётов, и где поместить центр сортировки почты? Очевидно, что **λ** должна быть по возможности меньше, поскольку лёгкие самолёты дешевле. А почтовый центр следует расположить так, чтобы все острова оказались достижимы в пределах этой минимальной *взвешенной* дальности.

На рис. 20-5 почтовый центр размещен в вершине *C*. Наибольшие взвешенные расстояния пролегают от него к вершинам *D* и *E* и составляют **λ=2•4=8** условных единиц, — именно таким должно быть оптимальное произведение дальности на грузоподъёмность самолёта.

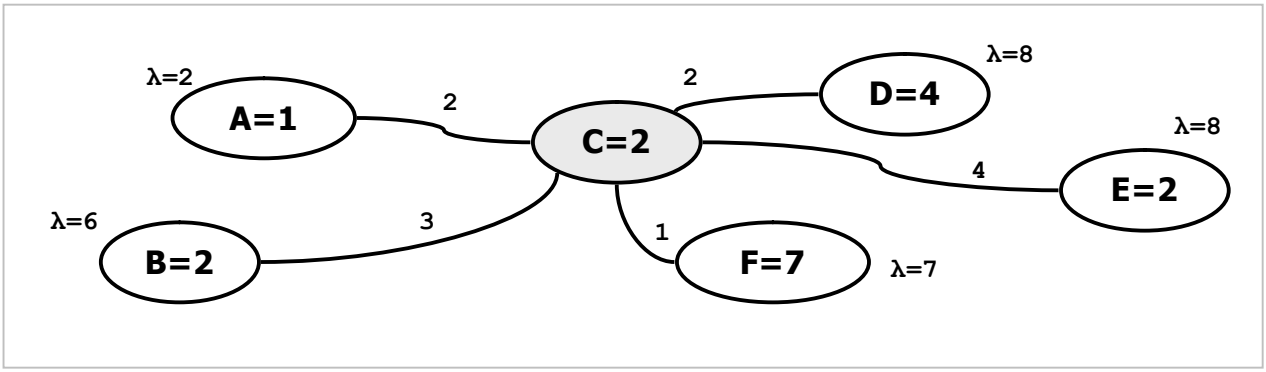


Рис. 20-5 — Центр сортировки почты в вершине ***C***

Отметим, что в отличие от обычного расстояния, взвешенное расстояние в общем случае *не симметрично* даже в неориентированном графе. Например, взвешенное расстояние $A \to C$ составляет **2•2=4** единицы, а обратное $C \to A$ равно **1•2=2** единицы.

В поисках центра будем оперировать взвешенными расстояниями. В графах, где вершины не взвешены, они совпадают с обычными расстояниями. Поэтому в целях наглядности последующие примеры приведены на графах, в которых веса всех вершин равны единице.

20.3. Разделения и радиусы

Рассмотрим орграф с единичными весами вершин (рис. 20-6). Возьмём вершину *D*, и найдём наиболее удалённую от неё по направлению стрелок. Такой вершиной является вершина *B*, расстояние до неё составляет **2+4+1+3+1=11** единиц. Если бы вес вершины *B* составлял, например, **5**, то взвешенное расстояние к ней составило бы **11•5=55**. Взвешенное расстояние к самой удалённой вершине по направлению стрелок называют *внешним числом разделения* для данной вершины. Итак, внешнее число разделения для вершины *D* составляет 11.

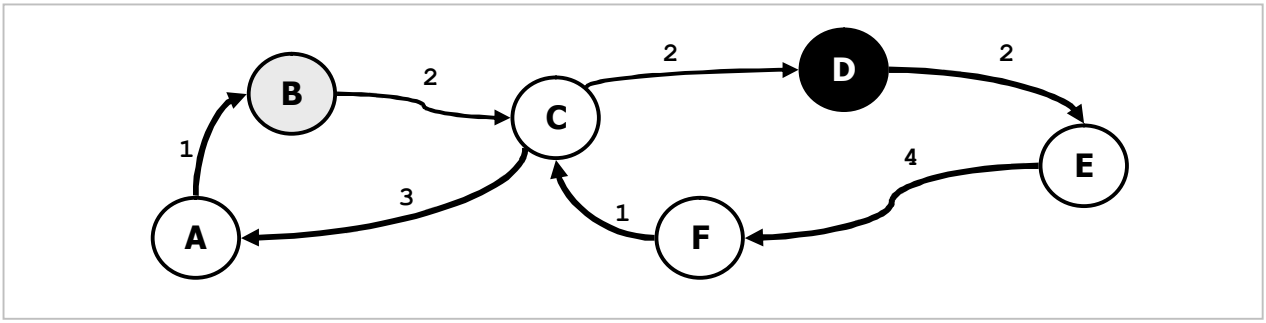


Рис. 20-6 — Внешнее число разделения для вершины ***D*** равно 11 (расстояние до ***B***)

Определив внешние числа разделения для всех вершин, можно выбрать вершину (или несколько вершин), число разделения которой будет минимально, —

она или они составят *внешний центр* графа. Минимальное внешнее число разделения называют *внешним радиусом*.

Теперь поступим наоборот, и найдём вершину, *из которой* путь к выбранной нами вершине *D* максимален, это будет вершина *E* (рис. 20-7). Соответствующее расстояние — *внутреннее число разделения* — для вершины *D* равно **2+1+4=7**. Заметим, что если бы вес вершины *E* составлял, к примеру, **5**, то внутреннее число разделения для вершины *D* составило бы **7•5=35**.

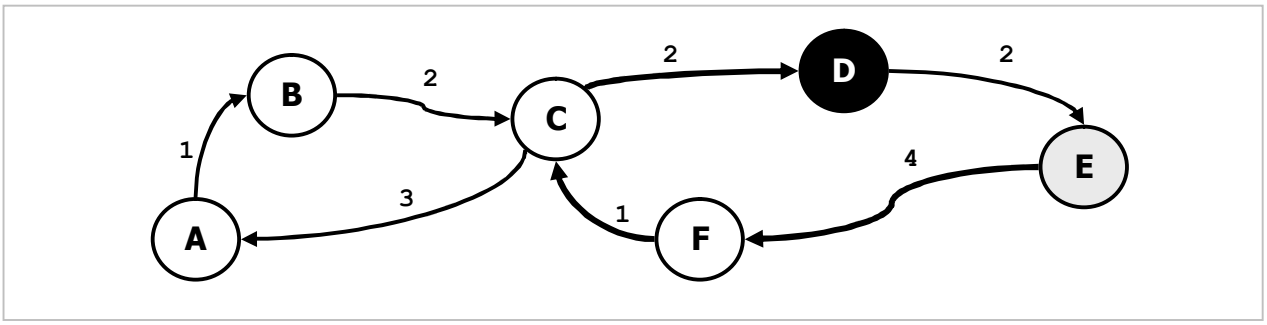


Рис. 20-7 — Внутреннее разделение для вершины ***D*** равно 7 (расстояние от ***E***)

Найдя внутренние числа разделения для всех вершин, можно отобрать одну или несколько вершин с минимальным внутренним числом разделения — *внутренний центр* графа. Соответствующее минимальное внутреннее число разделения будет *внутренним радиусом*.

Очевидно, что для неориентированного графа *внешние* и *внутренние* числа разделения, соответствующие им центры и радиусы совпадают.

20.4. Алгоритм поиска центров

Из приведенных выше рассуждений следует, что для поиска центра надо сначала найти обычные расстояния между всеми вершинами. По ним легко определить *взвешенные* расстояния от любой вершины ко всем остальным путём умножения обычных расстояний на вес целевых вершин. Останется лишь дважды перебрать вершины: сначала в поиске минимального числа разделения, а затем в розыске множества вершин, которое ему удовлетворяет.

Очевидно, что некоторую трудность доставляет лишь первая стадия — вычисление всех не взвешенных расстояний между вершинами графа. Однако схожая проблема уже решена в главе 15 методом Флойда-Уоршелла через составление карты дальних указателей. Здесь поступим схожим образом, и даже воспользуемся теми же элементами данных: дальними указателями **TFarLink**, хранящими в числе прочего расстояние к целевой вершине, а также полем **TNode.mFarLinks** — списком таких указателей. Однако модифицированная процедура составления карты указателей будет снабжена параметром, указывающим её назначение: поиск внутреннего, внешнего или внешне-внутреннего центра. Для неориентированного графа этот параметр может быть

любым и на результат не влияет. Тип параметра определён в модуле **Graph** как перечисление:

```
TCenter = (   Undefined,  // направление карты не определено
              InCenter,   // внешний центр и медиана
              OutCenter,  // внутренний центр и медиана
              InOutCenter // внешне-внутренний центр и медиана
            );
```

Итак, для поиска центров создадим основной и два вспомогательных метода:

- **TGraph.GenCenter** — основной метод поиска центра;
- **TGraph.InitMap** — вспомогательный метод построение карты;
- **TGraph.DoneMap** — вспомогательный метод уничтожения карты;

Начнём со вспомогательных методов, выполняющих основную работу.

Метод **InitMap** строит распределённую по вершинам графа карту дальних указателей:

Листинг 20-1 — Вспомогательный метод инициализации карты дальних указателей

```
procedure TGraph.InitMap(aDirect: TCenter);

  // aDirect = InCenter | OutCenter | InOutCenter — направление поиска

  //- - - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей узла
  procedure InitFarLinks(aNode: TNode; aDirect : TCenter);
  var     Node : TNode;     // текущая вершина
          Link : TLink;     // ближний указатель
            FL : TFarLink;  // дальний указатель
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Для поиска внутреннего центра орграфа или центра графа
    procedure LocalIn;
    begin
      with Node do begin
        Link:= OutLinkFirst;            // первая исходящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mDest = aNode then begin  // сравнение с конечной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;   // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= OutLinkNext;     // следующая исходящая связь
        end // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - - - - -
    // Для поиска внешнего центра орграфа
    procedure LocalOut;
    begin
      with Node do begin
        Link:= InLinkFirst;             // первая входящая связь
```

```
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mOwner = aNode then begin // сравнение с исходной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;   // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= InLinkNext;     // следующая исходящая связь
        end; // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - -
  begin   { InitFarLinks }
    PosPush;  // сохранить позицию перебора
    Node:= NodeFirst;                // перебор всех вершин графа
    while Assigned(Node) do begin    // цикл создания дальних связей
      if Node = aNode then begin
        FL:= TFarLink.Create(Node, Node, 0);   // это связь на себя
      end else with Node do begin
        // Создать связь на очередную вершину
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        if (aDirect=InCenter) or not mDirect
          then LocalIn   // Внутренний центр орграфа или центр графа
          else LocalOut; // Внешний центр орграфа
      end; // else
      aNode.mFarLinks.Insert(FL); // вставить в множество
      Node:= NodeNext;  // перебор всех вершин графа
    end;
    PosPop;  // восстановить позицию перебора
  end;
  //- - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
         FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin  { Handle }
    Result:= false;
    FL1:= aNode.mFarLinks.GetFirst as TFarLink;  // указатель из aNode
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
```

```
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)             // если определён
             and (mDist<>0)                  // и не сам на себя
             and (mNodeFar<>aNode)           // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear) // то проверяем расстояние
                   then Result:= true;       // признак того, что изменён
          FL2:= GetNext as TFarLink;         // следующая дальняя связь
        end; // while
        PositionPop;
      end;
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
  procedure Create(aDirect : TCenter);
  var  Node : TNode;     // текущая вершина
       Step : integer;   // этап обработки (номер цикла)
       Flag : boolean;   // признак продолжения обработки
  begin
    // Предварительная очистка карты:
    Node:= NodeFirst;
    while Assigned(Node) do begin          // перебор вершин
      with Node do if Assigned(mFarLinks) // если карта существует
        then mFarLinks.ClrAndDestroy       // очищаем
        else mFarLinks:= CreateSet;        // иначе создаём пустую
      InitFarLinks(Node, aDirect);         // инициализация дальних указателей
      Node:= NodeNext;
    end;

    // Обработка вершин

    for Step:= 1 to mNodes.GetCount-2  do begin
      Flag:= false;
      Node:= NodeFirst;
      while Assigned(Node) do begin // перебор вершин
        if Handle(Node, Step)       // если обновлялись дальние указатели,
          then Flag:= true;         // то отметить это установкой флага
        Node:= NodeNext;
      end;
      // если дальние указатели не обновлялись, то прервать цикл
      if not Flag then Break;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -

var  Node : TNode;     // текущая вершина
     Buf : TBuffer;    // буфер для хранения списков дальних указателей
  //- - - - - - - - - - - - - - - - - - - - -
  // Сохранение в буфере текущих расстояний
  procedure SaveInBuf;
  begin
    Node:= NodeFirst;
    // Сохранение списков в буфере:
    while Assigned(Node) do begin
      Buf.Put(Node.mFarLinks);
      Node.mFarLinks:= nil;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
```

```
  // Извлечение из буфера и добавление расстояний
  procedure AddFromBuf;
  var      S : TSet;        // список из буфера
         FL1 : TFarLink;    // дальний указатель из S
         FL2 : TFarLink;    // дальний указатель из Node
  begin
    Node:= NodeFirst;
    // Перебор вершин:
    while Assigned(Node) do begin
      S:= Buf.Get as TSet;
      FL1:= S.GetFirst as TFarLink;
      while Assigned(FL1) do begin
        FL2:= Node.GetFarLink(FL1.mNodeFar);
        if Assigned(FL2) then FL2.mDist:= FL2.mDist + FL1.mDist;
        FL1:= S.GetNext as TFarLink;
      end;
      S.ClrAndDestroy;
      S.Free;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
begin  { TGraph.InitMap }

  if mInitMap then Exit;  // Защита от повторной инициализации
  mInitMap:= true;        // Признак, что карта создана
  mFlagMap:= aDirect;     // запоминаем направление карты

  case aDirect of
    InCenter,   // внешний центр и медиана
    OutCenter:  // внутренний центр и медиана
      Create(aDirect);
    InOutCenter: // внешне-внутренний центр и медиана
      begin
        // Строим карту для входящих гамм:
        Create(InCenter);
        // Создаём буфер и сохраняем в нём расстояния между вершинами:
        Buf:= TBuffer.Create;
        SaveInBuf;
        // Строим карту для исходящих гамм:
        Create(OutCenter);
        // Прибавляем расстояния, сохранённые в буфере,
        // и освобождаем буфер:
        AddFromBuf;
        Buf.ClrAndDestroy;
        Buf.Free;
      end;
  end;
  // Сортируем списки дальних указателей по неубыванию расстояния
  // (необходимо при поиске медиан)
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.SortFarLinks;
    Node:= NodeNext;
  end;

end;
```

Метод включает ряд внутренних локальных процедур. Он создаёт распределённую по вершинам графа карту дальних указателей между парами вершин. Обратим внимание на построение внешне-внутренней карты. Здесь

последовательно строятся две карты: сначала внутренняя, затем внешняя, после чего прямые и обратные расстояния к вершинам суммируются. Для сохранения данных внутренней карты, перед построением внешней карты она переносится во временный буфер.

Метод **DoneMap** уничтожает ранее созданную карту:

```
// Освобождение карты дальних указателей

procedure TGraph.DoneMap;
var  Node : TNode;     // текущая вершина
begin
  // Если очищена, то выход:
  if not mInitMap then Exit;
  mInitMap:= false;         // признак очищенной карты
  mMapDirect:= Undefined;   // признак очищенной карты
  // Очистка распределённой по вершинам карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      if Assigned(mFarLinks) then begin
        mFarLinks.ClrAndDestroy;    // уничтожаем указатели
        mFarLinks.Free;             // и список
        mFarLinks:= nil;
      end;
    end;
    Node:= NodeNext;
  end;
end;
```

Пользуясь этой картой, основной метод **GenCenter** формирует центр графа (множество вершин), а также возвращает взвешенный радиус графа **aLambda**.

Листинг 20-2 — Функция формирования центра графа

```
// Возвращает множество вершин центра и взвешенный радиус aLambda
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenCenter(var aLambda: integer; aDirect: TCenter): TSet;

  //- - - - - - - - - - - - - - - - - - - - -
  // Поиск максимального взвешенного расстояния от вершины aNode.
  // Использует ранее построенную карту дальних указателей mFarLinks
  function GetMaxDist(aNode: TNode): integer;
  var FL: TFarLink;
      Value: integer;
  begin
    Result:= 0;
    with aNode.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if mLoadNodes   // если вершины графа нагружены...
          then Value:= FL.mNodeFar.mValue   // для нагруж. вершин
          else Value:= 1;                   // если вершины не нагружены
        if Result < FL.mDist * Value
          then Result:= FL.mDist * Value;
        FL:= GetNext as TFarLink;
      end;
```

```
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - -
var Dist, MinDist: integer;  // текущее и минимальное взвеш. расстояние
    Node: TNode;

begin  { TGraph.GenCenter }
  // Предварительно формируем карту дальних указателей,
  // в которой каждый элемент содержит расстояние между парами вершин.
  DoneMapAndSet;     // если карта создана, то ликвидируем её
  InitMap(aDirect);  // строим карту с заданным направлением

  // Выбор минимального из максимальных взвешенных расстояний
  MinDist:= MaxInt;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);
    if MinDist > Dist then MinDist:= Dist;
    Node:= NodeNext;
  end;
  Result:= CreateSet;
  // Формирование множества-центра
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);    // число разделения очередной вершины
    if Dist = MinDist then Result.Insert(Node);
    Node:= NodeNext;
  end;
  aLambda:= MinDist;  // минимальное из максимальных расстояний
end;
```

20.5. Испытание

Следующая программа отыскивает все центры графа.

Листинг 20-3 — Программа поиска центров

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr: TGraphChars;
    Center: TSet;
    Lambda : integer;

begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;

  Writeln('- - - - - - - OUT CENTER - - - - - - - - - - -');
  Center:= Gr.GenCenter(Lambda, OutCenter);
```

```
  Center.Expo;
  Writeln('Lambda= ', Lambda);
  Center.Free;

  Writeln('- - - - - - - IN CENTER - - - - - - - - - - -');
  Center:= Gr.GenCenter(Lambda, InCenter);
  Center.Expo;
  Writeln('Lambda= ', Lambda);
  Center.Free;
  Writeln('- - - - - - IN-OUT CENTER - - - - - - - - - -');
  Center:= Gr.GenCenter(Lambda, InOutCenter);
  Center.Expo;
  Writeln('Lambda= ', Lambda);
  Center.Free;
  Gr.Free; // Освобождение графа
  Readln;
end.
```

Испытание выполнено на показанном ниже взвешенном орграфе (см. также рис. 20-8):

```
Орграф с взвешенными дугами и вершинами
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

Результат представлен в таблице:

Центр	Вершины центра	Взвешенный радиус λ
Внешний	G	126
Внутренний	A	128
Внешне-внутренний	A, B, E, G, H	296

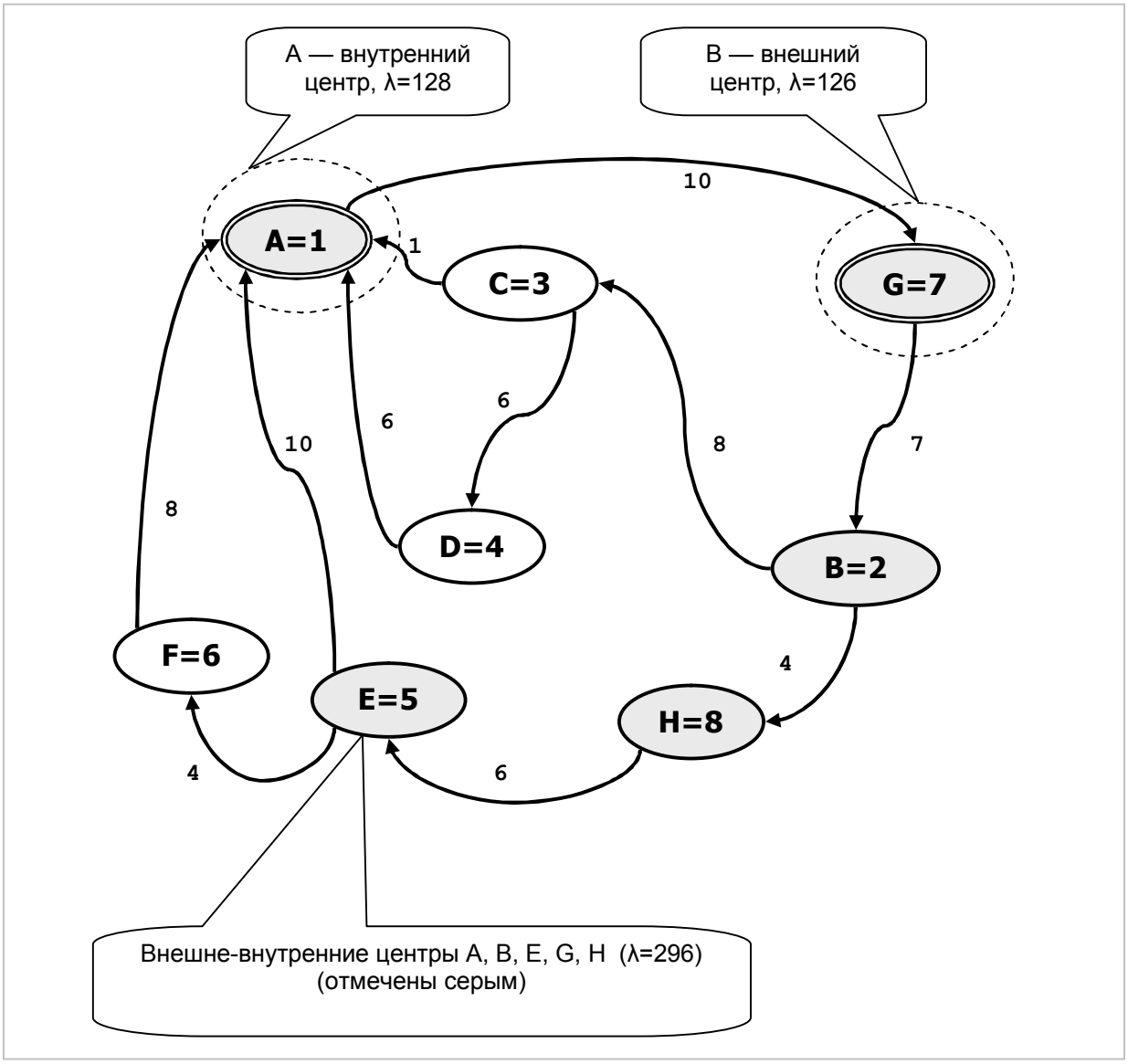


Рис. 20-8 — Центры орграфа с взвешенными дугами и вершинами

На роль внешне-внутреннего центра претендуют пять вершин, эти вершины образуют цикл *A->B->E->G->H->A*. Взвешенный радиус этого внешне-внутреннего центра равен длине цикла, умноженному на вес самой «тяжёлой» вершины в нём:

(10+7+4+6+10)•8 = 296.

Испытание на похожем неориентированном графе (рис. 20-9) дали иной результат.

```
Неориентированный граф с взвешенными дугами и вершинами
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

Центр	Вершины центра	Взвешенный радиус λ
Внешний	H	77
Внутренний	H	77
Внешне-внутренний	H	154

В неориентированном графе все три центра совпадают, а внешне-внутренний радиус, естественно, вдвое больше внутреннего и внешнего.

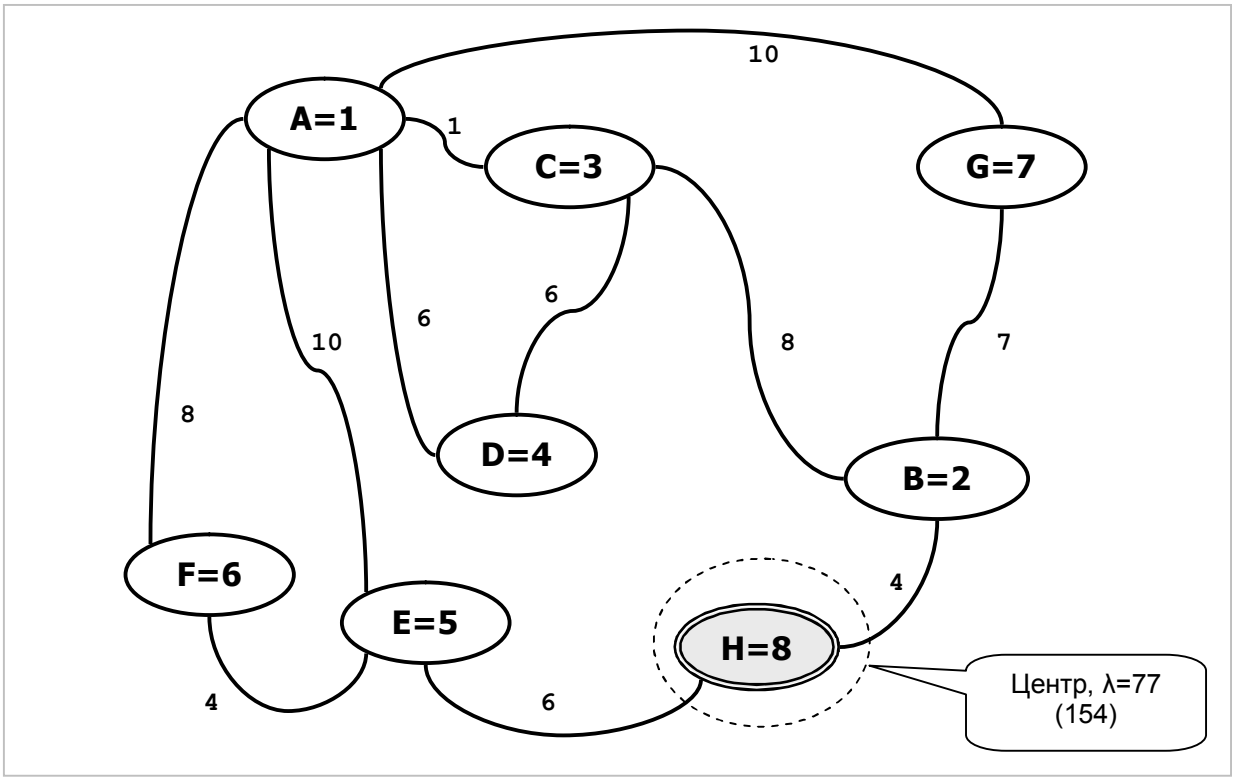


Рис. 20-9 — Центр неориентированного графа
с взвешенными дугами и вершинами

20.6. Итоги

- Задачи поиска внешних, внутренних и внешне-внутренних центров можно связать с размещением экстренных служб.
- При поиске центров в общем случае учитывается как длина дуг (рёбер), так и веса вершин, для чего вводится понятие взвешенного расстояния. Если вершины графа не взвешены или имеют одинаковый вес, то взвешенное расстояние становится эквивалентным обычному.
- Основная работа при поиске центров состоит в построении карты дальних указателей методом Флойда-Уоршелла.

20.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Глава 5, стр. 98
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Глава 8, стр. 265
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 21
P-центры

21.1. Что такое p-центры?

В предыдущей главе был найден *центр* графа — место для *единственной* экстренной службы (пожарной части, скорой помощи и т.п.). Службу размещают в вершине центра или в одной из вершин, если их несколько, так, чтобы взвешенное расстояние от неё до самой удалённой вершины оказалось минимальным.

Когда есть возможность разместить нескольких одинаковых служб (пожарных команд, больниц и т.п.), то собирать их в одном месте явно неразумно. Очевидно, что равномерное распределение служб по вершинам графа сократит расстояние до самой дальней точки. Задачи этого рода называют задачами о размещении *p-центров*, где *p* – это количество центров. Поли-центры (от греч. πολύς — «многочисленный»), так же как и центры, могут быть *внешними*, *внутренними* и *внешне-внутренними*.

Существуют два рода задач о p-центрах: *прямая* и *обратная*.

В *прямой задаче* задан предельный взвешенный радиус **λ** и требуется найти множество удовлетворяющих ему центров (каждый центр может включать одну или несколько вершин).

В *обратной задаче* известно *количество* центров, а требуется найти их распределение по вершинам и подобающий минимальный взвешенный радиус **λ.**

Поскольку обратная задача легко решается через прямую, основное внимание уделим прямой задаче.

21.2. Геометрическая разминка

Идею решения прямой задачи рассмотрим на геометрической аналогии. Пусть точки на плоскости (рис. 21-1) обозначают острова в океане, а число **R** — это предельная дальность полёта тамошних самолётов. Будем искать минимальное подмножество точек, подходящих для промежуточных аэродромов. Таковыми будут те из них, из которых остальные точки достижимы в пределах радиуса **R**.

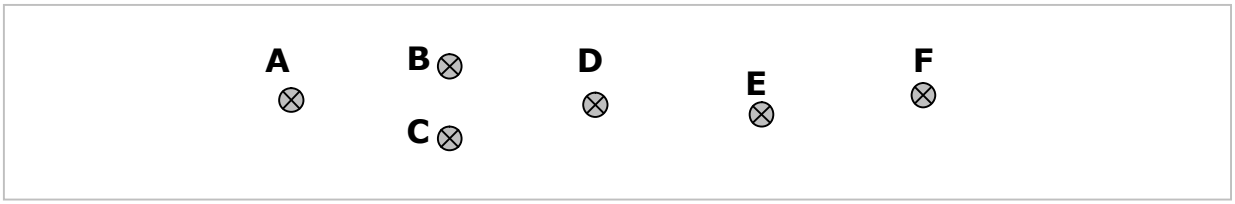


Рис. 21-1 — Ряд точек на плоскости

Из каждой точки очертим круг радиуса **R** (на рис. 21-2 показаны лишь три из шести кругов). Понятно, что центр круга достижим из всех лежащих в круге точек, включая и сам центр. Назовём точки в круге *ограниченной входящей гаммой* по отношению к центру круга. Например, входящая гамма для точки *D* — это точки *B, C, D, E*.

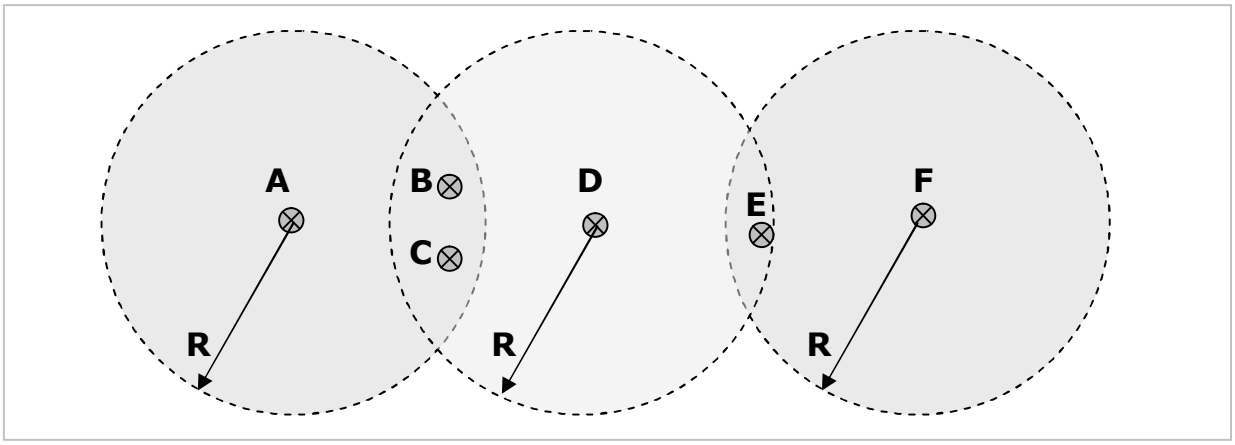


Рис. 21-2 — Круги, соответствующие центральным точкам A, D, F

Далее зададимся вопросом: если некоторая *центральная* точка не попадает в круг другой *центральной* точки, то из каких точек достижимы они обе? Ясно, что из тех точек, что лежат на пересечении входящих гамм (на пересечении кругов). Так, центральные точки *A* и *D* обе достижимы только из точек *B* и *C*. Аналогично пара точек *D* и *F* достижима только из точки *E* . А пара точек *A* и *F* (обе вместе) вообще не достижима из других точек, поскольку их круги (гаммы) не пересекаются. Очевидно, что точки, лежащие на *пересечении* кругов, с б**о**льшим правом претендуют на роль p-центров, поскольку из них достижимо больше других точек.

Найдя все области пересечений, из которых достижимы *пары* точек, попытаемся найти области, из которых достижимы *тройки* точек. Ясно, что к паре точек можно добавить третью, если пересечение её гаммы с гаммой этой пары не пусто. Продолжая процесс, можно отыскивать области, из которых достижимы *четвёрки*, *пятёрки* точек и так далее, пока дальнейшее расширение множества достигаемых точек станет невозможным (пересечения гамм окажутся пустыми). Так, если не обнаружится областей, из которых достижимы *четвёрки* точек, то нет смысла искать области для *пятёрок*.

В ходе накопления всё новых областей можно отбрасывать те подмножества достигаемых точек, которые входят в более мощные. Например, если какая-то пара точек входит ещё и в тройку, то пару отбрасываем. В итоге получим сравнительно небольшой набор из множеств центральных точек и соответствующих им *пересечений* — кандидатов на роль p-центров. Применив затем алгоритм поиска наименьшего покрытия (ЗНП), найдём минимальный набор множеств достигаемых точек, в сумме покрывающий все точки. Соответствующие этим множествам *области пересечений* и будут искомыми p-центрами.

На рис. 21-3 показан результат поиска точек, из которых все остальные достижимы в пределах радиуса **R**. Здесь найдены два центра. В первый входят две точки (*B* и *C*) — из любой из них достижимы точки *A*, *B*, *C*, *D*. Второй центр составляет точка *E*, из которой достижимы точки *D*, *E* и *F*.

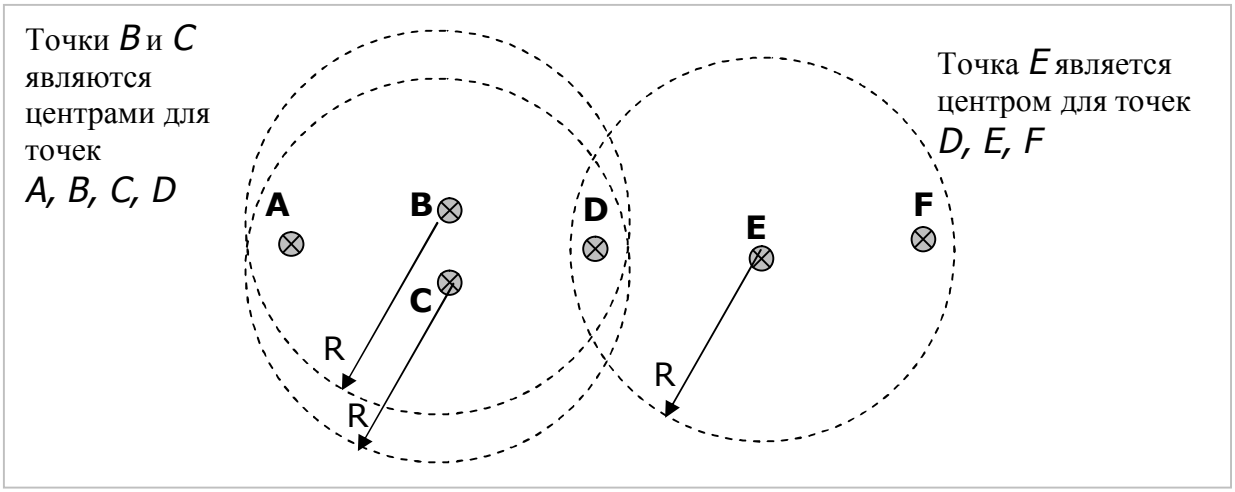


Рис. 21-3 — Левый центр (точки B и C) и правый центр (точка E)

Опираясь на эту геометрическую аналогию, опишем порядок поиска *внешних* р-центров графа с заданным взвешенным расстоянием **λ** (здесь **λ** является подобием радиуса **R**):

- Для каждой вершины графа построим *входящую* гамму, ограниченную взвешенным расстоянием **λ** (подобие круга).
- Последовательно сформируем набор из непустых пересечений гамм (множеств вершин), из которых достижимы: 1) отдельные вершины, 2) пары вершин, 3) тройки вершин, 4) четвёрки вершин и т.д. Если какое-то подмножество достигаемых вершин входит в более мощное, то менее мощное подмножество отбрасываем.
- К полученному набору подмножеств достигаемых вершин применим алгоритм поиска наименьшего покрытия (ЗНП). Соответствующие этому покрытию множества *пересечений* дадут р-центры графа для взвешенного расстояния **λ**.

Аналогично отыскиваются *внутренние* р-центры с тем отличием, что вначале строятся *исходящие* гаммы, ограниченные взвешенным расстоянием **λ.**

21.3. Решение прямой задачи

Перечисленные выше этапы решения прямой задачи рассмотрим на примере поиска р-центров в графе, показанном на рис. 21-4. Взвешенный радиус **λ** примем равным единице, веса вершин и длины рёбер этого графа также будут равны единице, что облегчает вычисления «в уме».

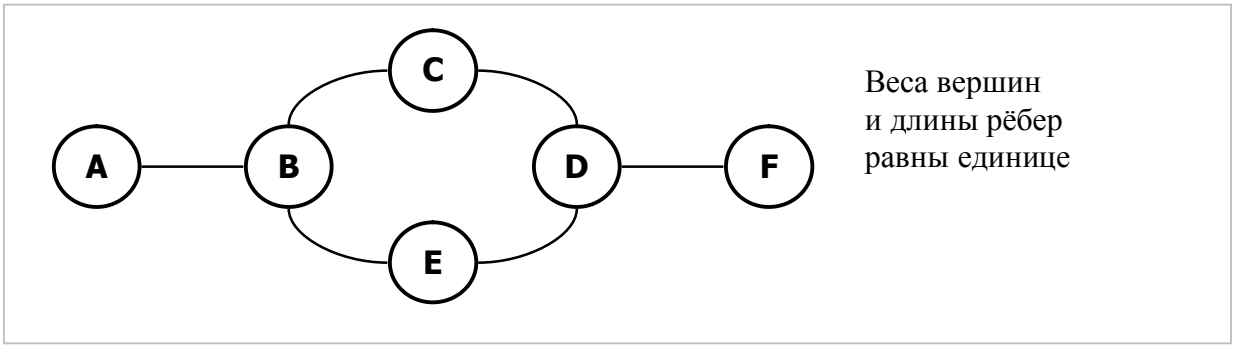


Рис. 21-4 — Пример графа с единичными длинами рёбер
и единичными весами вершин

21.3.1. Формирование входящих гамм

Сконструируем объект (аналог круга) для хранения *ограниченной входящей гаммы* вершины вместе с самой вершиной, вот его объявление:

```
TGamma = class(TItem)
    mRoot  : TNode;  // достигаемая (центральная) вершина
    mGamma : TSet;   // вершины, из которых она достижима
    constructor Create(aRoot: TNode; aGamma: TSet);
    destructor Destroy;  override;
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;
```

Реализацию этого простого класса ищите в листинге модуля **Graph**.

Найти каждую такую ограниченную гамму можно слегка модифицированным алгоритмом Дейкстры. Но для многократного поиска гамм (для разных вершин и для разных значений **λ**) гораздо выгодней предварительно построить карту дальних указателей, как мы сделали это в предыдущей главе модифицированным методом Флойда-Уоршелла. После формирования карты методом **InitMap**, все входящие гаммы вершин, ограниченные взвешенным расстоянием **λ**, строятся вспомогательными методами **GenLimitGamma** и **GenLimitGams** (первый метод вызывается из второго).

Листинг 21-1 — Функции формирования множества
ограниченных входящих гамм

```
// Формирование множества вершин, из которых узел aNode
// достижим в пределах взвешенного расстояния aLambda
// (ограниченная входящая гамма)
// Применяется созданная ранее карта дальних указателей.

function TGraph.GenLimitGamma(aNode: TNode; aLambda: integer): TSet;
var FL : TFarLink;     // очередная дальняя связь
begin
  Result:= CreateSet;
  with aNode.mFarLinks do begin  // просматриваем дальние связи вершины
    FL:= GetFirst as TFarLink;
    while Assigned(FL) do begin
      if aLambda >= (FL.mDist * aNode.mValue) // если в достижимых пределах
```

```
        then Result.Insert(FL.mNodeFar);       // то вставляем в множество
      FL:= GetNext as TFarLink;
    end;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Построение множества ограниченных гамм типа TGamma

function TGraph.GenLimitGams(aLambda: integer): TSet;
var  Node : TNode;
     GammaSet: TSet;
     GammaItem: TGamma;
begin
  Result:= CreateSet;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    GammaSet:= GenLimitGamma(Node, aLambda);  // формируем огранич. вх. гамму
    GammaItem:= TGamma.Create(Node, GammaSet);// формируем пару вершина+гамма
    Result.Insert(GammaItem);
    Node:= NodeNext;
  end;
end;
```

Для рассматриваемого нами графа эти методы сформируют множество объектов класса **TGamma**. Шесть таких объектов построены по данным рис. 21-4 и показаны в табл. 21-1 (это построение легко проверить в уме).

Табл. 21-1 — Области достижимости отдельных вершин при λ=1

Корневая вершина mRoot	Откуда достигается при λ=1, mGamma					
A	A	B				
B	A	B	C		E	
C		B	C	D		
D			C	D	E	F
E		B		D	E	
F				D		F

21.3.2. Формирование пересечений

На втором этапе создаём пересечения ограниченных гамм с одновременным формированием множеств достигаемых вершин. Для хранения этих объектов тоже применим особый класс, построенный на основе **TCostSet**, — объекта, используемого при поиске наименьших покрытий (ЗНП):

```
  TAreal = class(TCostSet)
    // поле mSet - достигаемые вершины, унаследовано от TCostSet
    mGamma : TSet;  // вершины, из которых достижимы вершины в mSet
    constructor Create(aAreal, aGamma: TSet);
    destructor Destroy;  override;
    procedure Print(var aFile: TextFile); override;
  end;
```

В поле **mSet** будут накапливаться множества достигаемых вершин (одиноких вершин, затем пар, затем троек, четвёрок и т.д.). Поле **mGamma** хранит пересечение, из которого достижимы эти вершины в **mSet**. Обратите внимание, что с разрастанием множества достигаемых вершин **mSet**, сужается множество **mGamma**.

Для накопления объектов **TAreal** отлично подходит универсальный буфер **TBuffer**. Будем использовать его в режиме очереди (кольца), извлекая ранее созданные объекты функцией **Get**, и занося туда же новые процедурой **Put**.

Далее я ссылаюсь на метод **TGraph.GenIntersections** (листинг 21-2). Он принимает предварительно построенное множество ограниченных входящих гамм. При инициализации в пустой пока буфер результата заносится фиктивный элемент, содержимое которого показано в табл. 21-2. Этот «затравочный» элемент служит основой для последующего формирования достигаемых вершин в первом внешнем цикле, и будет удалён в этом же цикле.

Табл. 21-2 — Начальный фиктивный элемент TAreal

Достигаемые вершины mSet (пусто)	Откуда достигаются (λ=1) mGamma					
	A	B	C	D	E	F

Итак, *внешний* цикл извлекает и обрабатывает все имеющиеся на текущий момент элементы буфера. Напомню, что в первом цикле будет извлечён единственный фиктивный элемент. *Внутренний* цикл обработки элемента перебирает множества ограниченных гамм (из параметра **aGams**), пытаясь к текущему элементу «прилепить» ещё одну достигаемую вершину (расширить **mSet**). Присоединение возможно при двух условиях: 1) добавляемой вершины ещё нет в поле **mSet**, и 2) пересечение гаммы этой вершины с гаммой обрабатываемого элемента (с полем **mGamma**) не пусто. В случае успеха создаётся и помещается в кольцевой буфер новый элемент **TAreal**. Поскольку по отношению к фиктивному элементу оба условия выполняются, первый внешний цикл пополнит кольцевой буфер восемью элементами, показанными в табл. 21-3. Это *одиночные* вершины, достигаемые из соответствующих им гамм.

Табл. 21-3 — Содержимое кольцевого буфера после 1-го цикла

Достигаемые одиночные вершины mSet						Откуда достигаются (λ=1) mGamma					
A						A	B				
	B					A	B	C		E	
		C					B	C	D		
			D					C	D	E	F
				E			B		D	E	
					F				D		F

Обратите внимание, что после извлечения из буфера фиктивный элемент естественным образом исчез, поскольку если к очередному элементу не «лепится» ни одной вершины, он вновь возвращается в буфер, но здесь не тот случай.

Второй внешний цикл сформирует *пары* достигаемых вершин и соответствующие им гаммы (табл. 21-4). Принцип тот же, что и в первом цикле: добавляем к **mSet** отсутствующие вершины, когда пересечение гамм не пусто. Например, при добавлении к множеству **mSet**, состоящему из вершины *A* новой вершины *B* ищется пересечение их гамм:

$$(A\text{-}B) \bullet (A\text{-}B\text{-}C\text{-}E) = A\text{-}B$$

Заметим, что при последующем добавлении к вершине *B* вершины *A* порождается точно такая же пара (дубликат) и засоряет буфер. Во избежание дубликатов перед созданием нового элемента проверяется, создавался ли он ранее? Для проверки введено вспомогательное множество **Existed**, хранящее ранее созданные достигаемые подмножества.

Табл. 21-4 — Содержимое кольцевого буфера после 2-го цикла

Достигаемые пары вершин mSet						Откуда достигаются (λ=1) mGamma					
A	B					A	B				
A		C					B				
A				E			B				
	B	C					B	C			
	B		D					C			
	B			E			B			E	
		C	D					C	D		
		C		E			B		D		
		C			F				D		
			D	E					D	E	
			D		F				D		F
				E	F				D		

Обратите внимание, что теперь в буфере остались только *пары* достигаемых вершин, а *одиночные* при извлечении покинули буфер и не возвращены в него, поскольку являются подмножествами пар.

В третьем внешнем цикле обрабатываются пары вершин в попытке сформировать элементы с тройками достигаемых вершин (табл. 21-5).

Табл. 21-5 — Содержимое кольцевого буфера после 3-го цикла

Достигаемые тройки вершин mSet						Откуда достигаются (λ=1) mGamma					
A	B	C					B				
A	B			E			B				
	B	C	D					C			
	B	C		E			B				
	B		D	E						E	
		C	D		F				D		
		C	D	E					D		
			D	E	F				D		

Наконец в четвёртом цикле ищутся области из четырёх достигаемых вершин, их окажется лишь две. К ним добавятся и две тройки (*B-C-D* и *B-D-E*), поскольку эти тройки не удалось расширить, и они не вошли подмножествами в другие множества. По этой причине тройки возвращены в кольцевой буфер.

Табл. 21-6 — Содержимое кольцевого буфера после 4-го цикла

Достигаемые тройки и четвёрки вершин mSet						Откуда достигаются (λ=1) mGamma					
A	B	C		E			B				
	B	C	D					C			
	B		D	E						E	
		C	D	E	F				D		

В пятом цикле уже не удаётся присоединить ни одной вершины, и потому кольцевой буфер не изменится, — на этом процедура формирования буфера подмножеств **mSet** завершается.

Теперь готовый буфер «скармливаем» функции **CollectMinCover** и находим наименьшее покрытие. В табл. 21-6 найденное решение выделено цветом, здесь для **λ**=1 найдено два центра в вершинах *B* и *D* (рис. 21-5).

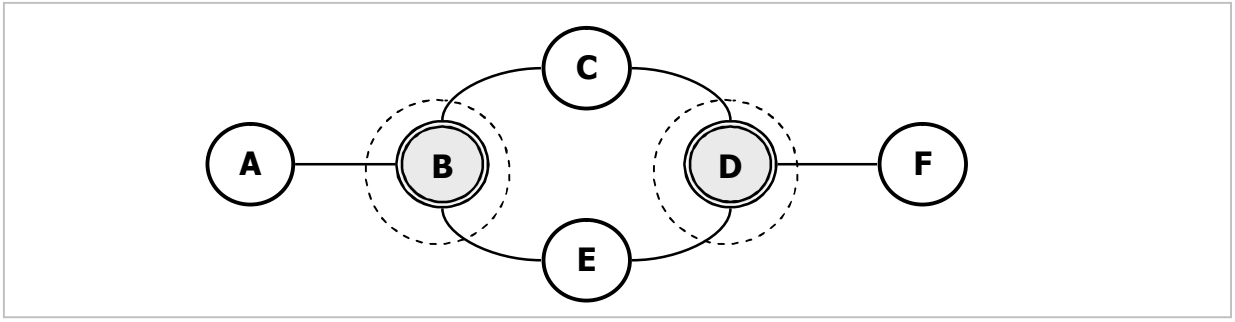


Рис. 21-5 — Два центра графа для λ =1

Построив такие же таблицы для взвешенного расстояния **λ**=2, получим один центр, состоящий из двух вершин, как показано на рис. 21-6. Здесь расстояние к любой вершине графа от вершин *C* и *E* составляет не более 2-х единиц. Обратите внимание: в первом случае для **λ=1** было найдено *два* центра по одной вершине в каждом, во втором случае для **λ=2** найден *один* центр из двух вершин, — экстренная служба может быть размещена в любой из них.

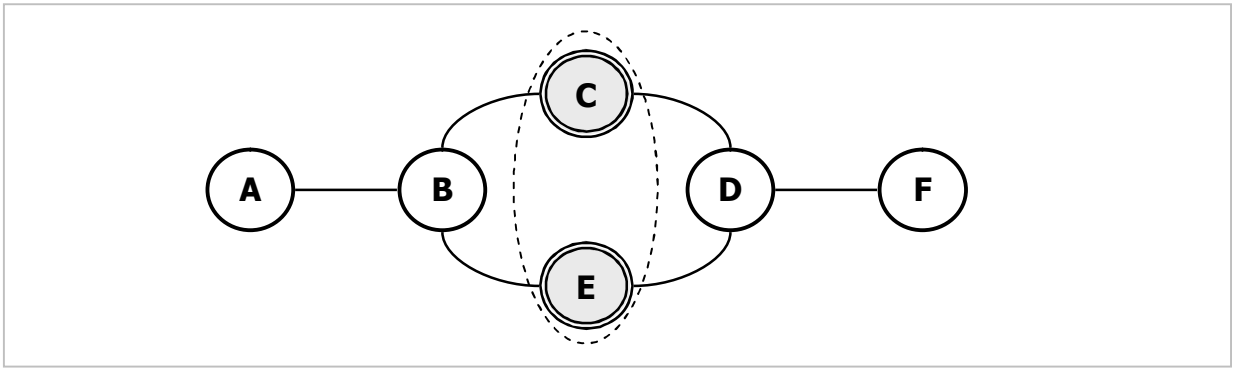


Рис. 21-6 — Один центр (из двух вершин) для λ =2

Листинг 21-3 содержит метод **GenPCentLambda**, решающий прямую задачу поиска p-центров по заданному значению **λ**:

Листинг 21-2 — Формирование всех возможных пересечений гамм

```
function TGraph.GenIntersections(aGams: TSet): TBuffer;
var
    GammaItem : TGamma;    // элемент для входящих гамм
        Nodes : TSet;      // достигаемые вершины (TAreal.mSet)
     GammaSet : TSet;      // откуда достигаются (TAreal.mGamma)
         Temp : TSet;      // здесь вычисляется пересечение гамм
      Existed : TSet;      // множество множеств уже обработанных областей
        Areal : TAreal;    // очередной элемент "область"
         Flag : boolean;   // признак пополнения буфера
          Cnt : integer;   // текущее количество элементов в буф. результ.
            i : integer;   // индекс
begin
  Result:= TBuffer.Create;     // буфер результата
  Temp:= CreateSet;            // здесь вычисляется пересечение гамм
  Existed:= CreateSet;         // множество множеств уже обработанных областей
  Nodes:= CreateSet;           // достигаемые вершины (TAreal.mSet)
  GammaSet:= CreateSet;        // откуда достигаются (TAreal.mGamma)
  GammaSet.CopyItems(mNodes); // копия множества всех вершин
  // Создаём и помещаем в буфер фиктивный элемент TAreal такой, что:
  // mNodes - множество достигаемых вершин ПУСТО
  // mGamma - множество вершин, из которых достигается, содержит ВСЕ вершины
  Result.Put(TAreal.Create(Nodes, GammaSet));
  repeat
    Cnt:= Result.GetCount;  // текущее количество элементов в буфере
    for i:=1 to Cnt do begin
      // Цикл по всем элементам буфера
      Flag:= false;                  // сброс признака пополнения буфера
      Areal:= Result.Get as TAreal; // очередной элемент из буфера
      // Перебор всех элементов исходного множества
      GammaItem:= aGams.GetFirst as TGamma;
      while Assigned(GammaItem) do begin
```

```
        if not Areal.mSet.Exist(GammaItem.mRoot) then begin
          // Если текущая область ещё не содержит корневой вершины
          // то вычисляем пересечение входящих гамм
          Temp.CopyItems(Areal.mGamma);      // гамма в текущей области
          Temp.Mul(GammaItem.mGamma);        // пересекаем с гаммой вершины
          if Temp.GetCount <> 0 then begin
            // если пересечение не пусто,
            // пытаемя создать и добавить в буфер новую область
            Nodes:= CreateSet;                // достигаемые (TAreal.mSet)
            Nodes.CopyItems(Areal.mSet);      // существующие вершины
            Nodes.Insert(GammaItem.mRoot);    // плюс ещё одна вершина
            if Existed.Exist(Nodes) then begin
              // Если это подмножество достигаемых уже существует в наборе
              Nodes.Free;     // то удаляем дубликат
              Nodes:= nil;    // и ничего не вставляем
            end else begin
              // Если получено новое подмножество достигаемых вершин
              Existed.Insert(Nodes);      // запоминаем для будущих проверок
              // создаём новую область и вставляем в буфер
              GammaSet:= CreateSet;      // откуда достигаются (TAreal.mGamma)
              GammaSet.CopyItems(Temp); // копия пересечения GammaSet= Temp
              Result.Put(TAreal.Create(Nodes, GammaSet));
            end; // else
            Flag:= true;  // установить признак пополнения буфера
          end; // if
        end; // if
        GammaItem:= aGams.GetNext as TGamma;  // перебор множества гамм
      end; // while
      // После обработки очередного элемента буфера проверяем:
      // Если обработанный элемент буфера породил хотя бы один новый элемент,
      // то он является подмножеством этого нового и исключается
      // из дальнейшей обработки, а иначе возвращается в буфер
      if Flag then begin
        Existed.Delete(Areal.mSet);// удаляем достигаемые из проверяемых
        Areal.Free;                // удаляем ненужный элемент буфера
      end else begin
        Result.Put(Areal);         // а иначе возвращаем элемент в буфер
      end;
    end; // for
    // выход, если буфер не изменился
  until not Flag and (Cnt=Result.GetCount);
  Temp.Free;
  Existed.Free;
end;
```

Листинг 21-3 — Метод, решающий прямую задачу нахождения Р-центров

```
// Поиск множества центров
// с заданной константой проникновения aLambda (дистанцией)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPCentLambda(aLambda: integer; aDirect: TCenter): TCostSet;
var   Gams : TSet;
       Buf : TBuffer;
begin
  // Формирование карты дальних указателей (направление карты инверсно):
  case aDirect of
    InCenter:    InitMap(OutCenter);
    OutCenter:   InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование входящих гамм
```

```
  // с заданной константой проникновения (дистанцией) aLambda
  Gams:= GenLimitGams(aLambda);
  // Формирование всех непустых пересечений
  Buf:= GenIntersections(Gams);
  // Решение задачи о минимальном покрытии (ЗНП)
  Result:= CollectMinCover(mNodes, Buf);
  // Удаляем результат из буфера
  RemoveItemsFromBuf(Buf, Result.mSet);
  // Освобождаем множество входящих гамм и буфер
  Gams.ClrAndDestroy;
  Gams.Free;
  Buf.ClrAndDestroy;
  Buf.Free;
end;
```

Обратите внимание, что перед вызовом метода **InitMap** параметр инвертируется, поскольку для поиска *внешних* p-центров строится карта, используемая при поиске *внутреннего* одиночного центра, и наоборот.

21.4. Решение обратной задачи

Вспомним формулировку обратной задачи о p–центрах: дана кратность центров **p**, следует найти эти центры (вершины графа), а также соответствующий им взвешенный радиус **λ**.

Поставленная цель достигается многократным решением прямой задачи с растущим значением **λ**. Так, при **λ=0** количество центров будет равно количеству вершин графа. По мере роста **λ** численность центров снижается, вплоть до единицы. Когда количество центров достигнет нужного значения **p**, задача будет решена. Вопрос лишь в том, с каким шагом наращивать **λ**? Слишком осторожное приращение повлечёт лишнюю работу, поскольку далеко не каждое приращение радиуса сокращает численность центров. Рассудим следующим образом. Предположим, мы нашли все *взвешенные* расстояния между всеми вершинами графа и расположили эти числа без повторений в порядке возрастания. Тогда очередное значение **λ** можно выбрать из этого ряда, поскольку промежуточные значения **λ** не уменьшат число центров. Итак, **λ** выбираем из множества **Lambdes**, заранее сформированного локальной процедурой **LambdasCreate**.

Листинг 21-4 — Решение обратной задачи поиска p-центров

```
function TGraph.GenPCenters(aPoly: integer;        // кратность центров
                            var aLambda: integer;  // радиус
                            aCenter: TCenter // InCenter,OutCenter,InOutCenter
                            ): TCostSet;

var Lambdes : TSet;  // лямбда-список (возрастающее множество расстояний)
       LRec : TSortedNum; // элемент лямбда-списка
     Lambda : integer;    // текущий взвешенный радиус
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества взвешенных расстояний (уникальных)
  procedure LambdasCreate;

    procedure AddLambdas(aNode: TNode);
```

```
    var FL : TFarLink;     // очередная дальняя связь
        SN : TSortedNum;   // элемент сортированного списка чисел
        Lambda: integer;   // взвешенное расстояние от текущей вершины
    begin
      with aNode.mFarLinks do begin
        FL:= GetFirst as TFarLink;
        while Assigned(FL) do begin
          Lambda:= FL.mDist * aNode.mValue;
          SN:= TSortedNum.Create(Lambda);
          if not Lambdes.Insert(SN) then SN.Free;
          FL:= GetNext as TFarLink;
        end;
      end;
    end;
  var  Node : TNode;     // текущая вершина
  begin { LambdasCreate }
    Lambdes:= CreateSet;
    Node:= NodeFirst;
    while Assigned(Node) do begin   // перебор вершин
      AddLambdas(Node);             // обработать вершину
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenPCenters }
  // Формирование карты дальних указателей:
  case aCenter of
    InCenter    : InitMap(OutCenter);
    OutCenter   : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование лямбда-списка (возрастающего множества расстояний)
  LambdasCreate;
  // Выбор первого (наименьшего) элемента взвешенного расстояния
  LRec:= Lambdes.GetFirst as TSortedNum;
  repeat
    Lambda:= LRec.mNumber;            // очередное взвешенное расстояние
    Result:= GenPCentLambda(Lambda, aCenter);  // множество центров
    Writeln('Lambda= ', Lambda:3, '  P= ', Result.mSet.GetCount:3);
    // Если количество центров достигнуто, то выход:
    if Result.mSet.GetCount <= aPoly then Break;
    // а иначе очищаем результат и берём следующий по величине радиус
    Result.ClrAndDestroy;
    Result.Free;
    Result:= nil;
    // Выбор очередного элемента взвешенного радиуса (по возрастанию)
    LRec:= Lambdes.GetNext as TSortedNum;
  until not Assigned(LRec);
  // Возвращаем взвешенный радиус:
  aLambda:= Lambda;
  // Очистка и уничтожение списка расстояний
  Lambdes.ClrAndDestroy;
  Lambdes.Free;
  // Очистка карты дальних указателей:
  DoneMap;
end;
```

21.5. Испытания

21.5.1. Прямая задача

Ниже дана программа для тестирования решения прямой задачи.

Листинг 21-5 — Программа поиска Р-центров
по заданному взвешенному расстоянию λ

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas',
  Assembly in '..\Common\Assembly.pas';

var Gr: TGraphChars;
    PCenters: TCostSet;
    Lambda : integer;
begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  repeat
    Write('Lambda= '); Readln(Lambda);
    if Lambda = 0 then Break;
    PCenters:= Gr.GenPCentLambda(Lambda, OutCenter);
    if Assigned(PCenters) then begin
      Writeln('Lambda = ', Lambda);
      PCenters.Expo;
      PCenters.ClrAndDestroy;
      PCenters.Free;
    end;
  until false;
  Gr.Free; // Освобождение графа
end.
```

Для испытания воспользуемся графом, показанным на рис. 21-7.

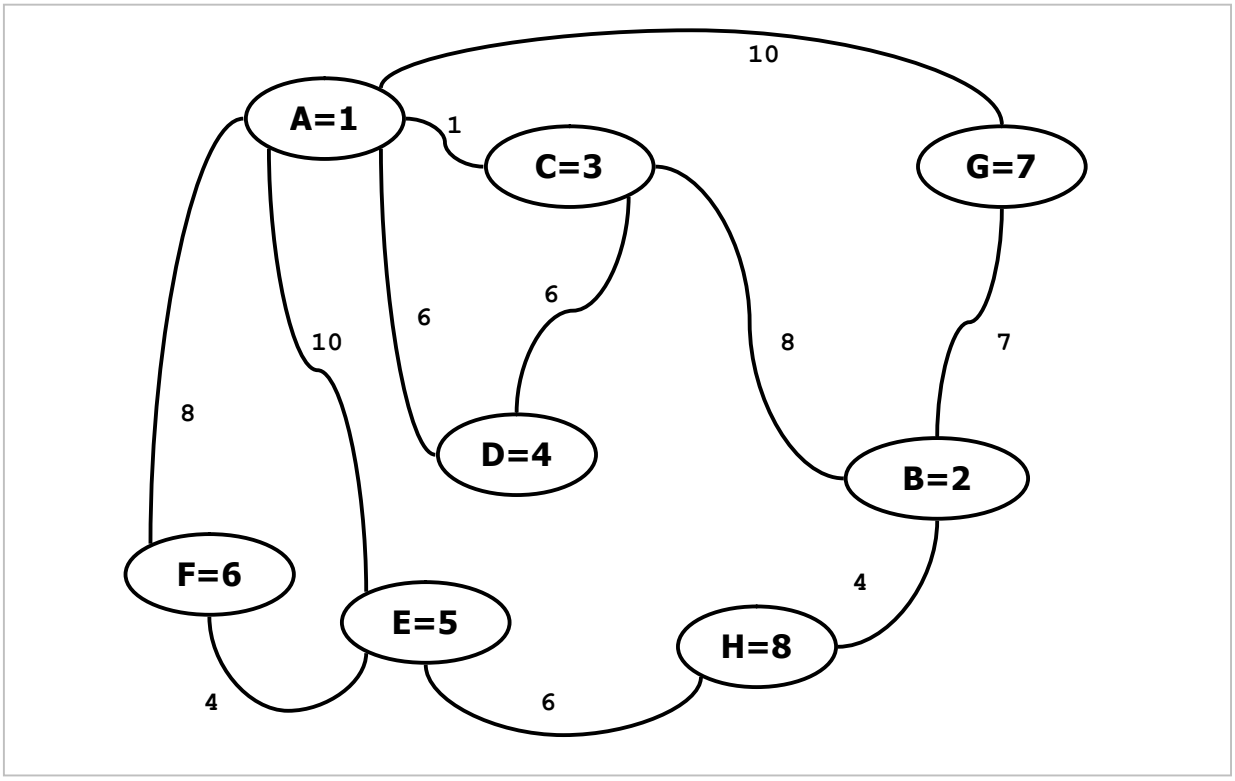


Рис. 21-7 — Взвешенный неориентированный граф

Текст для ввода графа таков:

```
Test.txt - Взвешенный неориентированный граф
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

Протокол работы программы представлен ниже. Для **λ=50** найдены два центра (A и B). Слева от стрелки ==> указаны центральные вершины, а справа — вершины, достигаемые из этих центров с заданным взвешенным расстоянием.

```
Lambda = 50
2 =
{
A ==> A,B,C,D,E,F
B ==> A,B,C,E,G,H
} : 2
```

21.5.2. Обратная задача

Здесь представлена программа для тестирования обратной задачи — поиска p-центров и взвешенного радиуса по заданному количеству центров **P**.

Листинг 21-6 — Программа поиска заданного количества p-центров и взвешенного расстояния λ

```
{$APPTYPE CONSOLE}
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr: TGraphChars;
    PCenters: TCostSet;
    P : integer;
    Lambda : integer;
    Start, Time : TDateTime;
    D : char;
    Direct : TCenter;

begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - -');
  Write('Direction (I/O/A) = '); Readln(D);
  case Upcase(D) of
    'I': Direct:= InCenter;
    'O': Direct:= OutCenter;
    else Direct:= InOutCenter;
  end;
  repeat
    Write('P= '); Readln(P);
    if P=0 then Break;
    case Direct of
      InCenter: Writeln('------------------- IN -------------------');
      OutCenter: Writeln('------------------- OUT -------------------');
      InOutCenter: Writeln('------------------- IN-OUT -------------------');
    end;
    Start:= Now;
    PCenters:= Gr.GenPCenters(P, Lambda, Direct);
    Time:= MilliSecondsBetween(Start, Now);
    if Assigned(PCenters) then begin
      Writeln('- - - - - - - - - - - - - - - - - -');
      PCenters.Expo;
      Writeln('Lambda= ', Lambda);
      Writeln('Time (ms) = ', Time:6:0);
      PCenters.ClrAndDestroy;
      PCenters.Free;
    end;
  until false;
end.
```

Протокол работы программы дан ниже. Промежуточные значения **Lambda** и **P** печатаются здесь для наглядности внутри метода **GenPCenters**, после испытаний эту печать надо отключить.

```
Test_4 - (P-центры)
{
A =  1 ->   { C= 1 D= 6 E= 10 F= 8 G= 10 } : 5
B =  2 ->   { C= 8 G= 7 H= 4 } : 3
C =  3 ->   { A= 1 B= 8 D= 6 } : 3
D =  4 ->   { A= 6 C= 6 } : 2
E =  5 ->   { A= 10 F= 4 H= 6 } : 3
F =  6 ->   { A= 8 E= 4 } : 2
G =  7 ->   { A= 10 B= 7 } : 2
H =  8 ->   { B= 4 E= 6 } : 2
} : 8
- - - - - - - - - - - - - - - - - - - -
Кратность центров P= 2
Lambda=  0  P=  8
Lambda=  1  P=  7
Lambda=  3  P=  7
Lambda=  6  P=  7
Lambda=  8  P=  6
Lambda=  9  P=  6
Lambda= 10  P=  6
Lambda= 13  P=  6
Lambda= 14  P=  6
Lambda= 16  P=  6
Lambda= 18  P=  5
Lambda= 20  P=  4
Lambda= 24  P=  4
Lambda= 27  P=  4
Lambda= 28  P=  4
Lambda= 30  P=  4
Lambda= 32  P=  4
Lambda= 33  P=  4
Lambda= 36  P=  4
Lambda= 48  P=  3
Lambda= 49  P=  3
Lambda= 50  P=  2
- - - - - - - - - - - - - - - - - - - -
{
A ==> A,B,C,D,E,F
B ==> A,B,C,E,G,H
} : 2
Lambda= 50
```

Для 2-центра графа, показанного на рис. 21-7, найдены вершины *A* и *B*, при этом взвешенный радиус составил 50 единиц. В табл. 21-7 даны результаты поиска ещё нескольких p-центров этого графа.

Табл. 21-7 — Р-центры графа, показанного на рис. 21-7

Кратность центров Р	Вершины центров	Взвешенный радиус λ
1	H	77
2	A, B	50
3	A, E, G	48
4	D, F, G, H	20
5	D, E, F, G, H	18

Следующее испытание проведено на не взвешенном орграфе, изображённом на рис. 21-8.

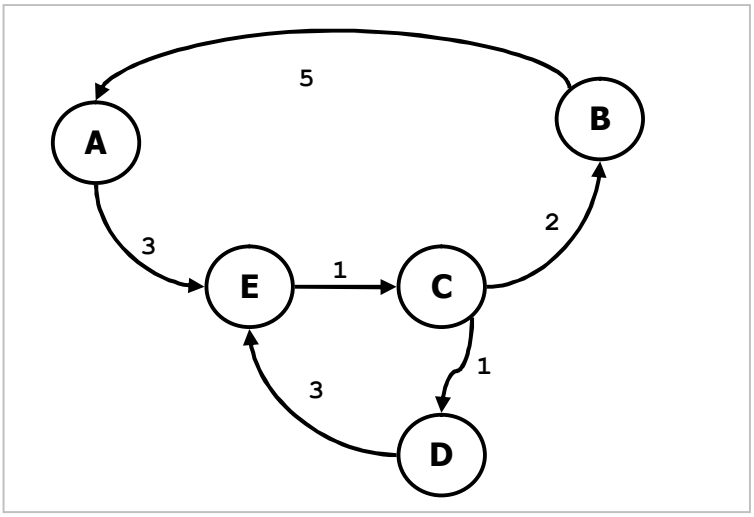


Рис. 21-8 — Не взвешенный орграф

Результаты представлены в табл. 21-8.

Табл. 21-8 — Результаты поиска р-центров на не взвешенном орграфе

Кратность центров Р	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	Радиус λ	Вершины	Радиус λ	Вершины	Радиус λ
1	A	6	B	6	C E	11
2	A, E	3	B, E	3	-	-
3	A, C, E	2	A, B, D	2	A-B-C, A-B-D, A-B-E	5
4	A, B, C, E	1	A, B, C, D	1	-	-

В данном примере не существуют внешне-внутренние 2-центр и 4-центр. Это значит, что расширение множества достигаемых вершин с одной до двух, и с трёх до четырёх не уменьшает взвешенного радиуса графа.

Наконец, ниже даны исходные данные и результаты поиска центров на взвешенном орграфе, показанном на рис. 21-9.

```
Взвешенный орграф (рис. 21-9)
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

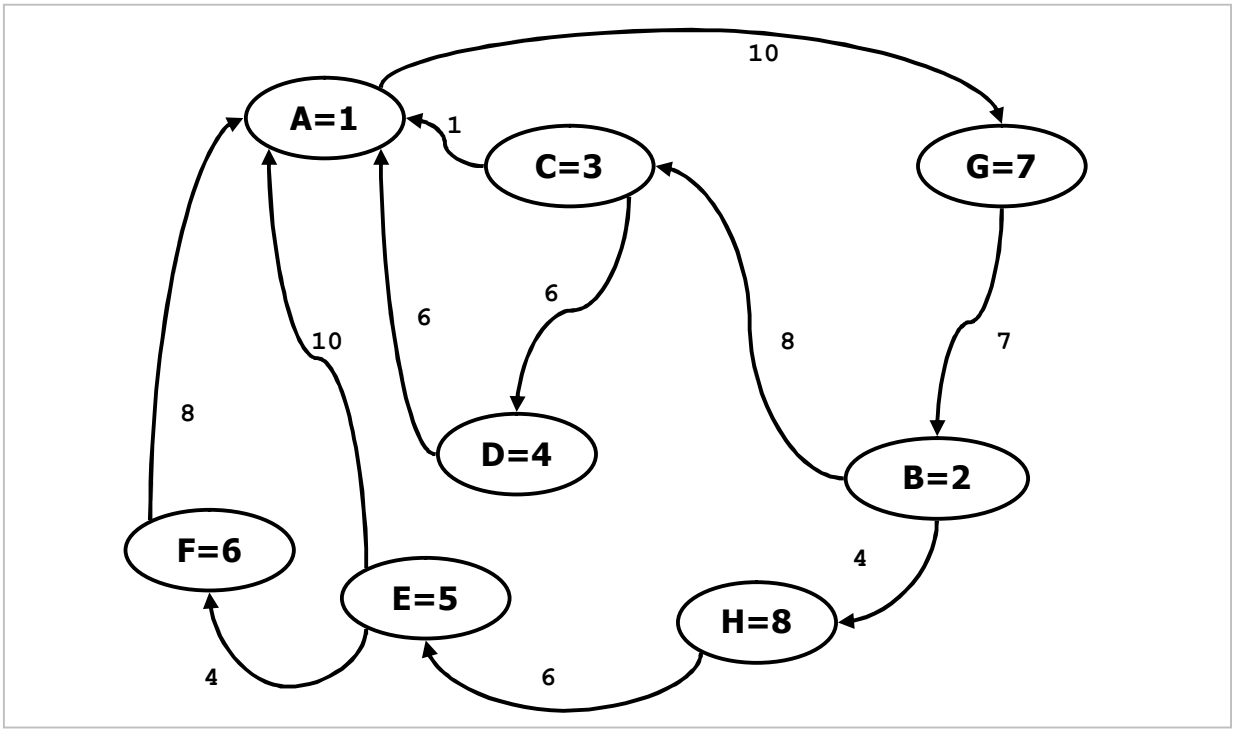


Рис. 21-9 — Взвешенный орграф

Табл. 21-9 — Р-Центры взвешенного орграфа (рис. 21-9)

Кратность Р	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	λ	Вершины	λ	Вершины	λ
1	G	93	A	128	A, B, E, G, H	296
2	C, H	77	A, H	77	A-H, B-H, G-H	234
3	B, E, G	56	A, E, G	48	A-F-H, B-F-H, G-F-H	185
4	C, E, G, H	24	D, F, G, H	20	E, F, G, H	148
5	-	-	D, E, F, G, H	18	D, E, F, G, H	78

Отметим отсутствие внешнего 5-центра, а также наличие нескольких вариантов внешне-внутренних 1-, 2- и 3-центров.

21.6. Итоги

- Задача о р-центрах заключается в поиске заданного количества центров, то есть, подмножеств таких вершин, расстояние от которых до самых дальних вершин минимально. Каждый из Р центров в общем случае является подмножеством вершин. Различают прямую и обратную задачу о р-центрах.
- В прямой задаче задан взвешенный радиус λ и требуется найти множество удовлетворяющих ему центров.
- В обратной задаче известна кратность центров Р, и надо найти их распределение по вершинам и соответствующий минимальный взвешенный радиус λ.
- Обратная задача о р-центрах решается через многократное решение прямой задачи с возрастающим значением λ.

21.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Глава 5, стр. 98
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Глава 8, стр. 265
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 22
Р-медианы

22.1. Медианы и р-медианы

При размещении *экстренных* служб (см. задачи о центрах графа) стараются минимизировать *максимальные* расстояния от центра к периферии. Но когда цель состоит в экономии, то стремятся снизить *сумму* взвешенных расстояний ко всем вершинам, — такие задачи называют *минисуммными*. Вот несколько примеров этого рода.

Пусть островной архипелаг с мостами образует неориентированный связанный граф. Поищем остров для постройки электростанции с тем, чтобы минимизировать общую длину линий электропередач, протянутых от неё к другим островам по существующим мостам. Искомая вершина графа называется *медианой*. Если острова разнятся уровнем потребления энергии, то это влияет на толщину проводов, и тогда затраты будут пропорциональны как расстоянию до острова, так и его «аппетиту» (весу вершины), — здесь целью будет свести к минимуму общую массу проводов.

Теперь обратимся к орграфу. Пусть архипелаг соединён мостами с односторонним движением, и эта связь будет сильной, что гарантирует взаимную достижимость любой пары островов. Поищем место для базы снабжения островов товарами так, чтобы минимизировать затраты на их доставку от базы к потребителям. Здесь не учитываем затраты на возврат порожнего транспорта (обратный маршрут порожняка в общем случае отличается от прямого). Если населённость островов различна, то количество доставляемых товаров будет зависеть от веса вершины-адресата, и тогда следует минимизировать *сумму взвешенных* расстояний от искомой вершины к остальным, — искомая вершина называется *внешней медианой* орграфа.

Теперь найдём место заводу для переработки отходов. В сравнении с базой снабжения, здесь груз доставляется в обратном направлении — от периферии к центру переработки, или точнее к вершине, называемой *внутренней медианой* орграфа.

Наконец, при выборе острова для постройки столицы желательно минимизировать транспортные издержки для населения архипелага (в столицу и обратно ездят все). Здесь минимальные транспортные затраты соответствуют вершине, называемой *внешне-внутренней медианой* орграфа.

Пока речь шла о единственной искомой вершине. Но пунктов обслуживания может быть несколько, для этого случая введём понятие кратных *р-медиан*, где *р* — это кратность медианы. Разумеется, что равномерное распределение пунктов по графу снижает сумму взвешенных расстояний. При этом любая вершина, не входящая в подмножество *р-медианы*, прикрепляется (в смысле своего

обслуживания) к ближайшей из этого подмножества. Для орграфов *кратные* р-медианы могут быть *внешними*, *внутренними* и *внешне-внутренними*.

Все задачи о медианах и кратных р-медианах решаются одним алгоритмом. Существуют задачи, где надо учесть ещё и затраты на постройку самой службы (когда эти затраты различны для различных вершин). Решение достигается вводом дополнительного поля в вершину графа и небольшой модификацией основного алгоритма, — этот случай предлагаю читателю в качестве упражнения.

22.2. Немного формализма

Возьмём *i*-ю вершину орграфа и определим для неё сумму взвешенных расстояний ко всем остальным *j*-м вершинам орграфа, эта характеристика вершины называется *внешним передаточным числом*:

$$\sigma_o(x_i) = \sum_{x_j \in X} v_j \bullet d(x_i, x_j)$$

Схожим образом определяется *внутреннее передаточное число*, но здесь суммируются расстояния от всех *j*-х вершин к *i*-й вершине:

$$\sigma_t(x_i) = \sum_{x_j \in X} v_j \bullet d(x_j, x_i)$$

Здесь X — это множество всех вершин графа, v_j — вес *j*–й вершины, $d(x_i, x_j)$ — прямое расстояние от *i*–й вершины к *j*–й, $d(x_j, x_i)$ — обратное расстояние от *j*–й вершины к *i*–й.

Вершина, претендующая на роль той или иной медианы, должна обладать подобающими минимальными передаточными числами. Для внешне-внутренней медианы минимальной должна быть сумма двух её передаточных чисел. Стало быть, задача поиска единственной медианы решается перебором всех вершин графа, с сопутствующим вычислением передаточных чисел.

Сложнее обстоит дело с кратными р-медианами, где передаточные числа определяются так:

$$\sigma_o(X_p) = \sum_{x_j \in X} v_j \bullet d(X_p, x_j)$$

$$\sigma_t(X_p) = \sum_{x_j \in X} v_j \bullet d(x_j, X_p)$$

Здесь Xp — некое подмножество из p вершин, $d(Xp,x_j)$ — прямое расстояние от ближайшей вершины этого подмножества к j–й вершине, $d(x_j,Xp)$ — обратное расстояние от j–й вершины к ближайшей из Xp подмножества.

При поиске кратных р-медиан необходимо перебрать все комбинации по p вершин. Для 1-медианы алгоритм вырождается в линейный перебор вершин, но для значений p, близких к $N/2$, задача обретает экспоненциальную сложность, что вынуждает искать пути сокращения перебора.

22.3. Числовые примеры

Найдём положение медиан в графе, показанном на рис. 22-1.

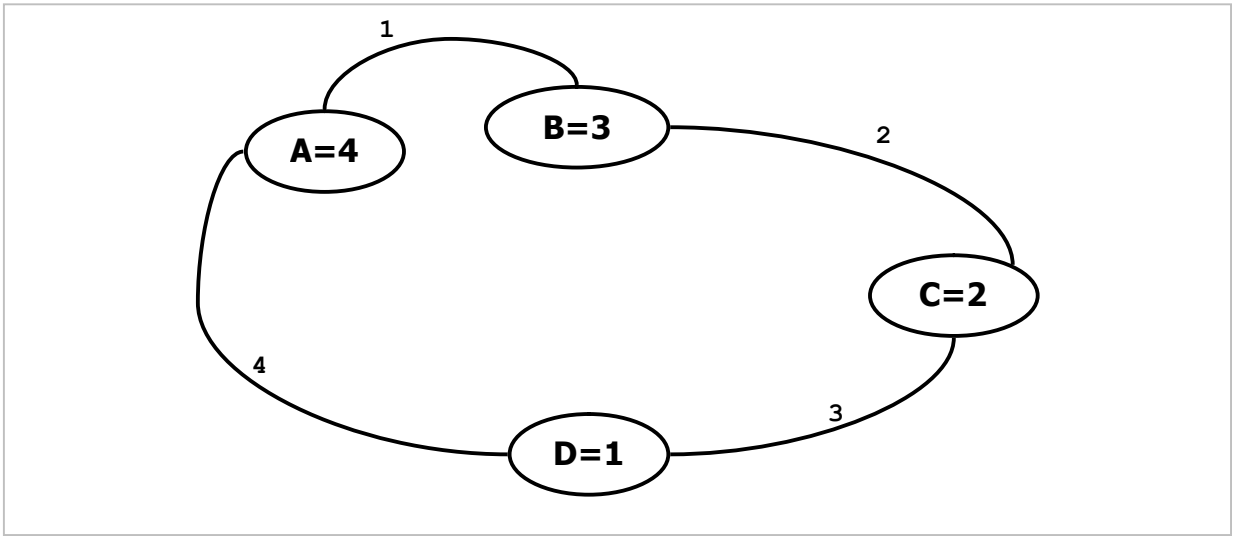


Рис. 22-1 — Неориентированный граф с взвешенными дугами и вершинами

По рисунку составим вспомогательную таблицу (карту) взаимных расстояний $d(i,j)$ между вершинами x_i и x_j (табл. 22-1).

Табл. 22-1 — Взаимные расстояния d(i,j) между i-й и j-й вершинами

Позиция	Исходные вершина Xi							
	A		B		C		D	
	Xj	d	Xj	d	Xj	d	Xj	d
1	A	0	B	0	C	0	D	0
2	B	1	A	1	B	2	C	3
3	C	3	C	2	A	3	A	4
4	D	4	D	5	D	3	B	5

В левом столбце указан порядковый номер целевой j–й вершины относительно исходной i–й вершины. Первой в списке идёт исходная вершина, поскольку

расстояние к ней равно нулю. Прочие вершины следуют далее в неубывающем порядке расстояний *d(i,j)*.

По таблице легко найти сумму взвешенных расстояний (передаточные числа) для любой вершины; так, к примеру, для вершины *A* эта сумма составит:

S = 4•0 + 3•1 + 2•3 + 1•4 = 13

Здесь первый множитель — вес *j*–й вершины, а второй — расстояние от *i*–й к *j*–й вершине. Медианой будет та из них, для которой сумма окажется минимальной (таких вершин может быть несколько), в данном случае это вершина *A*.

Для поиска 2-медиан перебираются все комбинации из *двух* вершин и вычисляются аналогичные суммы с учётом того, что каждая вершина, не принадлежащая медиане, *прикрепится* к ближайшей из *двух* медианных вершин. Сами медианные вершины не вносят вклад в сумму, поэтому слагаемые с их участием отброшены. Например, если центры обслуживания разместить в вершинах *A* и *B*, то сумма взвешенных расстояний подсчитывается только для вершин *C* и *D* и составит:

S = 2•2 + 1•4 = 8

Сходно ищутся все последующие кратные медианы, в табл. 22-2 представлены 1-, 2- и 3-медианы рассмотренного графа.

Табл. 22-2 — Кратные медианы графа на рис. 22-1

Кратность P	Вершины медианы	Стоимость
1	A	13
2	A, C	6
3	A, B, C	3

Теперь рассмотрим поиск медиан в орграфе, представленном на рис. 22-2.

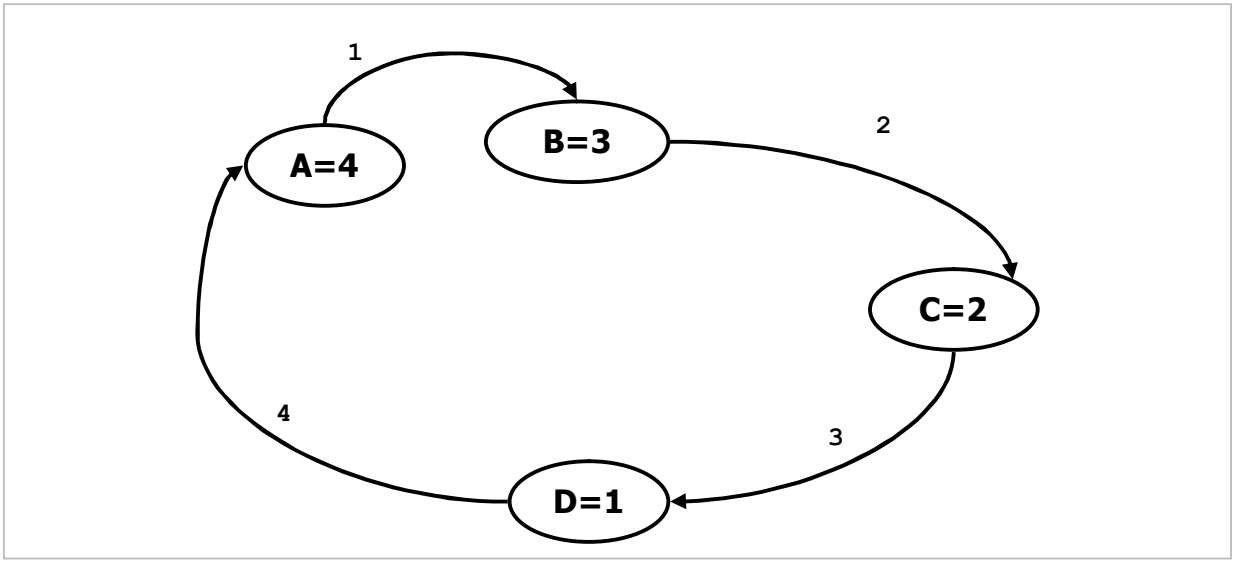


Рис. 22-2 — Орграф с взвешенными дугами и вершинами

Напомню, что в ориентированном графе существуют три рода р-медиан: внешние, внутренние и внешне-внутренние. Для поиска каждой из них надо построить свою таблицу взаимных расстояний. Таблица для поиска внешних медиан строится по направлению дуг (табл. 22-3).

Табл. 22-3 — Взаимные расстояния d(i,j) между i-й и j-й вершинами

Позиция	Исходные вершина Xi							
	A		B		C		D	
	Xj	d	Xj	d	Xj	d	Xj	d
1	A	0	B	0	C	0	D	0
2	B	1	C	2	D	3	A	4
3	C	3	D	5	A	7	B	5
4	D	6	A	9	B	8	C	7

Для поиска внутренних медиан таблица строится в направлении, обратном направлению дуг (табл. 22-4).

Табл. 22-4 — Взаимные расстояния d(j,i) между j-й и i-й вершинами

Позиция	Исходные вершина Xj							
	A		B		C		D	
	Xi	d	Xi	d	Xi	d	Xi	d
1	A	0	B	0	C	0	D	0
2	D	4	A	1	B	2	C	3
3	C	7	D	5	A	3	B	5
4	B	9	C	8	D	7	A	6

Далее ход решения ничем не отличается от описанного выше: перебираются все возможные комбинации из p вершин. Результат поиска представлен в табл. 22-5.

Табл. 22-5 — Р-медианы графа на рис. 22-2

Кратность Р	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	Стоимость	Вершины	Стоимость	Вершины	Стоимость
1	A	15	B	25	A	60
2	A-C	6	B-C	9	A-B	30
3	A-B-C	3	A-B-C	4	A-B-C	10

22.4. Основные идеи алгоритма

Итак, упрощённый поиск р-медиан состоит в решении двух подзадач:

- перебор всех комбинаций из N вершин по P, где N — количество вершин графа;
- подсчёт для каждой комбинации её стоимости и сохранение комбинации с наименьшей стоимостью.

Поскольку для значений P, близких к $N/2$, число комбинаций растёт с ростом N экспоненциально, желательно отсечь заведомо тупиковые ветви перебора. Отсюда следует план разработки улучшенного алгоритма:

- создать процедуру перебора всех комбинаций из N вершин по P;
- создать функцию вычисления стоимости текущей комбинации медианных вершин (текущего р-подмножества);
- разработать упрощённый метод поиска р-медиан *полным перебором* всех р-подмножеств;
- разработать алгоритм *отсечения* заведомо тупиковых ветвей, и тем ускорить перебор.

22.5. Принцип перебора комбинаций

Принцип перебора без повторений всех комбинаций из N вершин по P показан на рис. 22-3, где воображаемый лыжник стартует с одной из вершин верхнего (нулевого) уровня трассы, и спускается в коридоре, ширина которого определена выражением:

$$NN = N + 1 - P$$

Здесь N — число вершин графа, а P — кратность формируемого подмножества, ей соответствует глубина спуска. Из формулы следует, что чем глубже спуск, тем уже коридор для манёвра.

Лыжник маневрирует в направлении стрелок: при переходе на следующий уровень он может выбрать лишь один из узлов, находящийся правее от него, но в пределах заданного коридора. По ходу спуска посещённые вершины накапливаются в очередном p-подмножестве. Накопление завершается на уровне $Level = p-1$ (здесь нумерация уровней $Level$ идёт с нуля), — на этом уровне подмножество из P элементов будет сформировано.

Отметим два крайних случая. При $P=1$ ширина коридора будет максимальна и составляет N, а глубина спуска минимальна, — тут алгоритм вырождается в линейный перебор всех вершин на верхнем уровне. Алгоритм вырождается в линейный также и при $P=N$: здесь коридор сужается до единицы ($NN=1$), но зато лыжник достигает самого «подножья горы».

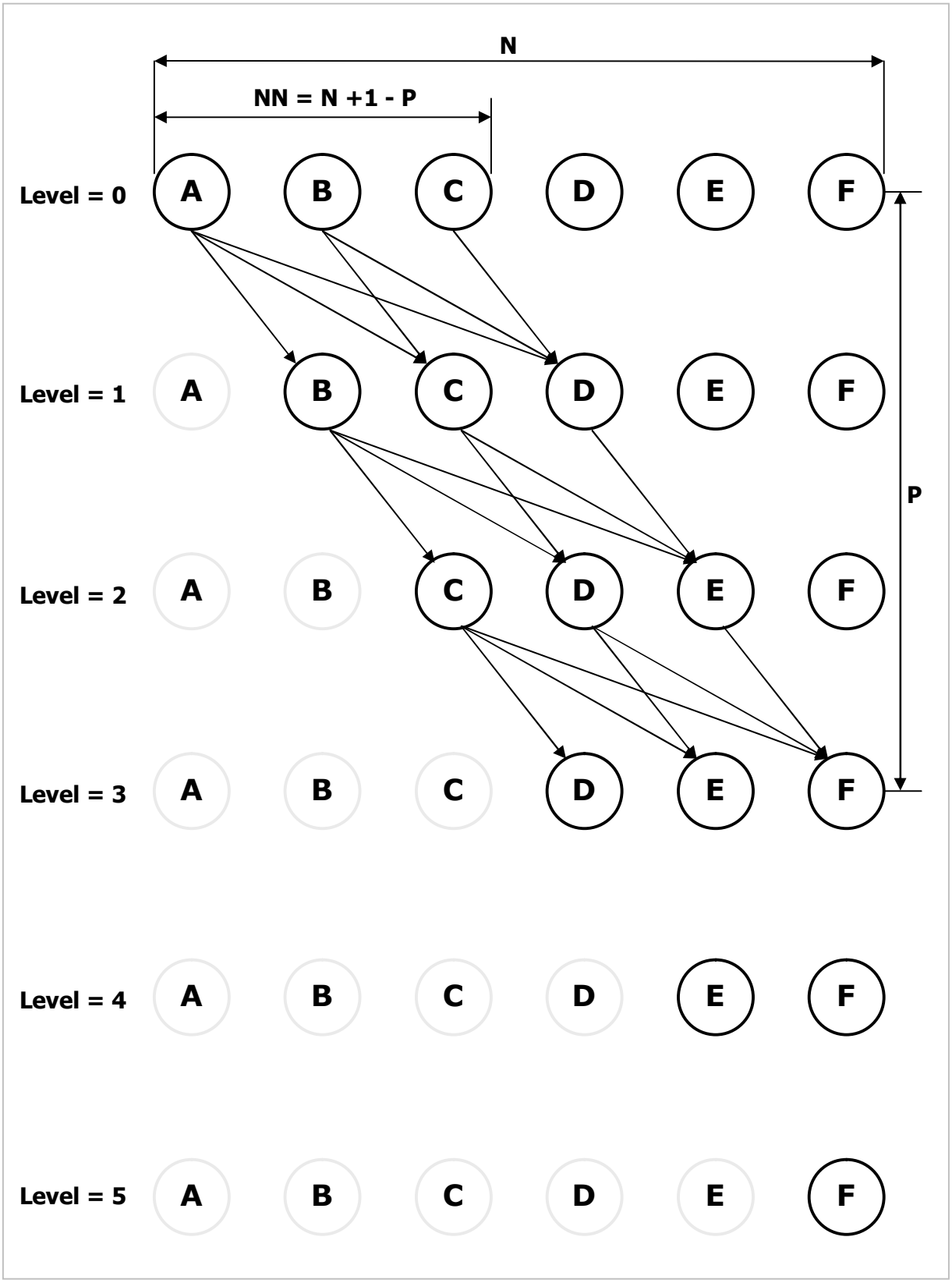


Рис. 22-3 — Порядок обработки вершин при формировании всех подмножеств из шести вершин по четыре (N=6, P=4)

Алгоритм реализован рекурсивной процедурой, для демонстрации которой служит следующая программа.

Листинг 22-1 — Программа для демонстрации перебора комбинаций

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

// Наследник графа с дополнительным демонстрационным методом

type  TGrTest = class(TGraphChars)
        procedure GenCombin(aPole: integer);
      end;

//////////////////////////////////////////////////////////////
// Метод перебора всех комбинаций по P вершин

procedure TGrTest.GenCombin(aPole: integer);
var
    S: TSet;           // накопитель комбинации
    Level : integer;   // текущий уровень - 1
    NN: integer;       // количество обрабатываемых узлов на уровне
    Count : integer;   // счётчик комбинаций

  // Локальная рекурсивная процедура перебора вершин
  // aIndex - начальный индекс для перебора
  procedure Local(aIndex: integer);
  var i: integer;
      N: TNode;         // текущая вершина
  begin
    for i:= aIndex to NN do begin
      // очередная вершина на уровне:
      N:= mNodes.GetItem(i + Level) as TNode;
      S.Insert(N);         // вставляем в комбинацию
      if Level+1=aPole     // комбинация готова?
        then begin
          S.Expo;          // да, показать
          Inc(Count);      // и нарастить счётчик комбинаций
        end
        else begin
          Inc(Level);      // нет, нарастить уровень
          Local(i);        // и вызвать эту процедуру рекурсивно
          Dec(Level);      // возврат на текущий уровень
        end;
      S.Delete(N);         // удалить вершину из комбинации
    end;
  end;

begin
  S:= CreateSet;    // множество для формирования комбинаций
  Count:=0;         // счётчик комбинаций
  Level:= 0;        // текущий уровень
  // Количество узлов, обрабатываемых на каждом уровне:
  NN:= mNodes.GetCount + 1 - aPole;
  Local(1);         // вызов локальной процедуры перебора
  Writeln('Count= ', Count);
  S.Free;
end;
```

```
var Gr: TGrTest;
    P : integer;

begin
    Gr:= TGrTest.GenRandom(false, 10, 10, 7, 12);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - -');
    repeat
      Write('P= '); Readln(P);
      if P=0 then Break;
      Gr.GenCombin(P);
      Writeln('- - - - - - - - - - - - - - - - - -');
    until false;
end.
```

Чтобы не изменять основной объект (**TGraph**), здесь создан наследник графа с методом **GenCombin**, перебирающим все подмножества из *P* вершин. В счётчике **Count** подсчитывается общее количество комбинаций. Представленный демонстрационный метод будет основой для «боевых» методов поиска медиан.

22.6. Вычисление стоимости

Прежде, чем рассмотреть функцию вычисления стоимости p-подмножества, условимся о передаче в неё данных. Вместо передачи подмножества в явном виде, будем окрашивать *p* вершин графа чёрным цветом, отмечая тем самым места размещения служб. Разумеется, что прочие вершины в этот момент будут окрашены как-то иначе, и в целом картина раскраски графа будет меняться по ходу решения задачи. Для этой цели понадобятся три краски, определённые константами, как описано в табл. 22-6:

Табл. 22-6 — Цвета и состояния вершин в ходе решения задачи

Цвет вершины		Состояние вершины
CWhite	Белая	Вершина ещё не опробована
CBlack	Чёрная	Вершина назначена для размещения службы (претендует на роль медианной)
CGray	Серая	Вершина уже опробована в качестве медианной, и на данном уровне обработки уже не будет назначена чёрной

Процесс окраски будет рассмотрен ниже, пока достаточно знать, что в момент вызова функции вычисления стоимости вершины, назначенные для размещения служб, окрашены чёрным. Тогда в ходе подсчёта стоимости текущего «чёрного подмножества» достаточно перебрать нечёрные вершины и определить расстояния каждой из них до *ближайшей* чёрной. Здесь вновь пригодится карта для быстрого поиска маршрутов. Напомню, что эта распределённая по вершинам карта строилась при поиске центров и p-центров процедурой **InitMap**. В каждой вершине графа она создаёт список дальних указателей типа **TFarLink**, где поля **mNodeFar** и **mDist** содержат соответственно ссылку на конечную вершину и расстояние между нею и данной вершиной:

```
TFarLink = class(TItem)    // Дальний указатель
    mNodeFar : TNode;      // целевая вершина
    mNodeNear: TNode;      // ближайшая вершина на пути к целевой
    mDist    : integer;    // расстояние от текущей к целевой
    . . .
  end;
```

За подробностями построения карты дальних указателей отсылаю к главам 15 и 20. Отметим, что элементы **TFarLink** упорядочены в списках по неубыванию расстояний **mDist**, а это важно для ускорения алгоритма. Считаем, что в момент вызова метода **MedianCost** карта дальних указателей уже построена.

Листинг 22-2 — Вычисление стоимости p-подмножества

```
// Вычисление стоимости p-подмножества
// (вершины в этот момент уже раскрашены)
// Цвета вершин означают:
// CWhite -- белые ещё не опробованы
// CBlack -- чёрные пробуются в качестве медианных
// CGray  -- серые уже побывали чёрными и не могут быть медианными

function TGraph.MedianCost: integer;
var Node: TNode;        // текущая вершина
    Dist: integer;      // расстояние к ближайшей чёрной
  //.....................................................
  // Возвращает расстояние к ближайшей чёрной вершине
  function FindMinDist: integer;
  var FL: TFarLink;      // элемент дальней связи
  begin
    Result:= CInfinity;
    with Node.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if FL.mNodeFar <> Node then begin
          if FL.mNodeFar.mColor = CBlack then begin
            Result:= FL.mDist;
            Break;
          end;
        end;
        FL:= GetNext as TFarLink;
      end;
    end; // with
  end;
  //.....................................................
begin  { MedianCost }

  // Перебираем вершины графа (списки расстояний),
  // (вклад в стоимость дают только белые и серые вершины)
  Result:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mColor <> CBlack then begin
      // чёрные вершины не дают вклад в стоимость
      Dist:= FindMinDist;  // минимальное расстояние для Node
      Inc(Result, Node.mValue * Dist);   // накопление результата
    end;
    Node:= NodeNext;
  end;
end;
```

22.7. Упрощённый перебор

Теперь построим упрощённый алгоритм, перебирающий все комбинации по *p* вершин из **N**. Он раскрасит граф, подсчитает стоимость каждой комбинации, и сохранит самую «дешёвую». По ходу перебора комбинаций цвет каждой вершины графа будет меняться от белого сначала к чёрному, а затем к серому так, как показано на рис. 22-4.

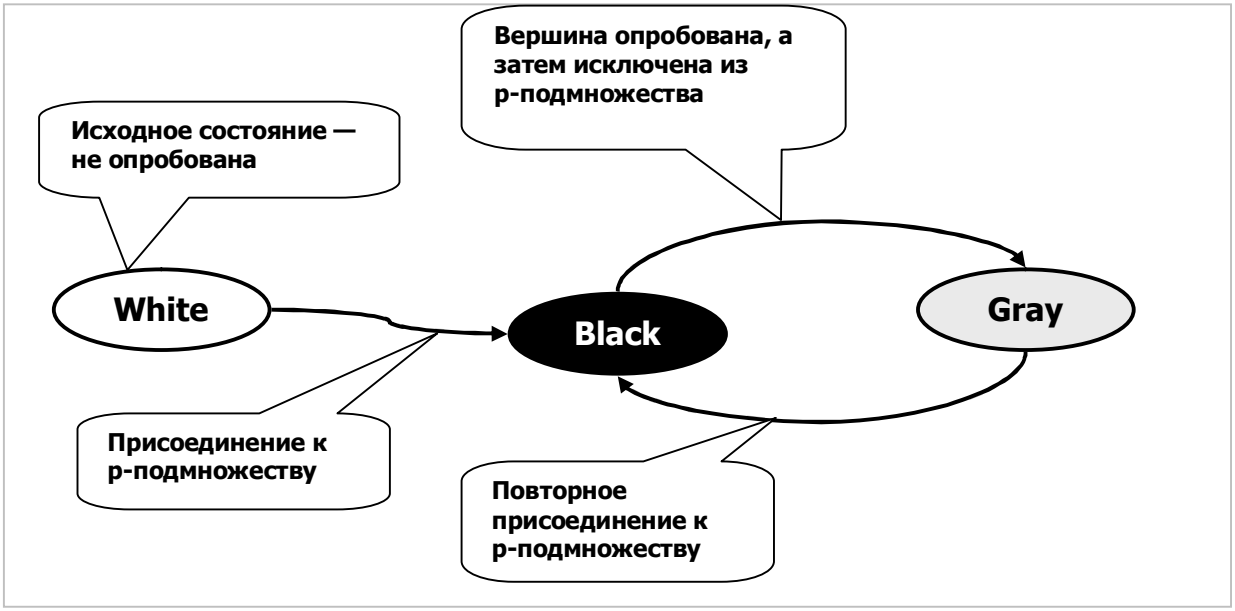


Рис. 22-4 — Диаграмма изменения цвета вершин в упрощённом алгоритме перебора

Изначально все вершины белые. По ходу перебора те из них, что неявно включаются в *p*-подмножество, окрашиваются чёрным. Опробовав очередное *p*-подмножество, удалим из него последнюю из добавленных вершин — окрасим её серым, — и заменим её следующей вершиной на этом же уровне перебора. После обработки всех вершин текущего уровня все они станут серыми, но при повторном входе на этот уровень (из другой вершины) некоторые из них снова будут опробованы, для чего временно окрашены чёрным.

Листинг 22-3 — Упрощённый поиск p-медианы (a-версия)

```
// Поиск медианы "в лоб" перебором
// всех комбинаций медианных подмножеств
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian_A (
                     aPoly: integer;      // кратность P
                     aDirect: TCenter;    // внешняя, внутр., внешне-внутр.
                     var aCost: integer   // стоимость (результат)
                    ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
  // Рекурсивный поиск перебором всех возможных
  // комбинаций чёрных вершин
  // (кандидатов в медианное множество)

  procedure Searching(aIndex: integer { стартовый индекс } );
  var i: integer;
      N: TNode;         // текущий узел
      Cost: integer;    // текущая стоимость p-подмножества
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже текущей, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут,
        // погружаемся на следующий уровень
        Inc(Level);
        Searching(i);      // рекурсивный вызов следующего уровня
        Dec(Level);
      end;
      N.mColor:= CGray;    // проверенной вершине назначаем серый цвет
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin { TGraph.GenPMedian_A }
  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
  // Сформировать списки ближайших вершин (карту дальних указателей):
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;                      // текущий уровень в дереве поиска
  NN:= mNodes.GetCount+1-aPoly; // количество узлов, обраб. на каждом уровне
  Searching(1);          // начать перебор с первой вершины
  DoneMap;               // удалить множества дальних указателей
  aCost:= BestCost;      // вернуть цену
end;
```

22.8. Перебор с досрочным возвратом

Здесь задача поиска медиан, казалось бы, решена (опытную проверку доверяю читателю). Однако при значениях *P* близких к *N/2* её сложность становится экспоненциальной, что снижает практическую ценность алгоритма. Поищем способ отсечь тупиковые ветви перебора. Пусть воображаемый лыжник, достигнув некоторого промежуточного узла перебора, хотя бы грубо, оценит минимальную

стоимость лучшего из p–подмножеств, которые возможно сформировать на последующих уровнях перебора. Если такая оптимистичная оценка окажется *выше*, чем текущая оценка лучшего из уже найденных p–подмножеств, то дальнейший спуск из узла теряет смысл, и выполнять его не следует. Иначе говоря, существующие операторы:

```
// Здесь размер p-подмножества не достигнут,
// погружаемся на следующий уровень
Inc(Level);      // Level+1
Searching(i);    // рекурсивный вызов следующего уровня
Dec(Level);      // Level-1
```

надо заключить внутрь условия:

```
// Здесь размер p-подмножества не достигнут.
// Сравним оценку минимальной стоимости и лучшей текущей.
// Если она меньше текущей, то "ныряем" на уровень вниз
if  EvalBest(BestCost) then begin
  // погружаемся на следующий уровень
  Inc(Level);     // Level+1
  Searching(i);   // рекурсивный вызов следующего уровня
  Dec(Level);     // Level-1
end;
```

где **EvalBest** — некая булева функция, сравнивающая оптимистичную оценку с текущей лучшей. Рассмотрим принцип работы этой функции.

Исходим из того, что в промежуточном узле «горнолыжного» спуска чёрное подмножество сформировано лишь частично, и количество чёрных вершин B ещё не достигло P (здесь $B < P$). В этот момент все вершины графа разбиты на три подмножества: W — белые, B — чёрные и G — серые, а сумма мощностей этих подмножеств равна количеству вершин графа N:

$$W + B + G = N$$

Здесь оптимистичная оценка может быть получена *прикреплением* белых и серых вершин не только к ближайшим *чёрным* (как в точном подсчёте стоимости), но и к ближайшим *белым* вершинам, если в списке дальних указателей **mFarLinks** они окажутся ближе чёрных. Это оправдано тем, что некоторые белые вершины, количество которых равно ($P - B$), в последующем станут чёрными, так почему бы не испытать их заранее? Какие именно белые вершины почернеют, в этот момент ещё не известно. Но поскольку ищется не точная, а *оптимистичная*, заведомо *заниженная* оценка стоимости, то это обстоятельство не существенно.

В ходе суммирования вклад серых вершин накапливается сразу, а подсчёт вклада белых откладывается. Дело в том, что к сумме надо прибавить вклад не всех, а только ($W - P + B$) белых вершин, вносящих *наименьший* вклад в стоимость. Поэтому элементы, хранящие эти вклады, предварительно сохраняются в промежуточном множестве в порядке неубывания их стоимости. Затем, после

обработки всех вершин, вклады первых ($W - P + B$) белых вершин, взятые из промежуточного множества, добавляются к сумме оценки. Так вычисляется оптимистичная оценка стоимости потенциально возможного и пока не сформированного p-подмножества.

Для корректного подсчёта необходимо временно изменить раскраску вершин, и потому перед возвратом на предыдущий уровень уже опробованные на данном уровне серые вершины следует вновь обелить, сделав их потенциальными кандидатами в чёрные. Скорректированная схема изменения состояния вершин приобретает вид, показанный на рис. 22-5:

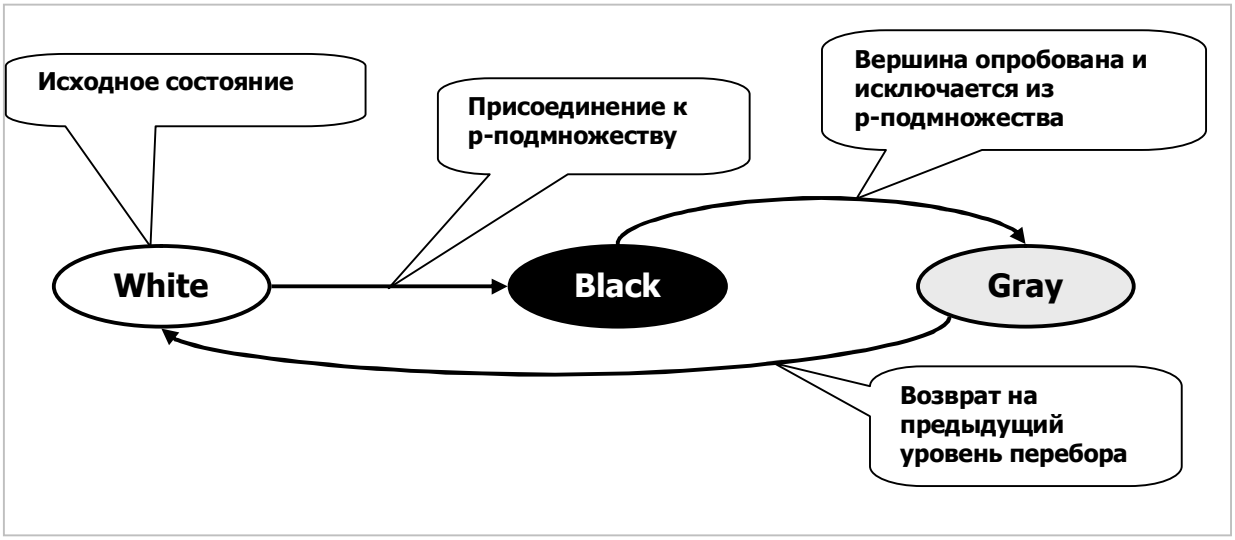


Рис. 22-5 — Диаграмма изменения цвета вершин в усовершенствованном алгоритме

Заметим, что в момент добавления очередной чёрной вершины X и вызова оценочной функции **EvalBest** все вершины, предшествующие X, будут либо серыми, либо чёрными (порядок перебора вершин фиксирован), а все последующие вершины — белыми.

Второе замечание касается параметра, передаваемого в функцию **EvalBest**, — это текущая лучшая оценка. Она пригодится для досрочного прекращения суммирования, когда накопленная сумма достигнет этой оценки. Описанный выше улучшенный метод поиска медиан представлен в следующем листинге.

Листинг 22-4 Поиск медиан частичным перебором p-подмножеств

```
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian(aPoly: integer;   // кратность P
                           aDirect: TCenter;  // внешн., внутр., внешне-внутр.
                           var aCost: integer // стоимость (результат)
                           ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 // Оценка минимальной стоимости текущего распределения вершин
 // Цвета вершин означают:
 // CWhite -- белые ещё не опробованы
 // CBlack -- чёрные пробуются в качестве медианных
 // CGray  -- серые побывали чёрными и не могут быть медианными

 function EvalBest(aBest: integer): boolean;
 var Node: TNode;        // текущая вершина
       FL: TFarLink;     // элемент дальней связи
      Sum: integer;      // накопитель суммы
     Dist: integer;      // расстояние к ближайшей не серой вершине
        S: TSet;         // неубывающее множество элементов FL
     WhiteCnt: integer; // счётчик белых вершин
   //.....................................................
   // Поиск ближайшей не серой вершины
   // Node -- текущий не чёрный узел
   function FindMinDist: integer;
   var FL: TFarLink;      // элемент дальней связи
   begin
     Result:= CInfinity;
     with Node.mFarLinks do begin
       FL:= GetFirst as TFarLink;
       while Assigned(FL) do begin
         if (FL.mNodeFar <> Node) and
            (FL.mNodeFar.mColor <> CGray) then begin
           // Можно прикреплять к чёрным и белым
           Result:= FL.mDist;
           Break;
         end;
         FL:= GetNext as TFarLink;
       end;
     end; // with
   end;
   //.....................................................

 begin  // Eval
   // Перебираем вершины графа (матрицу расстояний),
   // определяем минимальное расстояние к не серым вершинам,
   // и формируем список из неубывающих чисел (расстояний)
   // (вклад в стоимость дают только белые и серые вершины)

   Sum:= 0;  // накопитель суммы
   // начальное значение счётчика суммируемых белых вершин
   WhiteCnt:= mNodes.GetCount - aPoly;
   S:= CreateSet;
   Node:= NodeFirst;
   while Assigned(Node) do begin
     if Node.mColor <> CBlack then begin
       // чёрные не дают вклад в стоимость
       Dist:= FindMinDist;  // минимальное расстояние для Node
       case Node.mColor of
         CWhite: begin
                   // взвешенное расстояние к белой вершине
                   // вставляем в неубывающий список
                   FL:= TFarLink.Create(nil, Node, Node.mValue*Dist);
                   S.Insert(FL);
                 end;
         CGray:  begin
                   // серую суммируем и уменьшаем счётчик белых
                   Inc(Sum, Node.mValue*Dist); // накопление результата
                   Dec(WhiteCnt);     // счётчик суммируемых вершин -1
```

```
                    if Sum>=aBest then Break;
                  end;
          end;
        end;
        Node:= NodeNext;
      end;
      // По окончании формирования множества расстояний
      // добавляем WhiteCnt первых из них к накопленному результату
      FL:= S.GetFirst as TFarLink;
      while (WhiteCnt > 0) and (Sum < aBest) do begin
        Inc(Sum, FL.mDist);           // накопление суммы
        FL:= S.GetNext as TFarLink;   // следующий элемент списка
        Dec(WhiteCnt)
      end;
      // Очистка и удаление временного списка:
      S.ClrAndDestroy;
      S.Free;
      Result:= Sum < aBest;
    end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивный поиск перебором части комбинаций чёрных вершин
  // (кандидатов в медианное множество)

  procedure Searching(aIndex: integer { стартовый индекс } );
  var i: integer;
      N: TNode;         // текущий узел
      Cost: integer;    // текущая стоимость p-подмножества
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже текущей, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут.
        // Сравним оценку минимальной стоимости с лучшей текущей.
        // Если она меньше текущей, то "ныряем" на уровень вниз
        if  EvalBest(BestCost) then begin
          // погружаемся на следующий уровень
          Inc(Level);     // Level+1
          Searching(i);   // рекурсивный вызов следующего уровня
          Dec(Level);     // Level-1
        end;
      end;
      N.mColor:= CGray;  // проверенной вершине назначаем серый цвет
    end;
    // Перед возвратом на уровень вверх возвращаем белый цвет
    // всем обработанным вершинам текущего уровня
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CWhite;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
begin  { TGraph.GenPMedian }
  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
  // Сформировать множества ближайших вершин:
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;                      // текущий уровень в дереве поиска
  NN:= mNodes.GetCount+1-aPoly; // кол-во узлов, обраб. на каждом уровне
  Searching(1);          // начать перебор с первой вершины
  DoneMap;               // удалить множества дальних указателей
  aCost:= BestCost;      // вернуть цену
end;
```

22.9. Испытания

Для испытания метода поиска медиан применена следующая программ:

Листинг 22-5 — Программа для испытания метода поиска медиан

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;           // граф
    Median: TSet;          // медиана
    Cost: integer;         // стоимость медианы
    Direct : TCenter;      // направление медианы
    P : integer;           // кратность
    D : char;              // для ввода данных о направлении
begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - - -');
  Write('Direction (I/O/A) = '); Readln(D);
  case Upcase(D) of
    'I': Direct:= InCenter;
    'O': Direct:= OutCenter;
    else Direct:= InOutCenter;
  end;
  repeat
    Write('P= '); Readln(P);
    if P=0 then Break;
    Median:= Gr.GenPMedian (P, Direct, Cost) ;
    Median.Expo;
    Median.Free;
```

```
    Writeln('Cost  = ', Cost:7);
  until false;
end.
```

«Подопытный» граф представлен следующим текстом и показан на рис. 22-6, а результаты поиска его p-медиан даны в табл. 22-7.

```
Test.txt - орграф с взвешенными дугами и вершинами
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

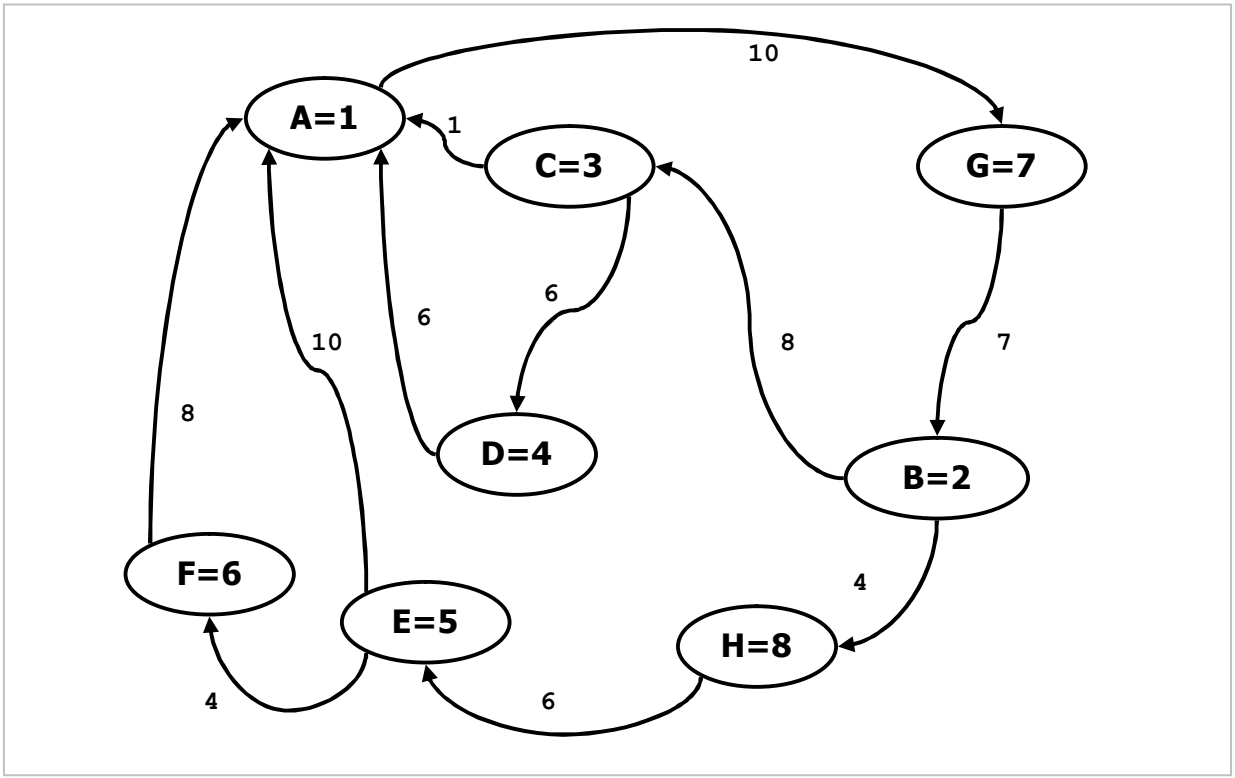


Рис. 22-6 — Орграф с взвешенными дугами и вершинами

Табл. 22-7 — Р-медианы графа, показанного на рис. 22-6

Кратность Р	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	Стоимость	Вершины	Стоимость	Вершины	Стоимость
1	B	388	A	383	G	1019
2	C, H	228	A, H	210	G, H	723
3	C, G, H	129	A, F, H	132	F, G, H	489
4	C, E, G, H	63	A, F, G, H	55	E, F, G, H	304
5	C, D, E, G, H	39	A, D, F, G, H	31	D, E, F, G, H	156

Следующая программа сравнивает быстродействие полного и частичного перебора:

```
{$APPTYPE CONSOLE}
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;            // граф
    Median: TSet;           // медиана
    Cost: integer;          // стоимость медианы
    Direct : TCenter;       // направление медианы
    P : integer;            // кратность
    D : char;               // для ввода данных о направлении

procedure TestTime;
const C1 =20; C2 =5; // количество повторений
var i: integer;
    Start, Time : TDateTime;
begin

  case Direct of
    InCenter: Writeln('------------------- IN -------------------');
    OutCenter: Writeln('------------------- OUT -------------------');
    InOutCenter: Writeln('------------------- IN-OUT -------------------');
  end;
  Start:= Now;
  for i:= 1 to C1 do begin
    Median:= Gr.GenPMedian (P, Direct, Cost);
    if i < C1 then Median.Free;
  end;
  Time:= MilliSecondsBetween(Start, Now) / C1;
  Median.Free;
  Writeln('Cost  = ', Cost:7, '  Time  = ', Time:7:2);

  Start:= Now;
  for i:= 1 to C2 do begin
    Median:= Gr.GenPMedian_A (P, Direct, Cost);
    if i < C2 then Median.Free;
  end;
  Time:= MilliSecondsBetween(Start, Now) / C2;
```

```
  Median.Free;
  Writeln('Cost_A= ', Cost:7, '  Time_A= ', Time:7:2);
end;

begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - -');
  Write('Direction (I/O/A) = '); Readln(D);
  case Upcase(D) of
    'I': Direct:= InCenter;
    'O': Direct:= OutCenter;
    else Direct:= InOutCenter;
  end;
  repeat
    Write('P= '); Readln(P);
    if P=0 then Break;
    TestTime;
    Writeln;
  until false;
end.
```

Для испытания взят граф из 24 вершин, его текстовое представление дано ниже:

```
Test.txt
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
24 - количество вершин
A=9 B=3 C=3 D=7 E=4 F=2 G=4 H=5 I=1 J=5 K=1 L=9 M=1 N=3 O=10 P=4 Q=8 R=4 S=7
T=9 U=8 V=4 W=2 X=4
A -> E=4 H=10 I=8 O=8 T=2
B -> C=8 E=7 H=2 L=9 R=1
C -> B=8 D=1 H=8 J=1 O=7 Q=5
D -> C=1 E=10 F=10 G=6 I=10 L=9 M=1 O=8 R=3 X=1
E -> A=4 B=7 D=10 G=6 J=9 K=7 N=7 S=10 X=2
F -> D=10 K=2 L=9 N=3 X=7
G -> D=6 E=6 H=8 I=6 L=2 P=9 Q=1 U=8 W=8
H -> A=10 B=2 C=8 G=8 M=5 P=7 S=8 V=5
I -> A=8 D=10 G=6 J=1 O=8 T=9 U=4 V=4
J -> C=1 E=9 I=1 T=7
K -> E=7 F=2 M=5 R=5
L -> B=9 D=9 F=9 G=2 R=7 V=6 X=9
M -> D=1 H=5 K=5 Q=1 W=4
N -> E=7 F=3 O=9 Q=1 R=5 U=7 W=7
O -> A=8 C=7 D=8 I=8 N=9 S=9 U=3 W=7
P -> G=9 H=7 S=8
Q -> C=5 G=1 M=1 N=1 T=6 X=7
R -> B=1 D=3 K=5 L=7 N=5 V=5
S -> E=10 H=8 O=9 P=8 U=10 X=8
T -> A=2 I=9 J=7 Q=6
U -> G=8 I=4 N=7 O=3 S=10 W=7
V -> H=5 I=4 L=6 R=5
W -> G=8 M=4 N=7 O=7 U=7
X -> D=1 E=2 F=7 L=9 Q=7 S=8
```

Результаты сравнения двух методов даны в табл. 22-8 и представлены на графике рис. 22-7.

Табл. 22-8 — Время поиска p-медиан
частичным (T1) и полным (T2) перебором, мс

Кратность медианы P	Частичный перебор (T1)	Полный перебор (T2)	Кратность медианы P	Частичный перебор (T1)	Полный перебор (T2)
1	5,5	6,2	13	18,8	5534,0
2	6,2	6,0	14	12,5	4260,0
3	14,0	15,6	15	8,6	2838,0
4	39,0	43,8	16	7,0	1597,0
5	57,0	147,0	17	7,0	762,0
6	65,0	419,0	18	6,2	312,0
7	69,0	988,0	19	6,2	106,0
8	62,0	1975,0	20	6,2	34,2
9	55,0	3344,0	21	6,2	12,2
10	40,0	4810,0	22	5,5	6,2
11	30,0	6010,0	23	5,5	6,2
12	23,5	6225,0			

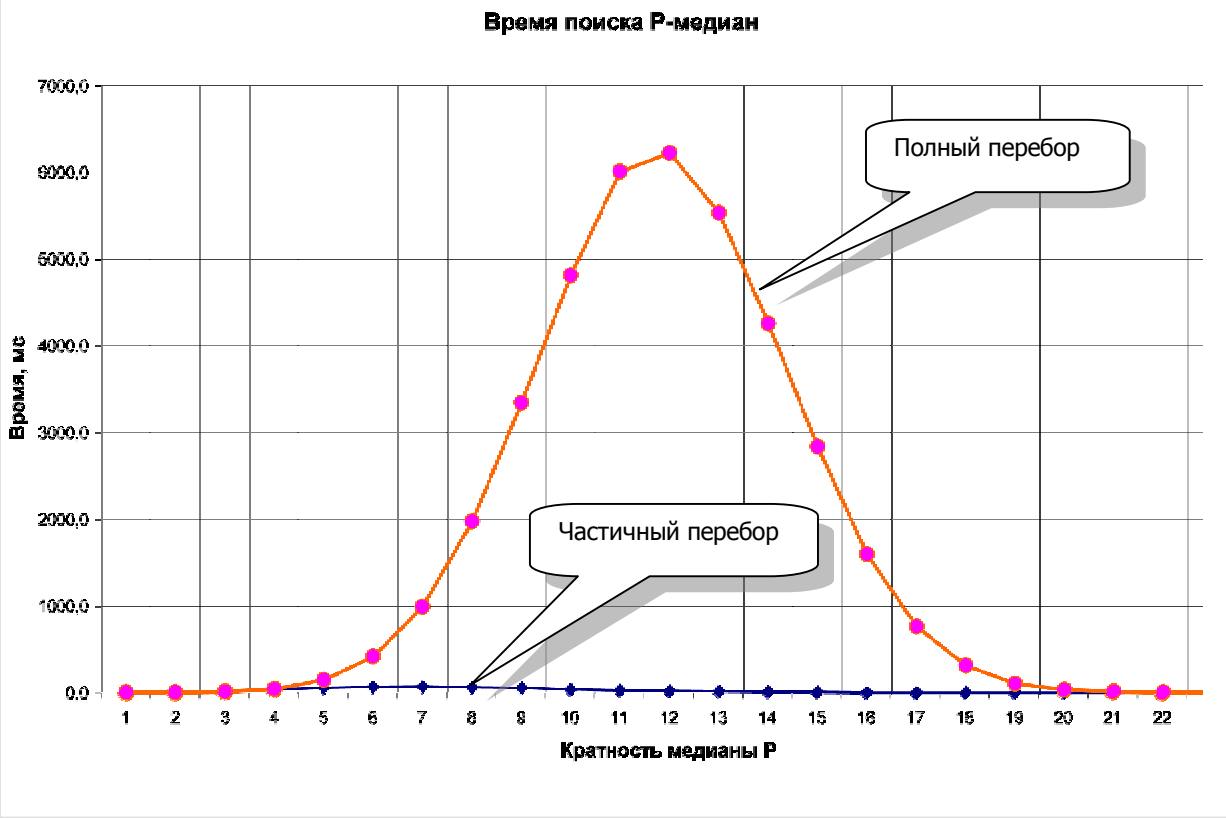


Рис. 22-7 — Время поиска p-медиан в зависимости от кратности P

Как и следовало ожидать, время поиска р-медиан полным перебором подчинено «колокольному» биномиальному закону и многократно возрастает с приближением *P* к значениям, близким к *N/2*. В переборе с возвратом эта зависимость выражена слабо, и в целом этот более сложный алгоритм оправдывает себя.

22.10. Итоги

- Медианой графа называют вершину, для которой сумма взвешенных расстояний между нею и другими вершинами минимальна. Для ориентированных графов различают три рода медиан: внешнюю, внутреннюю и внешне-внутреннюю.
- Кратной Р-медианой графа называют подмножество из Р вершин, для которого сумма взвешенных расстояний между этими вершинами и прочими вершинами графа минимальна. В орграфах различают три рода Р-медиан: внешнюю, внутреннюю и внешне-внутреннюю.
- Поиск медиан и Р-медиан выполняется одним алгоритмом, упрощённая версия которого состоит в полном переборе всех возможных р–подмножеств и выборе самого «дешёвого».
- Перебор всех р–подмножеств при значениях Р, близких к N/2 приобретает экспоненциальную сложность, и потому непригоден для больших значений N (от нескольких десятков и более).
- Для отсечения заведомо тупиковых ветвей с целью ускорения перебора выполняют предварительную оценку нижней границы стоимости потенциальных р–подмножеств в промежуточных узлах перебора. Это на несколько порядков ускоряет поиск.

22.11. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 23
Остовные деревья

23.1. Неориентированные деревья

Однажды у администрации известного читателям архипелага появились, наконец-то, средства на реконструкцию обветшалых мостов. Из экономии новые двунаправленные мосты построили на месте старых так, что при минимальной их общей длине сохранилась взаимная доступность островов. Старые мосты в целях безопасности демонтировали. Новая мостовая схема образовала *остов* или *покрывающее дерево* графа. На рис. 23-1 представлен минимальный по стоимости остов архипелага, удалённые мосты показаны пунктиром. Данное остовное дерево построено на рёбрах, оно не ориентировано.

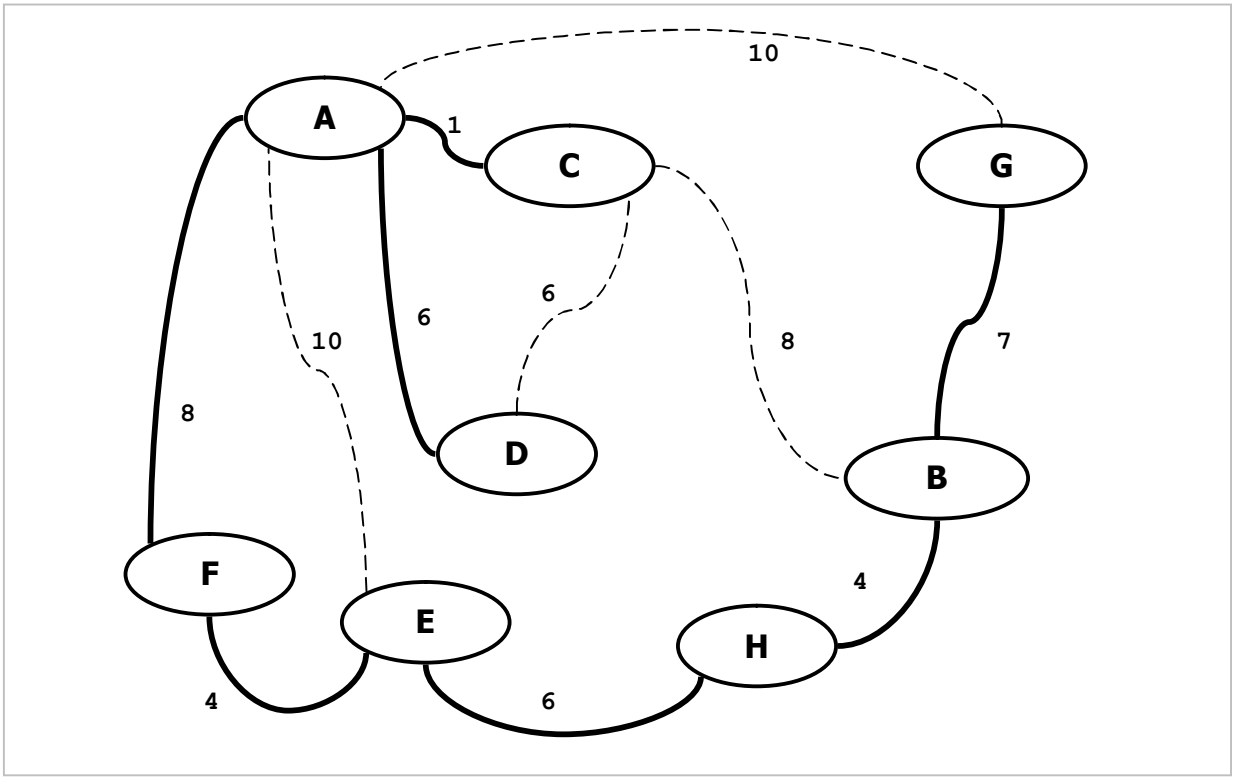


Рис. 23-1 — Один из двух минимальных остовов графа

Поскольку алгоритмы построения остовных деревьев используют свойства *дерева*, дадим три эквивалентных его определения (любые два из них следуют из третьего):

- дерево — это связанный граф, состоящий из N вершин и $N-1$ рёбер;
- дерево — это связанный граф, не имеющий циклов;
- дерево — это граф, в котором каждая пара вершин связана одной и только одной простой цепью.

В самом деле, будь в мостовой схеме цикл, некоторые острова оказались бы связанными двумя цепочками, и часть мостов была бы избыточной, — такой граф не будет деревом.

Наряду с *минимальным* остовом, на графе можно построить и другие деревья, например, остов *максимального* веса. При решении некоторых проблем необходимо перебрать *все* или *часть* остовных деревьев графа, поэтому ниже рассмотрим две задачи:

- построение остовов *минимального* или *максимального* веса;
- перебор *всех* остовных деревьев графа.

23.2. Минимальный (максимальный) остов — алгоритм Прима

Принцип построения *минимального* или *максимального* остова крайне прост, и основан на равноправии вершин неориентированного графа. Ввиду равноправия вершин, постройку дерева можно начать с любой из них, например, с первой по порядку. Взяв её и окрасив чёрным (изначально все вершины белые), тем самым пометим её, как обработанную. Далее находим среди белых вершин *ближайшую* к ней, пусть это будет вершина *X*, красим её чёрным, а ребро *A-X* включаем в состав дерева. На следующем шаге среди белых вершин ищем ближайшую к «чёрному» подмножеству, то есть, либо к вершине *A*, либо к вершине *X*. Пусть это будет вершина *Y*. Тогда *Y* очерняем, а кратчайшее из рёбер (*A-Y* или *X-Y*) присоединяем к дереву. Этот процесс продолжаем вплоть до почернения всех вершин графа. Максимальный остов строится аналогично, но вместо *кратчайших* выбираются *длиннейшие* рёбра.

Представленный тут алгоритм *Прима* слегка похож на расширение гаммы в алгоритме Дейкстры для поиска кратчайших путей. Но если там расширяется чёрное подмножество включением всех соседей вершины, то здесь на каждом шаге присоединяется только одна вершина и ребро. Для ускорения алгоритма следует позаботиться о быстром доступе к ближайшему белому соседу, для чего создадим особый объект. Он нужен ещё и потому, что в неориентированном графе рёбра представлены парами встречно направленных линков **TLink**. Однако для нужд перебора каждую такую пару желательно представить одним объектом. Итак, на основе класса **TLink** сконструируем вспомогательный линк **TTreeLink**:

```
TTreeLink = class (TLink)
  mMaxTree: boolean;  // false -- минимальный, true - максимальный остов
  constructor Create(aLink: TLink);
  function  Compare(arg: TItem): TCompare; override;
  procedure Print(var aFile: TextFile); override;
end;
```

Эти вспомогательные линки будут соответствовать рёбрам графа, но не будут включены в его структуру. Метод сравнения **Compare** обеспечит линкам нужный порядок сортировки при вставке их в приоритетную очередь: либо по не

убыванию, либо по не увеличению длины, а также отсечение возможных дубликатов.

Листинг 23-1 — Метод сравнения линков, устраняющий дубликаты

```
function TTreeLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  // Для неориентированного графа сравниваем исходную и конечную вершины
  // на предмет встречных линков
  if not mOwner.mOwner.mDirect and  // если не орграф
    (mOwner=(arg as TTreeLink).mDest) and (mDest=(arg as TTreeLink).mOwner)
    then begin
      // Здесь линки направлены встречно, отвергаем дубликат:
      Result:= cmpEq;
      Exit;
    end;
  // Если линки не совпадают, то сортируем по неубыванию веса (длины)
  Result:= cmpLess;
  if mValue > (arg as TTreeLink).mValue then Result:= cmpGreate
  // Для построения дерева максимального веса результат инвертируем
  if mMaxTree
    then if Result=cmpLess then Result:= cmpGreate else Result:= cmpLess;
 end;
```

Напомню, что когда метод **Compare** возвращает значение **cmpEq**, метод **TSet.Insert** не вставляет элемент в множество, и возвращает **false**.

Порождение вспомогательных линков вершины поручим методу **TNode.AddTreeLinks**, куда множество-накопитель линков (приоритетная очередь) передаётся через параметр:

Листинг 23-2 — Добавление линков, ведущих к ближайшим белым вершинам

```
// aLinks - накопитель линков, выполняющий функцию приоритетной очереди
// Линки выстраиваются по неубыванию (aMaxTree=false)
// или невозрастанию дистанции (aMaxTree=true)

procedure TNode.AddTreeLinks(aMaxTree: boolean; aLinks: TSet);
var  L : TLink;
    TL : TTreeLink;
begin
  // Обработка исходящих связей
  L:= OutLinkFirst;
  while Assigned(L) do begin
    // Вставляем только линки, ведущие к белым вершинам
    if (L.mDest.mColor = CWhite) then begin
      TL:= TTreeLink.Create(L, aMaxTree);
      if not aLinks.Insert(TL) then TL.Free;
    end;
    L:= OutLinkNext;
  end;
end;
```

Здесь, если при попытке вставки дубликата методом **TSet.Insert** возвращается **false**, дубликат уничтожается.

Обсудив создание служебных линков, вставляемых в приоритетную очередь, рассмотрим построение остова минимального или максимального веса алгоритмом Прима.

Листинг 23-3 — Построение минимального остовного дерева методом Прима

```
function TGraph.GenCoverTree(aMaxTree: boolean;  // false= min, true = max
                             var aCost: integer  // возвр. общую стоимость
                            ): TSet;             // множество TTreeLink
var Node: TNode;       // текущая вершина
      TL: TTreeLink;   // текущий линк для дерева
   Links: TSet;        // вспомогательное множество линков
begin
  aCost:=0;               // накопитель суммы
  Result:= CreateSet;     // множество линков для результата
  Links:= CreateSet;      // множество линков для буфера
  ResetNodes;             // очистка для всех вершин mColor = CWhite
  Node:= NodeFirst;       // исходная вершина
  // Присоединение всех вершин графа (линков на единицу меньше)
  while Assigned(Node) and
       (Result.GetCount < mNodes.GetCount-1) do  begin
    Node.mColor:= CBlack;      // присоединяем текущую к множеству чёрных
    // Добавляем в множество Links линки к ближайшим белым вершинам:
    Node.AddTreeLinks(aMaxTree, Links);
    // Перебираем линки Links в поиске ближайшей белой вершины
    TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
    Node:= nil;                         // искомая вершина пока пуста
    while Assigned(TL) do begin
      Links.Delete(TL);                 // удаляем линк из буфера Links
      // Искомая вершина может быть как источником,
      // так и приёмником связи:
      if TL.mOwner.mColor = CWhite
        then Node:= TL.mOwner   // источник связи
      else if TL.mDest.mColor = CWhite
        then Node:= TL.mDest;   // приёмник связи
      if Assigned(Node) then begin
        // Здесь ближайшая белая вершина найдена:
        Result.Insert(TL);      // вставляем линк в результат
        Inc(aCost, TL.mValue);  // наращиваем стоимость
        Break;
      end;
      // здесь линк связывает две чёрные вершины:
      TL.Free;                          // удаляем ненужный линк
      TL:= Links.GetNext as TTreeLink; // и берём следующий
    end;
  end;
  // Если остов не построен, то граф не связан
  if Result.GetCount < mNodes.GetCount-1 then begin
    Result.ClrAndDestroy;
    aCost:=0;
  end;
  // Очистка и удаление вспомогательного множества линков:
  Links.ClrAndDestroy;
  Links.Free;
end;
```

Выбрав очередную вершину и окрасив её чёрным, вставим исходящие из неё линки в приоритетную очередь — множество **Links**. Если строится остов минимального веса, первый линк в очереди будет кратчайшим, а иначе

длиннейшим. Следующий далее вызов **`Links.GetFirst`** выбирает этот кратчайший линк, после чего линк из буфера удаляется. Затем проверяются цвета смежных вершин линка. Если одна из них окажется белой, то линк присоединяется к остову, а белая вершина окрашивается чёрным. Если же линк связывает две чёрные вершины (порождает цикл), он отвергается, а из приоритетной очереди извлекается следующий линк.

На рис. 23-2 — рис. 23-5 показаны состояния вершин и линков в ходе обработки первых четырёх вершин. Обратите внимание: после присоединения вершины *C* в начале приоритетной очереди окажутся два линка одинаковой длины: *A-D* и *C-D*. Если первым из буфера буден извлечён *A-D*, то к моменту обработки линка *C-D* вершина *D* станет чёрной, и тогда линк *C-D* будет отвергнут (рис. 23-5). Конечный результат построения минимального остова этого графа дан на рис. 23-1.

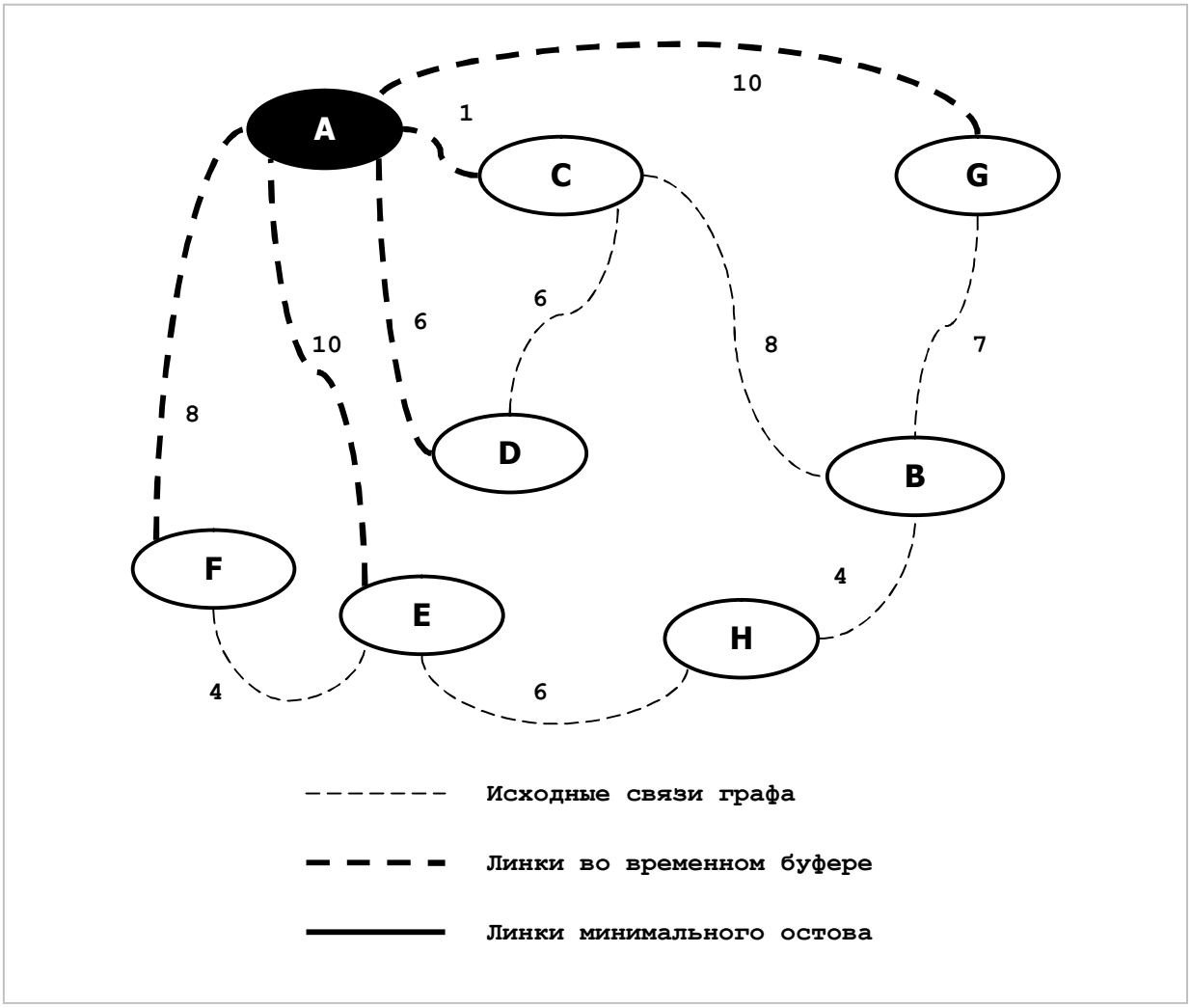


Рис. 23-2— Состояние вершин и линков после обработки 1-й вершины

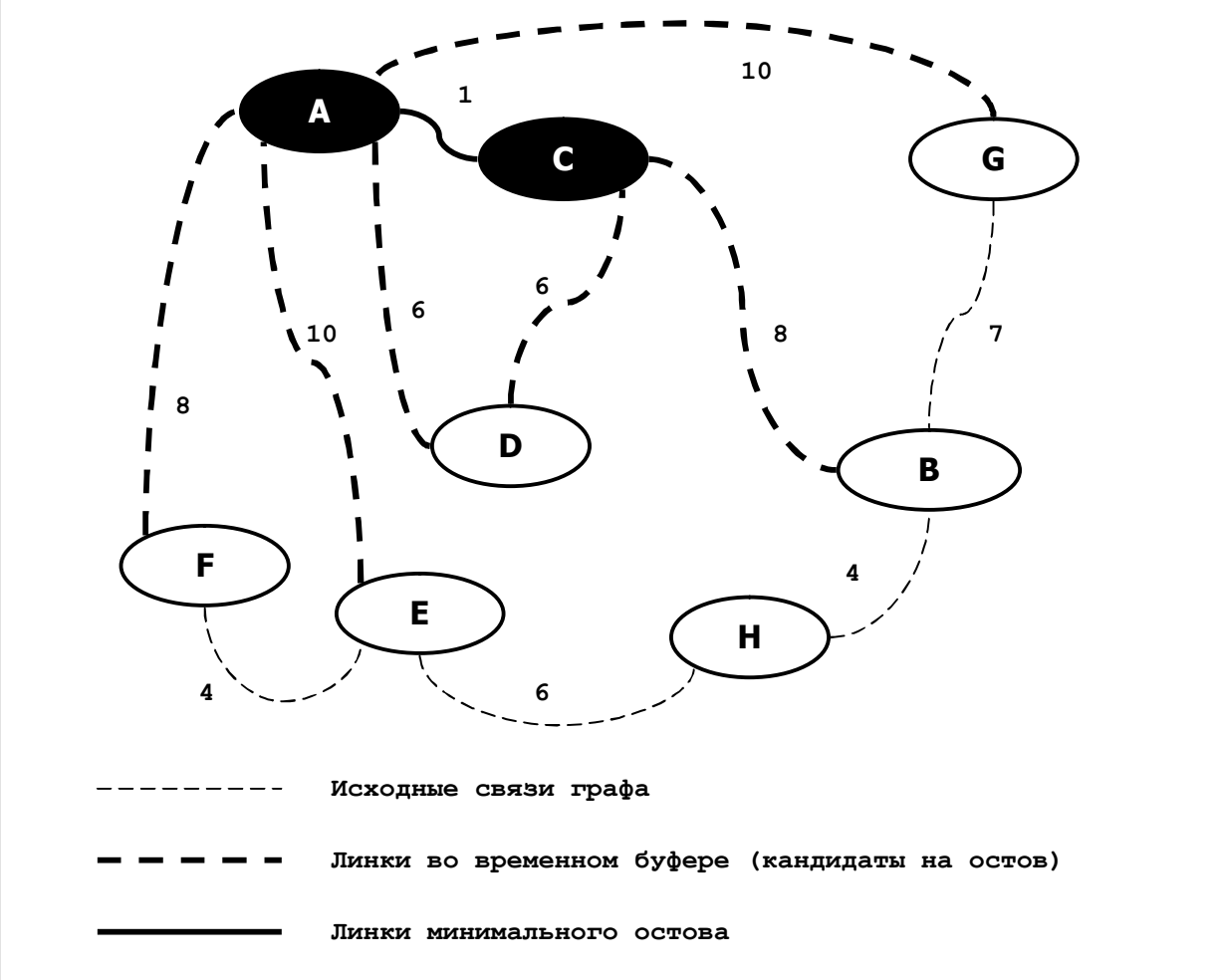


Рис. 23-3 — Состояние вершин и линков после обработки 2-й вершины

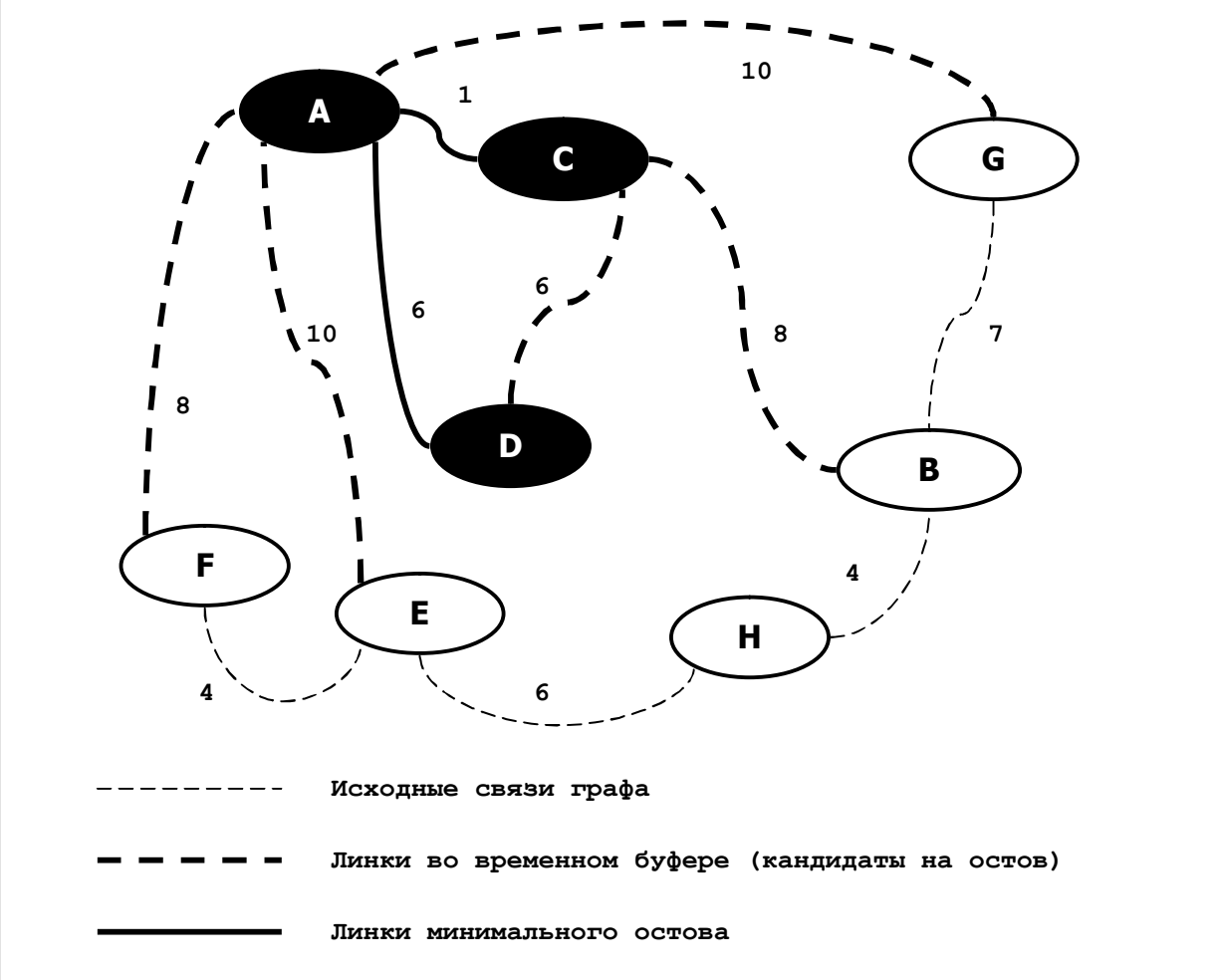


Рис. 23-4— Состояние вершин и линков после обработки 3-й вершины

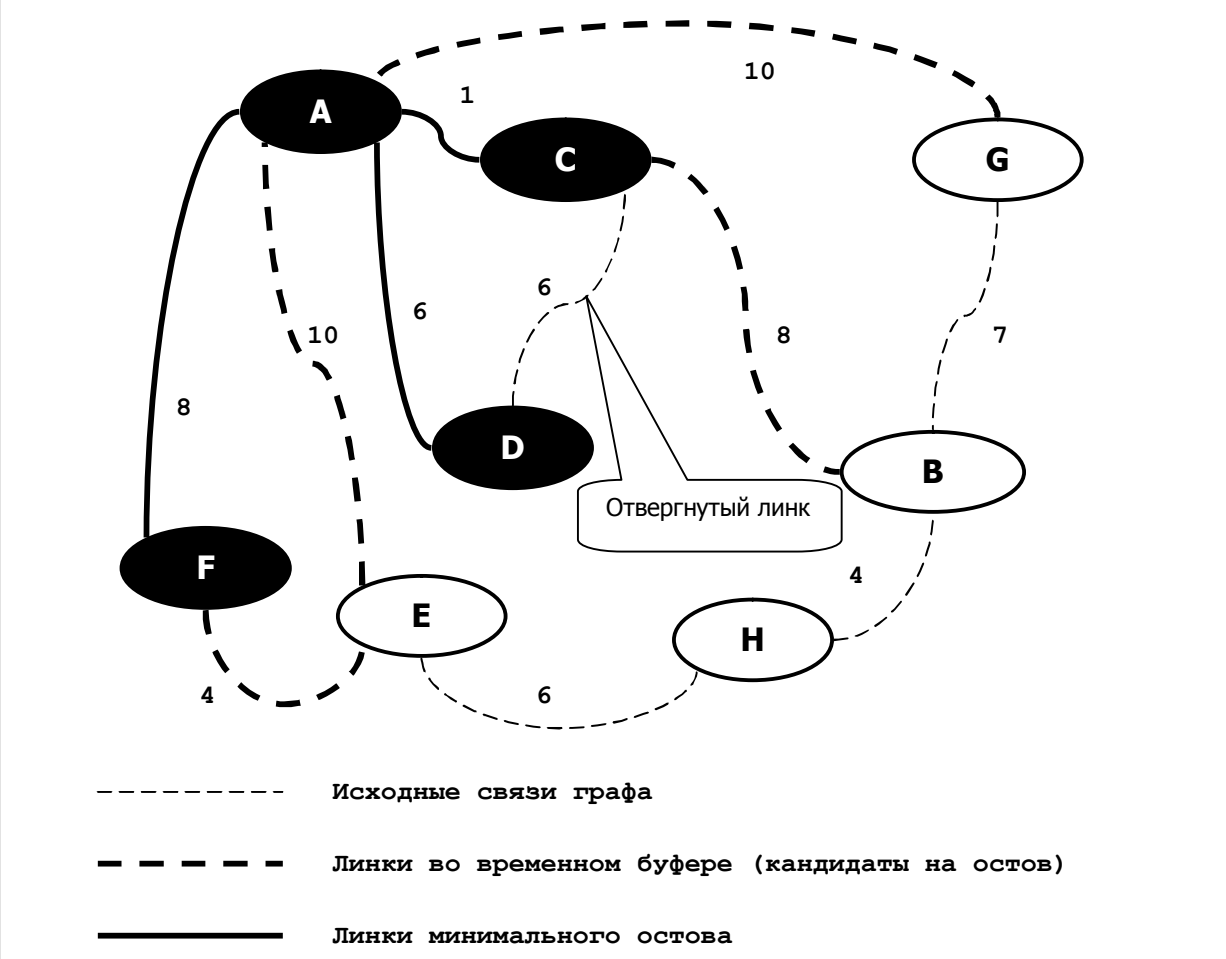


Рис. 23-5 — Состояние вершин и линков после обработки 4-й вершины

23.3. Построение всех остовов неориентированного графа

Некоторые задачи требуют перебора *всех* или *части* остовных деревьев, вот один из примеров. Пусть на острове *X* добывают уголь, а на *Y* — руду, и для нужд металлургии, дабы не гонять порожняк, перевозят уголь и руду между ними встречными составами. Пусть также на острове *Z* (или на нескольких островах) устроены курорты и потому движение товарных составов там нежелательно. Тогда при планировании мостов можно потребовать, чтобы в минимальном остове путь между вершинами *X* и *Y* не пролегал через указанные курортные зоны. Таких задач, где выбор подходящих остовов специфичен, немало, и потому имеет смысл построить метод перебора всех остовов, вынося их обработку за скобки.

23.3.1. Оценка сложности

Когда речь заходит о переборе, первым делом оценивают вычислительную сложность задачи. Поскольку любой остов содержит *N-1* рёбер, где *N* — количество вершин графа, то для насыщенного связями графа количество возможных комбинаций будет расти экспоненциально с ростом *N*. Многие из них не составят деревьев, но и пригодных вариантов будет немало. Доказано, что точное количество остовных деревьев вычисляется через определитель матрицы,

получаемой в результате перемножения матрицы инциденций графа на её транспонированную копию. Почему бы вам не проверить это?

Но мы, как сказал великий полководец, «пришли их бить, а не считать», и потому займёмся не подсчётом, а генерацией всех остовных деревьев графа. Обсуждаемые ниже алгоритмы и процедуры войдут в состав метода **ExpoAllTrees**, формирующего и выводящего на экран все остовы графа.

23.3.2. Инициализация

Поскольку все возможные остовы порождаются перебором комбинаций из *L* связей, вначале позаботимся о «сырье», необходимом для этой работы, то есть подготовим множество вспомогательных линков **TTreeLink**, описанных ранее при построении минимального остова. Эту работу выполним при инициализации, в результате чего будет сформировано множество **Links**, —приоритетная очередь, упорядоченная в порядке не убывания (или не увеличения) длины линков. Напомню, что эти линки соотносятся с рёбрами исходного графа. Здесь же инициируются некоторые поля вершин, о назначении которых будет сказано позднее.

Листинг 23-4 — Инициализация вершин и построение множества линков

```
procedure Local_Init;
var Node: TNode;   // текущая вершина
begin
  ResetNodes;          // для всех вершин mColor=0, mPred=nil
  Tree:= CreateSet;    // множество для остовного дерева
  Links:= CreateSet;   // множество для всех линков
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      mRoot:= Node;                // корень поддерева
      AddTreeLinks(false, Links);  // добавление линков вершины
    end;
    Node:= NodeNext;
  end;
end;
```

23.3.3. Проверка на цикл

Пока множество линков остовного дерева **Tree** ещё пусто, вершины графа можно трактовать как множество изолированных поддеревьев будущего остовного дерева. Другими словами, эти вершины пока логически не связаны, изолированы, их связи в исходном графе тут не в счёт. При построении остова будем постепенно «соединять» эти вершины линками, извлекаемыми из множества **Links**, комбинируя линки всеми возможными способами. Это «соединение» вершин будет воображаемым, виртуальным. Логическую связь между вершинами организуем через поле вершины **mPred**, которое применялось ранее для поиска кратчайших путей, детали обсуждаются ниже.

В ходе сращивания поддеревьев (логического соединения вершин) необходимо проверять очередной извлекаемый из множества линк на предмет того, объединяет ли он *два* поддерева, или связывает две вершины *одного*, замыкая тем самым не нужный здесь цикл. Если обе вершины линка принадлежат *одному* поддереву, линк надо отбросить. В алгоритме Прима для такой проверки вершины окрашивали двумя цветами. Но сейчас соединяемых частей будет много, и потому для проверки на цикл введём в вершину поле **mRoot**, — метку, указывающую на корень поддерева. Этот корень будет един для всех вершин одного поддерева, но в разных поддеревьях корни будут разными. Корнем может быть любая вершина поддерева. После инициализации вершин поле **mRoot** указывает на саму эту вершину.

Для проверки того, что линк не создаёт цикла, необходимо сравнить поля **mRoot** вершины-источника и вершин-приёмника линка:

```
Result:= mOwner.mRoot <> mDest.mRoot; // true, если линк не образует цикла
```

Отметим, что роль *корня* подобна роли вожака в стае животных. Одиночка сам себе вожак, но в паре или в стае главенствует один. После разделения стаи в каждой отделившейся группе неминуемо появится свой вожак (не обязательно бывший). Здесь на «вожака стаи» показывает поле **mRoot** — указатель на одну из вершин поддерева, а иерархия в «стае» поддерживается полем **mPred**, направленным от периферии к корню.

23.3.4. Сращивание деревьев

Напомню, что основу будущего алгоритма составит перебор без повторений комбинаций из *N-1* линков. Перебор выполняется попеременно в двух направлениях. *Спускаясь* по дереву перебора, добавляем линки к текущему множеству, формируя постепенно очередной остов, а возвращаясь *вверх*, откатываемся в предыдущее состояние с тем, чтобы присоединять другие линки. По мере спуска отдельные поддеревья леса *сращиваются*, а при подъёме вновь *расщепляются* путём удаления последних присоединённых к дереву линков. Здесь потребуются две процедуры: для *сращивания*, и для *расщепления* поддеревьев.

Обратимся к рис. 23-6, где показаны два поддерева, уже порождённые на предыдущих этапах спуска.

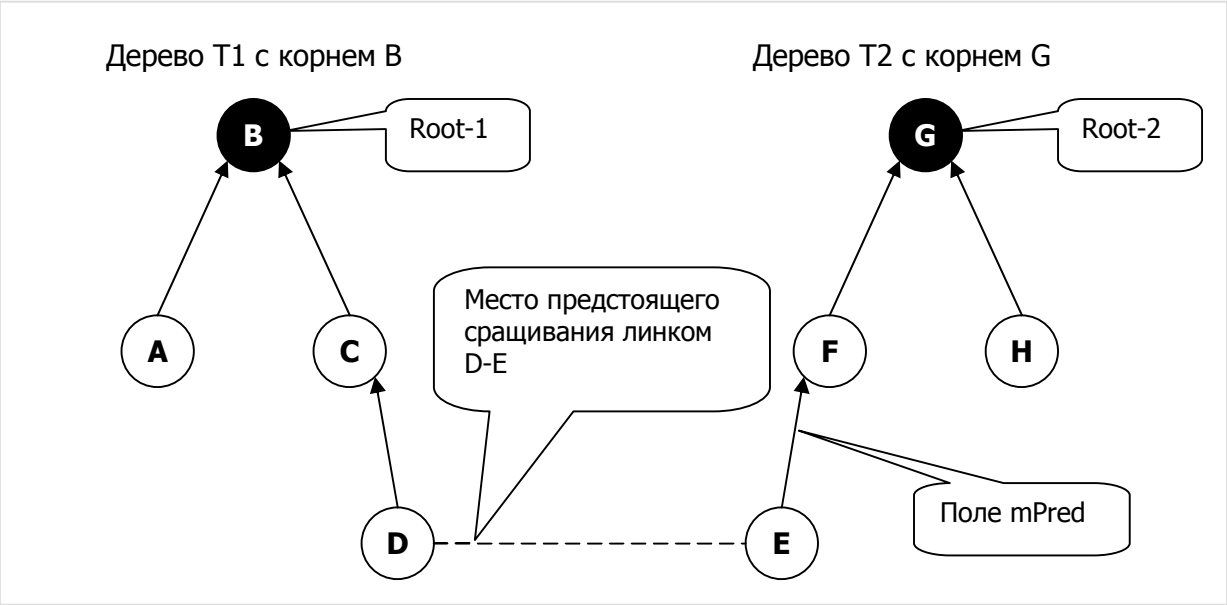


Рис. 23-6 — Исходное состояние перед сращиванием поддеревьев

Стрелки, исходящие из вершин, образованы полями **mPred**, и организованы в ходе предшествующего сращивания. Каждой такой стрелке соответствует линк — ребро исходного графа. Вершины, не содержащие исходящих стрелок, являются корнями поддеревьев. Все вершины в дереве *T1* помечены корнем *B* (поле **mRoot**), а в дереве *T2* помечены корнем *G*.

Пусть в некоторый момент для опробования выбран линк *D-E*. Поскольку он соединяет два разных поддерева, то цикл не создаёт, и две «стаи» могут соединиться. Для соединения надо выбрать общего «вожака» — один из двух корней, а также исправить некоторые стрелки так, чтобы развернуть их к новому корню. Который из двух корней выбрать? Оба они равноправны, но для определённости выберем тот, что является корнем для источника (**mOwner**) обрабатываемого линка. Поскольку источником линка *D-E* является вершина *D*, то общим корнем назначим вершину *B* (см. рис. 23-6).

После выбора корня необходимо отразить изменения в подчинённом поддереве *T2*, а именно: исправить поле **mRoot** и развернуть стрелки, которые пока не ведут к новому корню. Эти стрелки (Рис. 23-7) расположены между местом состыковки *E* и старым корнем *G*.

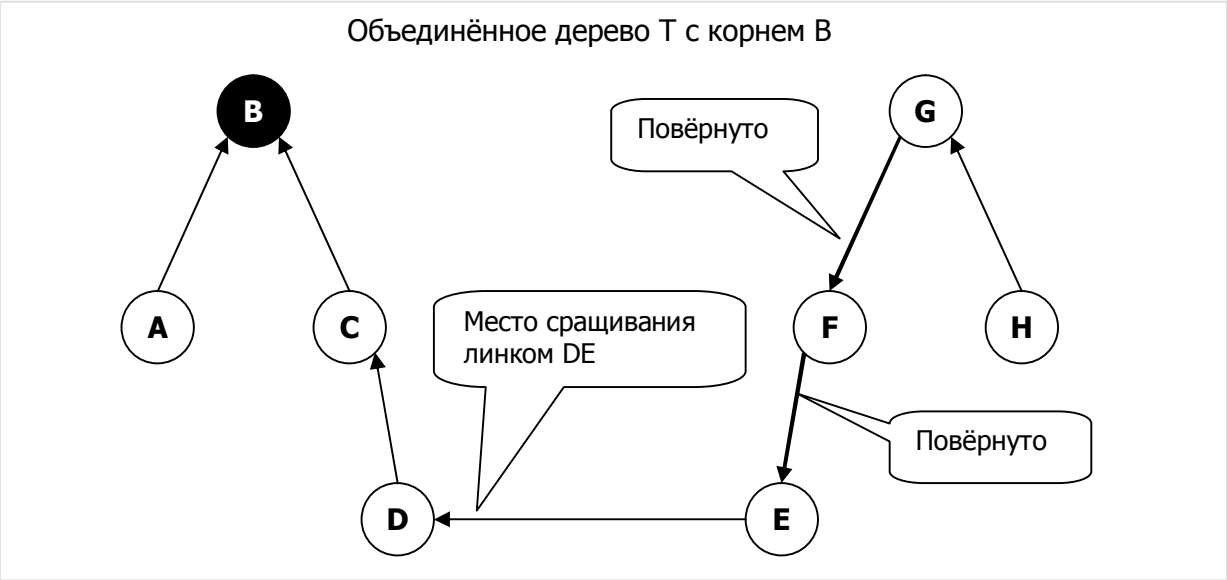


Рис. 23-7 — Состояние после сращивания

Результат сращивания поддеревьев показан на рис. 23-7, процедура сращивания, входящая в состав метода **ExpoAllTrees**, представлена в листинге.

Листинг 23-5 — Сращивание двух поддеревьев

```
function Union(aTL: TTreeLink): boolean;
var   N2: TNode;    // узел в T2
      p2: TNode;    // предшествующий узел
      tmp: TNode;   // временное хранилище
begin
  with aTL do begin
    // Если концы линка принадлежат одному поддереву, то выход
    Result:= mOwner.mRoot <> mDest.mRoot;
    if not Result then Exit;
    // Подготовка к реверсу обратных ссылок mPred
    N2:= mDest;          // текущая вершина в поддереве T2
    p2:= N2.mPred;       // предшествующая вершина в T2
    N2.mPred:=mOwner;    // новая предшествующая взята из T1
    // Реверсирование обратных ссылок:
    while Assigned(p2) do begin
      tmp:= p2.mPred;    // временно сохранить ссылку в T2
      p2.mPred:= N2;     // обновить обратную ссылку
      N2:= p2;           // сдвинуть вершину по направлению к 2-му корню
      p2:= tmp;          // восстановить ссылку в T2
    end;
    // Замена 2-го корня на 1-й:
    // root1 = mOwner.mRoot.Expo
    // root2 = mDest.mRoot.Expo
    p2:= mDest.mRoot;
    N2:= NodeFirst;
    while Assigned(N2) do begin
      if N2.mRoot = p2 then N2.mRoot:= mOwner.mRoot;
      N2:= NodeNext;
    end;
  end;  // with
end;
```

23.3.5. Расщепление дерева

Расщепление дерева выполняется для восстановления состояния леса при подъёме по дереву перебора. Если последним присоединённым линком был линк *D-E*, то удаляется именно он (рис. 23-8).

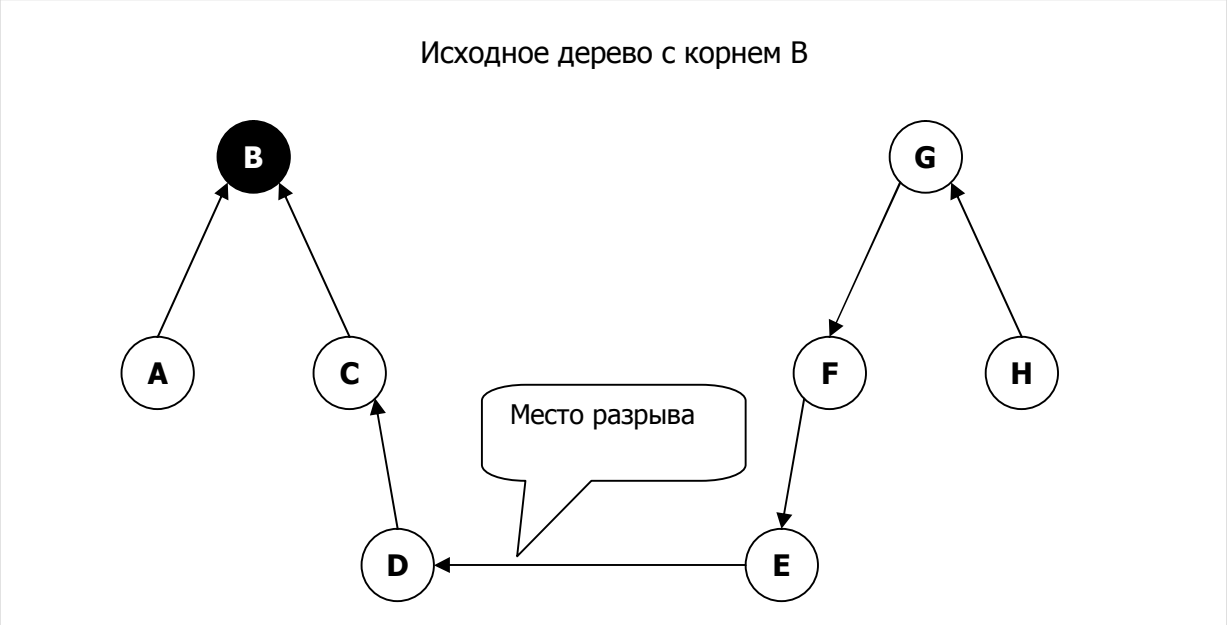


Рис. 23-8 — Состояние перед расщеплением дерева

Здесь, как и при сращивании поддеревьев, никаких операций с левым поддеревом не потребуется, поскольку тамошний «вожак» уже выбран, и все стрелки ведут к нему. Но для правого поддерева надо выбрать новый корень и направить стрелки в его сторону. Поскольку нам безразлично, которая из вершин поддерева станет его новым корнем, выберем ту, с которой будет меньше хлопот. Поскольку все стрелки правого поддерева уже ведут к вершине *E* (а это, заметим, место разрыва), выгодней назначить корнем именно эту вершину (рис. 23-9).

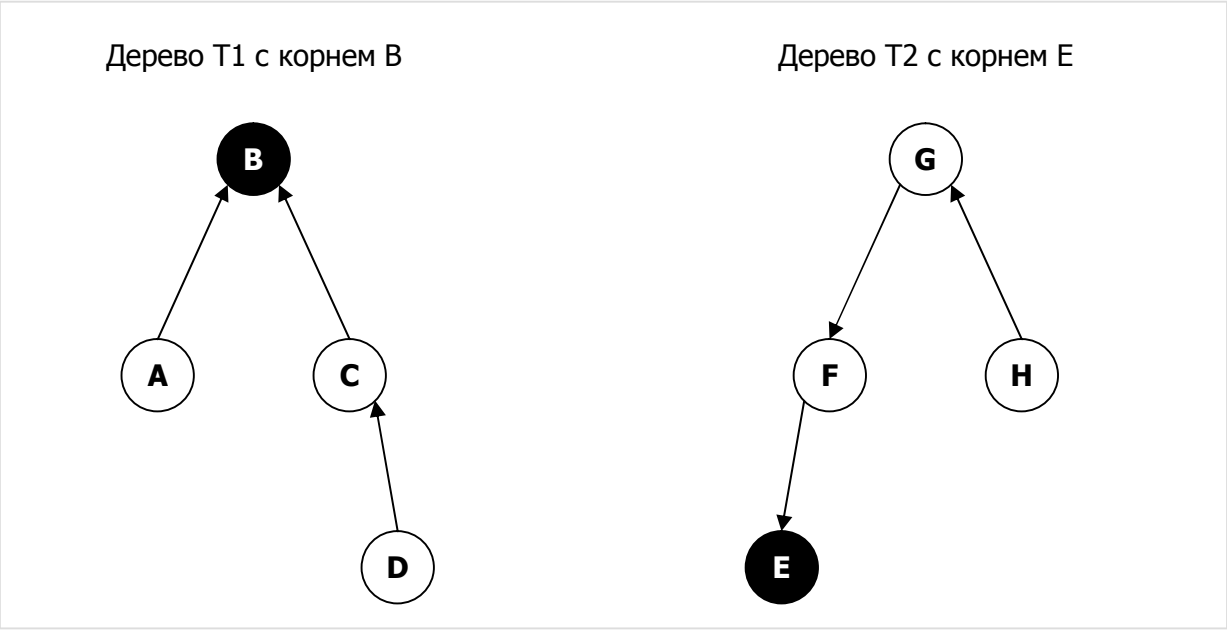


Рис. 23-9 — Состояние после расщепления дерева

Далее надо пройти по вершинам правого поддерева *T2* , заменяя прежний корень *B* на новый корень *E*. Здесь, — увы — придётся «грести против течения», поскольку стрелки ведут к корню, а не наоборот. Будем обрабатывать вершины «послойно». Изначально среди всех вершин поддерева *T2* доступна только корневая вершина; исправим в ней метку **mRoot** (будет указывать на неё же) и поместим эту вершину в множество **S1**, представляющее *внутренний слой*. Далее переберём все вершины графа в поисках тех, что примыкают своими стрелками (полями **mPred**) к этому *внутреннему слою*. Найденные вершины вставим в множество *внешнего* слоя **S2**, попутно заменяя в этих вершинах корневую метку **mRoot**. Если множество **S2** окажется пустым, то процедура завершается. Иначе *внешний* слой копируем во *внутренний*, после чего внешний слой очищаем и повторяем поиск примыкающих вершин. Листинг 23-6 с процедурой расщепления дерева представлен ниже.

Можно заметить, что после расщепления (рис. 23-9) поддерево *T2* не вполне совпадает с тем, каким оно было до сращивания (рис. 23-6), — изменилась и корневая вершина, и направление стрелок. Однако эти изменения не влияют на ход решения, поскольку по-прежнему все вершины поддерева связаны, и все помечены одним корнем. Да, возможно бывший «вожак стаи» — вершина *G* — обижен, но тут, как говорится, ничего личного...

Листинг 23-6 — Расщепление дерева

```
  procedure Remove(aTL: TTreeLink);
  var Root: TNode;  // Новый корень для 2-го поддерева
      S1 : TSet;    // внутренний слой вершин
      S2 : TSet;    // внешний слой вершин
      Node: TNode;
  begin
    S1:= CreateSet;    S2:= CreateSet;
    // Выбираем новый корень для 2-го поддерева
    with aTL do if mOwner.mPred = mDest
      then Root:= mOwner    // источник связи
      else Root:= mDest;    // приёмник связи
    Root.mRoot:= Root;      // пометка корня самим собой
    Root.mPred:= nil;       // обратная ссылка в корне пуста
    S1.Insert(Root);        // организуем нулевой слой из корневой вершины
    repeat
      // Накопление внешнего слоя:
      Node:= NodeFirst;
      while Assigned(Node) do begin
        // если примыкает к внутреннему слою, то добавляем к внешнему
        if S1.Exist(Node.mPred) then begin
          Node.mRoot:= Root;
          S2.Insert(Node);
        end;
        Node:= NodeNext;
      end;
      // Выход, если внешний слой пуст
      if S2.GetCount=0 then Break;
      // Копирование внешнего слоя во внутренний и его очистка
      S1.CopyItems(S2);
      S2.Clear;
    until false;
    S1.Free; S2.Free;
  end;
```

23.3.6. Перебор связей

Мы близки к финалу и можем представить листинг 23-7 с методом, перебирающим без повторений все остовные деревья графа. Для экономии места тексты трёх рассмотренных выше локальных процедур заменены здесь ссылками на листинги.

Листинг 23-7 — Перебор всех остовных деревьев

```
procedure TGraph.ExpoAllTrees;
var
  Links: TSet;        // множество всех линков графа
   Tree: TSet;        // множество линков дерева
  Level: integer;     // текущий уровень
     NN: integer;     // количество линков, обрабатываемых на каждом уровне
    Cnt: integer;     // счётчик остовных деревьев
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инициализация
  procedure Local_Init;
  {Листинг 23-4 }
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Объединение двух поддеревьев
  function Union(aTL: TTreeLink): boolean;
  {Листинг 23-5}
```

```
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Удаление ребра и расщепление дерева
  procedure Remove(aTL: TTreeLink);
  {Листинг 23-6}
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вспомогательная функция вычисления стоимости остова
  // как суммы весов его рёбер
  function Cost(aTree: TSet): integer;
  var TL: TTreeLink;  // текущий линк для дерева
  begin
    Result:= 0;
    TL:= aTree.GetFirst as TTreeLink;
    while Assigned(TL) do begin
      Inc(Result, TL.mValue);
      TL:= aTree.GetNext as TTreeLink;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивный перебор всех подмножеств (комбинаций) из L по N-1 линков
  // L - количество рёбер или дуг графа = Links.GetCount
  // N - количество вершин графа = mNodes.GetCount
  procedure Searching(aIndex: integer { стартовый индекс } );
  var  i: integer;
      TL: TTreeLink;  // текущий линк для дерева
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      TL:= Links.GetItem(i + Level) as TTreeLink;  // очередной линк
      if Union(TL) then begin
        // Этот линк не создаёт цикла, вставляем в остов
        Tree.Insert(TL);
        if Level = mNodes.GetCount-2 then begin
          // Остов сформирован, вывод:
          Inc(Cnt);       // счётчик остовных деревьев
          Write(Cnt:4, '':2, Cost(Tree):3, '':2);
          Tree.Expo;
        end else begin
          // Здесь размер подмножества не достигнут, остов не сформирован,
          // погружаемся на следующий уровень
          Inc(Level);     // Level+1
          Searching(i);   // рекурсивный вызов следующего уровня
          Dec(Level);     // Level-1
        end;
        // Возвращаем предыдущее состояние поддеревьев
        Tree.Delete(TL);  // удаляем из остовного множества
        Remove(TL);       // и расщепляем дерево
      end; // if
    end; // for
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin
  Local_Init;
  Cnt:= 0;      // счётчик остовных деревьев
  Level:=0;     // текущий уровень
  // количество линков, обрабатываемых на каждом уровне:
  // (mNodes.GetCount - 1) = количество рёбер дерева
  NN:= Links.GetCount + 1 - (mNodes.GetCount - 1);
  Searching(1);
  Links.ClrAndDestroy;
  Links.Free;
end;
```

23.4. Испытание

Испытание методов построения минимального остова и перебора всех остовов выполнено на следующем графе:

```
Test.txt
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

Ниже дана тестирующая программа:

Листинг 23-8 — Программа для тестирования методов поиска остовных деревьев

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Tree: TSet;
    Cost: integer;
begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('------------------------------');
  Tree:= Gr.GenCoverTree(false, Cost); // минимальный остов
  Tree.Expo;
  Writeln('Cost= ', Cost);
  Tree.ClrAndDestroy;  Tree.Free;
  Writeln('------------------------------');
  Tree:= Gr.GenCoverTree(true, Cost); // максимальный остов
  Tree.Expo;
  Writeln('Cost= ', Cost);
  Tree.ClrAndDestroy;  Tree.Free;
  Gr.ExpoAllTrees;           // все остовные деревья
  Readln;
end.
```

Здесь показана первая и последняя десятка из 118 остовных деревьев, полученных в ходе перебора:

```
   1   36  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 BC= 8 } : 7
   2   36  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 AF= 8 } : 7
   3   38  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 AG= 10 } : 7
   4   38  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 AE= 10 } : 7
   5   39  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BC= 8 AG= 10 } : 7
   6   39  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 AF= 8 AG= 10 } : 7
   7   41  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 AG= 10 AE= 10 } : 7
   8   38  { AC= 1 EF= 4 BH= 4 AD= 6 BG= 7 BC= 8 AF= 8 } : 7
   9   40  { AC= 1 EF= 4 BH= 4 AD= 6 BG= 7 BC= 8 AE= 10 } : 7
  10   40  { AC= 1 EF= 4 BH= 4 AD= 6 BG= 7 AF= 8 AG= 10 } : 7
. . .
 109   53  { BH= 4 AD= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 110   49  { BH= 4 CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AG= 10 } : 7
 111   49  { BH= 4 CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AE= 10 } : 7
 112   52  { BH= 4 CD= 6 EH= 6 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 113   53  { BH= 4 CD= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 114   51  { AD= 6 CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AE= 10 } : 7
 115   53  { AD= 6 CD= 6 EH= 6 BG= 7 AF= 8 AG= 10 AE= 10 } : 7
 116   54  { AD= 6 CD= 6 EH= 6 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 117   55  { AD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 118   55  { CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
```

Деревья следуют в порядке возрастания их стоимости (веса), что обусловлено порядком сортировки вспомогательных линков. При соответствующем изменении метода **`TTreeLink.Compare`** этот порядок можно обратить, а также построить метод поиска *максимального* остова, взяв за основу метод **`GenMinCover`**, и изменив там некоторые условия.

К этому добавим, что количество всех мыслимых комбинаций из **`11`** рёбер по **`7`** составляет **`330`**, но только **`118`** из них дали графы, не содержащие циклов.

23.5. Ориентированные деревья

Итак, остовные деревья для *неориентированных* связанных графов построены. А как обстоит дело с орграфами? Здесь интерес может представлять ориентированный остов, то есть, ориентированное дерево с корнем. Для начала ознакомимся с особенностями и свойствами таких деревьев.

Определения, данные для неориентированного дерева, справедливы и здесь, но в силу ориентации связей взаимная достижимость всех вершин тут невозможна. Особыми правами в таком дереве наделена вершина, называемая корнем (**`Root`**). В зависимости от направления связей, они могут вести либо от корня к периферийным вершинам, либо, наоборот — от периферии к корню (рис. 23-10). Назовём эти деревья соответственно *прямым* и *инверсным*. В *прямом* дереве корень недостижим из других вершин, но все вершины достижимы из корня. В *инверсном*, наоборот: корень достижим из любой вершины, но, ни одна периферийная вершина не достижима из него. В *прямом* дереве полустепень *захода* любой вершины, кроме корня, равна единице (поскольку имеется единственная входящая связь). В *инверсном* дереве это справедливо для полустепени *исхода*.

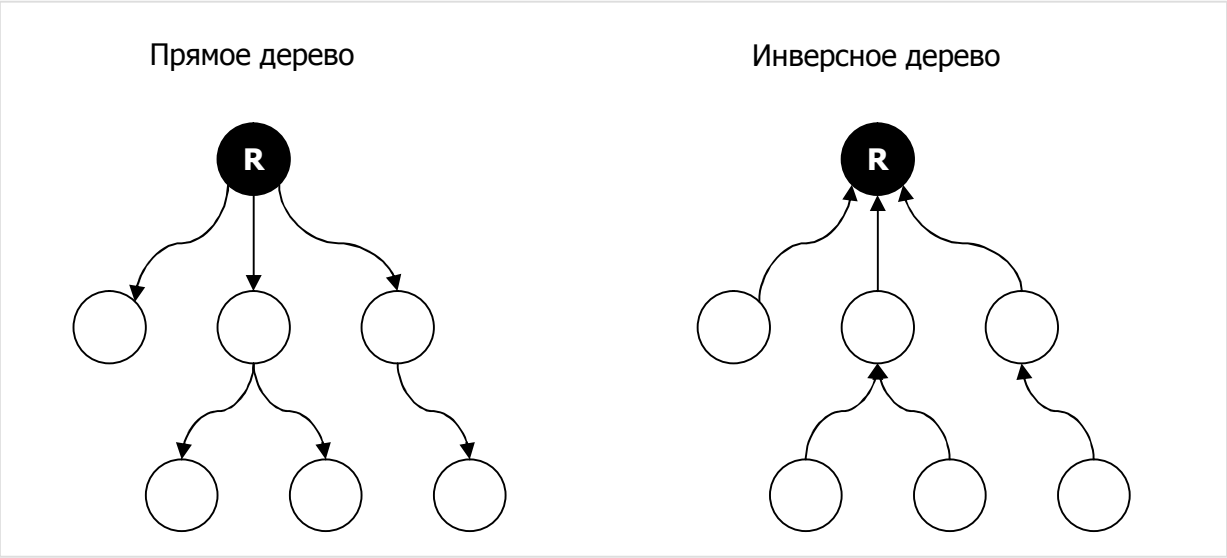


Рис. 23-10 — Ориентированные деревья с выделенным корнем

В качестве примера рассмотрим историю постройки минимального ориентированного остова, решившую судьбу государства. Обратимся к рис. 23-11, где показан архипелаг и мосты с односторонним движением.

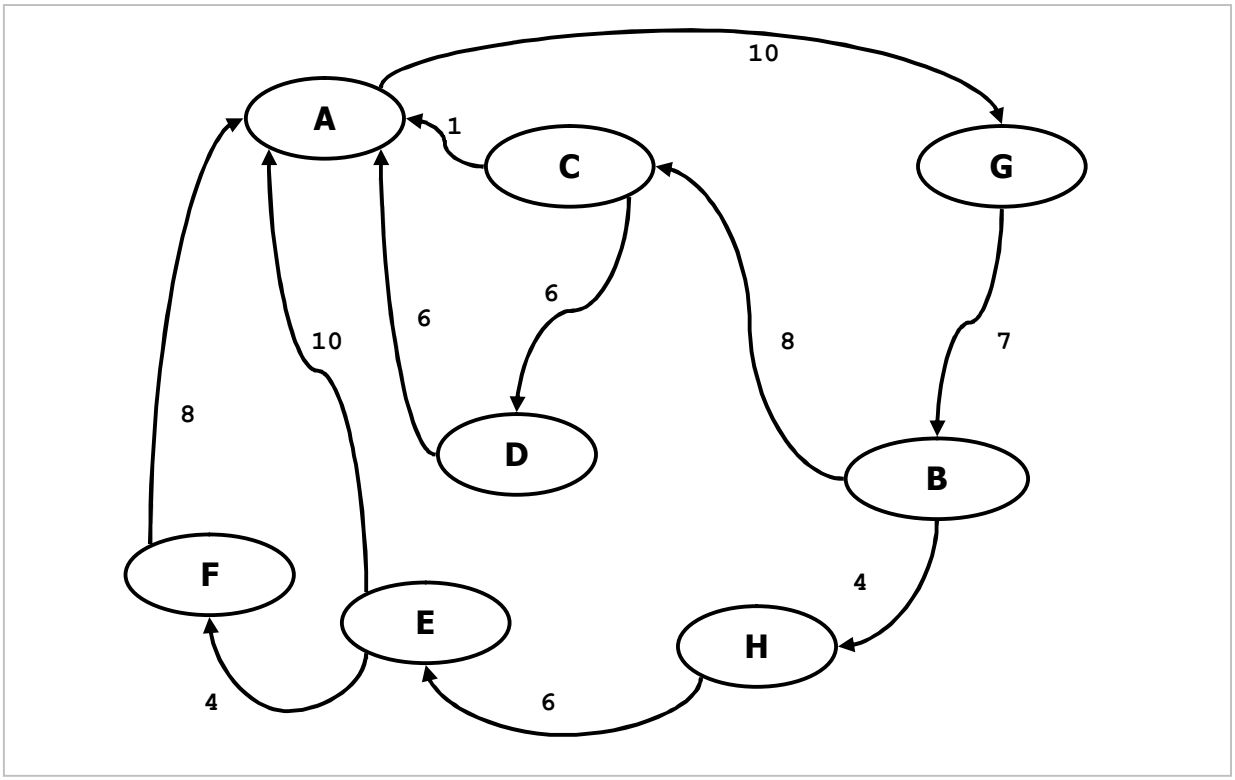


Рис. 23-11 — Ориентированный граф

В этом сильно связанном орграфе все вершины взаимно достижимы. Когда правительство — в целях экономии — вывело из эксплуатации часть мостов, граф остался связным, однако не сильно связным. Для обеспечения взаимной достижимости островов, необходимой для нормальной жизни, прибегли к

периодической *инверсии* движения на оставшихся мостах, меняя направления на противоположные.

После этой реформы возникла потребность выбрать новый столичный остров с тем, чтобы гарантировать экономичную почтовую связь столицы со всеми островами. Эта связь была организована следующим образом. Посыльные из столицы доставляли почту к *ближайшим* соседям, а затем, после инверсии направлений на мостах, возвращались с почтой назад. На последующие периферийные острова почта попадала точно также, но с другими посыльными, курсирующими между соседними островами. Таким образом, на каждый остров со стороны центра прибывал ровно *один* посыльный. Обратно в направлении центра их приходило столько, сколько было у данного острова ближайших периферийных соседей. Таким образом, система являла собой дерево с выделенной вершиной — столицей архипелага. И задача состояла в том, чтобы выбрать столицу (корень) и множество мостов (ориентированный остов) с минимальной общей длиной.

Какое дерево будем искать: *прямое*, или *инверсное* ? Поскольку ориентация их дуг взаимно противоположна, и одно легко получается из другого, поищем *прямое* дерево. Выбрав в качестве корня одну из вершин графа, все последующие «подцепим» к ней кратчайшими *исходящими* дугами, следуя алгоритму Прима. Поскольку орграф с выделенной вершиной не сильно связан, может случиться, что далеко не каждый корень породит покрывающий остов. Есть вероятность, что в ходе перебора корней ни один покрывающий ориентированный остов сформировать так и не получится (рис. 23-12).

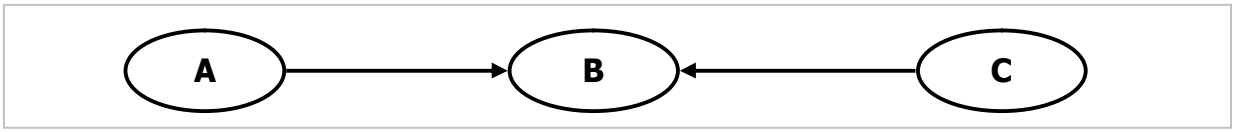


Рис. 23-12 — Для этого графа не существует направленного остова с выделенной вершиной

Ниже представлен метод поиска минимального покрывающего ориентированного остова и соответствующего ему корня.

Листинг 23-9 — Поиск минимального ориентированного остова с выделенной вершиной (алгоритм Прима)

```
function TGraph.GenCoverDir(var aRoot: TNode;    // корень
                            var aCost: integer   // вес (стоимость) дерева
                           ): TSet;              // само дерево
var Tree: TSet;      // текущее дерево
    Links: TSet;     // текущее множество линков
  //.............................................................
  // Локальная функция формирует дерево, начиная с вершины aRoot
  // алгоритмом Прима и возвращает его вес (стоимость).
  // Не сформированное дерево очищается
  function RootCover(aRoot: TNode): integer;
  var Node: TNode;        // очередная вершина
      TL : TTreeLink;     // очередной линк
  begin
    Result:= 0;           // вес дерева
```

```
    ResetNodes;           // для всех вершин mColor = CWhite
    Node:= aRoot;         // исходная вершина -- корень
    // Пытаемся присоединенить все вершины графа (линков на единицу меньше)
    while Assigned(Node) and
         (Tree.GetCount < mNodes.GetCount-1) do  begin
      Node.mColor:= CBlack;     // текущую в множество чёрных
      // добавляем линки к ближайшим белым вершинам:
      Node.AddTreeLinks(false, Links);
      // Перебираем линки Links в поиске ближайшей белой вершины
      TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
      Node:= nil;                         // искомая вершина пока пуста
      while Assigned(TL) do begin         // пока существуют линки
        Links.Delete(TL);                 // удаляем линк из буфера Links
        // Искомая вершина может быть только приёмником связи:
        if TL.mDest.mColor = CWhite then begin
          // Здесь ближайшая белая вершина найдена:
          Node:= TL.mDest;        // будет обработана в следующем цикле
          Tree.Insert(TL);        // вставляем линк в результат
          Inc(Result, TL.mValue); // наращиваем стоимость
          Break;
        end;
        // здесь линк связывает две чёрные вершины:
        TL.Free;                          // удаляем ненужный линк
        TL:= Links.GetNext as TTreeLink; // и берём следующий
      end;
    end;
    Links.ClrAndDestroy; // очистка оставшихся в очереди линков
    // Если остов не построен, очищаем накопитель линков
    if Tree.GetCount < mNodes.GetCount-1 then begin
      Tree.ClrAndDestroy;  // очистка поддерева
      Result:= MaxInt;
    end;
  end;
  //.......................................................
var Root: TNode;     // очередной корень дерева
    Cost: integer;  // стоимость (вес) дерева
Begin  { TGraph.GenCoverDir }
  Result:=nil;  aRoot:= nil;  aCost:= MaxInt;
  Links:= CreateSet;  // текущее множество линков
  Tree:= CreateSet;   // текущее остовное дерево
  // Перебор всех вершин:
  Root:= NodeFirst;
  while Assigned(Root) do begin
    Cost:= RootCover(Root);
    // Если дерево построено:
    if Tree.GetCount > 0 then begin
      // Если стоимость меньше максимальной
      if Cost<aCost then begin
        // то сохраняем результат
        if Assigned(Result)
          then Result.ClrAndDestroy // удаление существующих линков
          else Result:=CreateSet;   // создание копии дерева
        Result.CopyItems(Tree);     // копируем дерево (линки)
        Tree.Clear;                 // удаляем сохранённые линки
        aRoot:= Root;               // запоминаем корень
        aCost:= Cost;               // запоминаем стоимость (вес)
      end else begin
        Tree.ClrAndDestroy;  // очистка дерева, если оно не минимально
      end;
    end;
    Root:= NodeNext;
  end;
```

```
  // Удаление рабочих буферов:
  Tree.Free;
  Links.Free;
end;
```

Метод тестировался следующей программой:

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;       // исходный граф
    Tree: TSet;        // множество дуг дерева
    Root: TNode;       // корень дерева
    Cost: integer;     // стоимость дерева

begin
    Gr:= TGraphChars.Load('Test.txt');
    Gr.Expo;
    Tree:= Gr.GenCoverDir(Root, Cost);
    if Assigned(Tree) then begin
       Root.Expo;
       Tree.Expo;
       Writeln('Cost= ', Cost);
       Tree.ClrAndDestroy;
       Tree.Free;
    end;
    Readln;
end.
```

Для испытаний взят граф, изображённый на рис. 23-11:

```
Test.txt
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

В результате получен минимальный остов с корнем в G и стоимостью 36, показанный на рис. 23-13. Пунктиром обозначены выведенные из эксплуатации мосты.

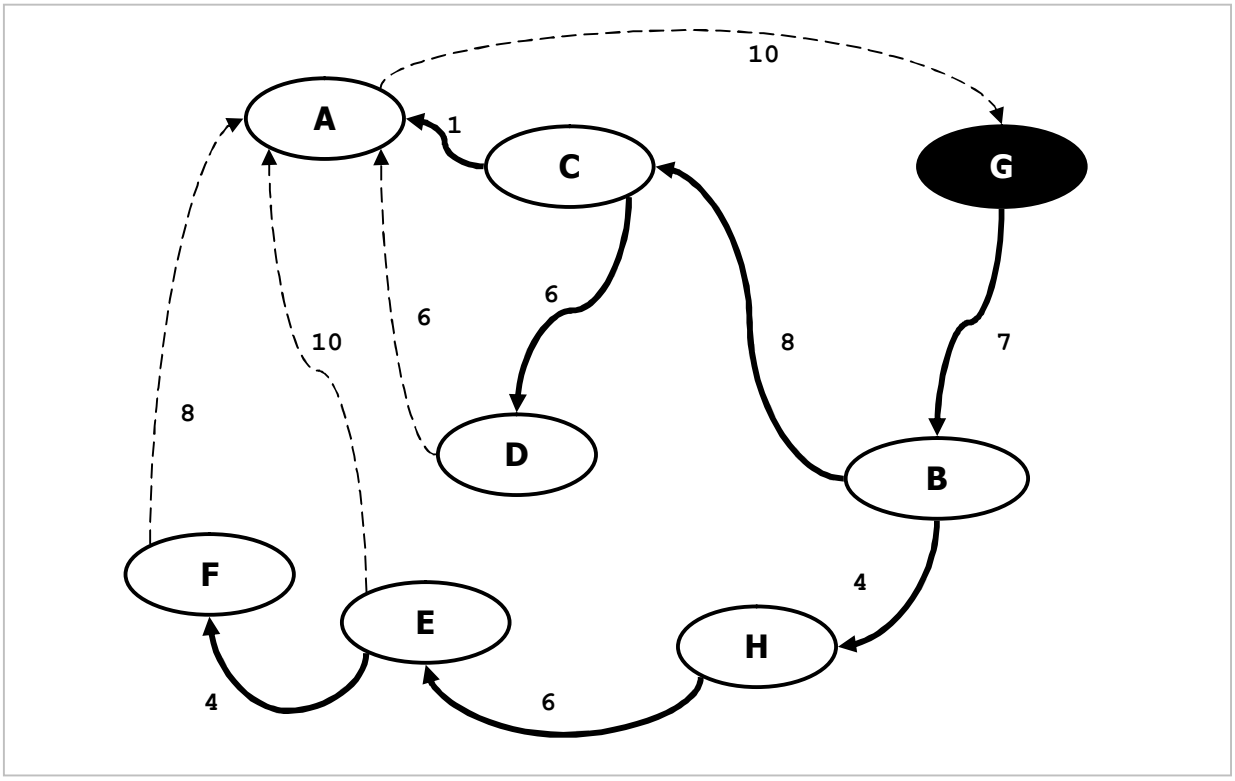


Рис. 23-13 — Минимальный ориентированный остов
с корнем в G и стоимостью 36

23.6. Итоги

- Остовным деревом графа называется связанный граф, содержащий N-1 ребро (N— число вершин графа). Остов не содержит циклов.
- Каждое ребро или дуга остовного дерева соответствует ребру или дуге исходного графа или орграфа.
- Количество различных остовов в графе может быть весьма велико, и в общем случае задача перебора всех остовов экспоненциально сложна.
- Алгоритм Прима находит минимальный либо максимальный остов графа, — в зависимости от порядка сортировки вспомогательных линков и условий проверки.
- Поиск всех остовов графа состоит в переборе комбинаций по N-1 линков, при этом линки проверяются на возможность образования циклов. В ходе перебора попеременно соединяются и расщепляются поддеревья.
- В ориентированном графе при подходящих условиях могут быть построены ориентированные остовы с выделенной вершиной — корнем дерева.

23.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 162
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 24
Максимальный поток

В этой и двух последующих главах рассмотрены задачи о *потоках* в сетях. *Сетью* называют ориентированный граф, каждой дуге которого приписана некоторая пропускная способность. Так моделируют перенос материальных или информационных объектов: автомобилей по улицам, жидкостей по трубам, пассажиров по маршрутам. Пропускная способность дуг выражается единицами объектов в единицу времени: автомобилями в час, кубометрами в минуту, информационными пакетами в секунду и т.п.

Рассмотрим город, пропускающий через себя оживлённую трассу. Его улицы подобны дугам, а перекрёстки — вершинам орграфа. Двустороннее движение по улицам можно представить парами дуг, причём встречные дуги могут обладать разной пропускной способностью. Пусть транспортный поток вливается в город через вершину *S*, далее растекается по улицам, затем собирается в вершине *T* и уходит на трассу. Касательно этого движения можно задать два вопроса: 1) каков здесь максимально возможный поток? 2) каким образом распределить поток по улицам с тем, чтобы достичь этого максимума? В этом состоят задачи о *максимальном потоке*, алгоритмы их решения лежат в основе решений других разновидностей потоковых задач.

24.1. Основные понятия

На рис. 24-1 показан орграф, отражающий суть обсуждаемой транспортной задачи. Две выделенные вершины представляют исток *S* и сток *T* потока. Все прочие вершины — промежуточные. Пропускная способность указана рядом с дугами и в некоторой степени отражена толщиной дуг.

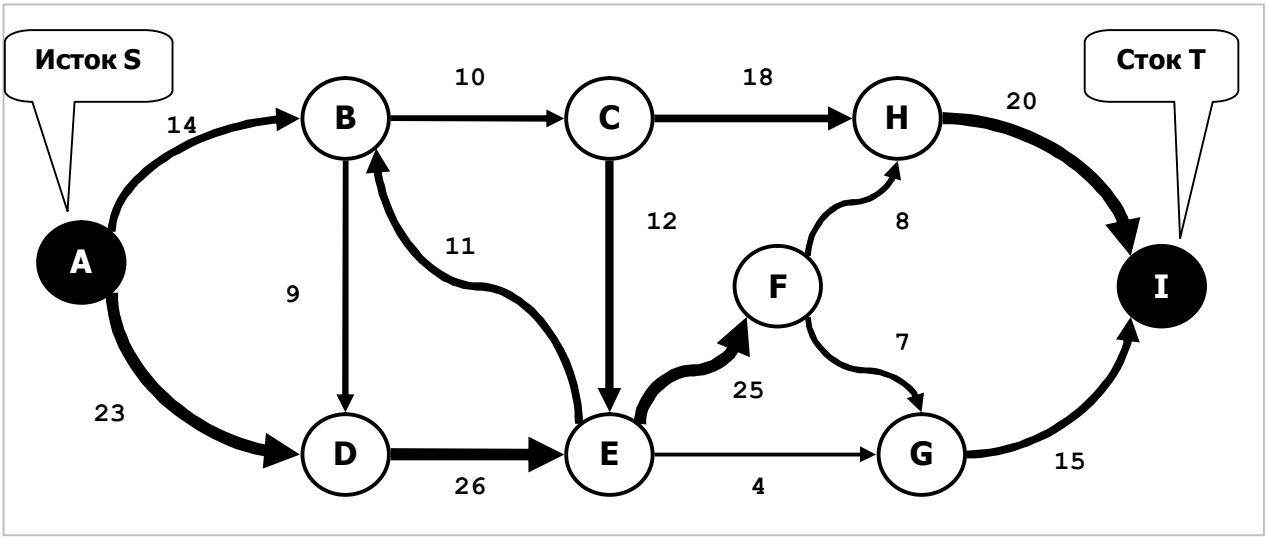


Рис. 24-1 — Граф для транспортной задачи с источником S и стоком T

Вершина-исток S порождает некие элементы потока и отправляет их в сеть. Соответственно вершина-сток T поглощает эти элементы. Все прочие вершины (промежуточные) только пропускают поток в направлении дуг. Справедливы очевидные утверждения:

- Количество транспортируемых элементов, входящих в промежуточную вершину, в точности равно количеству исходящих из неё. То есть, сумма потоков для промежуточной вершины равна нулю, если входящим и исходящим потокам приписаны разные знаки.
- Максимальный поток, передаваемый из источника в сток, не может превышать сумму пропускных способностей дуг, инцидентных *истоку*.
- Этот же максимальный поток не может превышать сумму пропускных способностей дуг, инцидентных *стоку*.
- Максимальный поток, следующий вдоль некоторой цепочки дуг от источника к стоку, ограничен пропускной способностью самой «узкой» дуги в этой цепочке — *критической* дуги.

24.2. Способы увеличения потока

Идея решения задачи о максимальном потоке состоит в постепенном наращивании потока, начиная с некоторого минимума. В этой связи рассмотрим два пути увеличения потока, протекающего от источника к стоку. На рис. 24-2 представлен простейший орграф, где стоку T отведена роль пресловутого бассейна с двумя трубами, через одну из которых — прямую — вода поступает в бассейн, а через другую — обратную — вытекает и возвращается в исток.

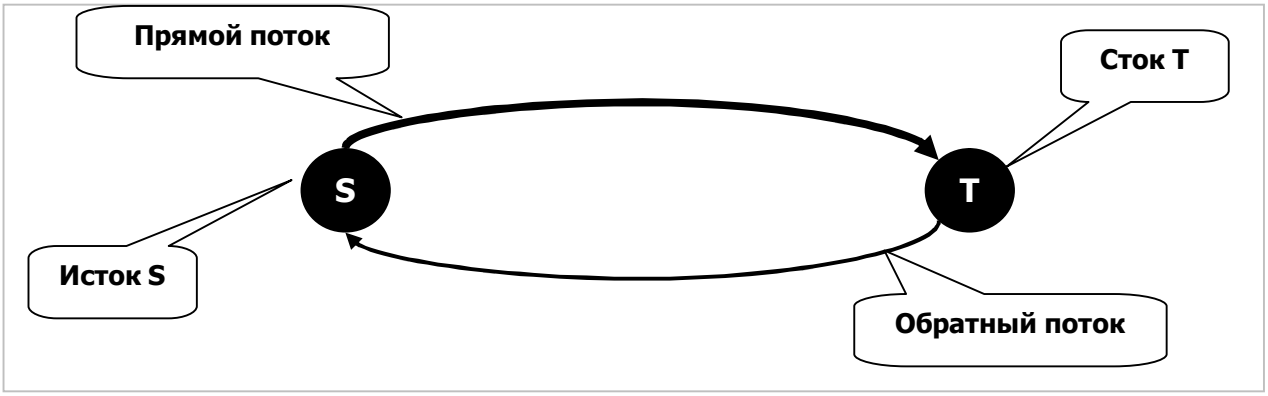


Рис. 24-2 — Упрощённая схема потоковой задачи

Пусть в какой-то момент вода течёт в обоих направлениях. Как можно ускорить наполнение бассейна T? Есть два очевидных способа:

- Увеличить прямой поток вплоть до насыщения трубы.
- Если имеется обратный поток, уменьшить эту утечку, вплоть до нуля.

В реальном графе картина течений сложнее, но основные идеи те же: будем искать такие цепочки дуг, вдоль которых можно либо *увеличить прямой* поток, либо *уменьшить обратный*.

24.3. Увеличивающие, уменьшающие и нейтральные дуги

Цепью называют ряд попарно инцидентных дуг, соединяющих исходную и конечную вершины (*исток* и *сток*), причём дуги этой цепи вовсе не обязательно имеют одно направление. Для поиска *увеличивающих* цепей распределим все дуги на три класса: *I* (Increase), *R* (Reverse), *N* (Neutral). Класс дуги определяется её *направлением* в выбранной цепи (прямая или обратная) и протекающим в ней *потоком*. И то, и другое будет меняться по ходу решения задачи, соответственно будет меняться и класс дуги.

На Рис. 24-3 показаны две возможные цепи, соединяющие исток и сток сети. В первой из них дуга *A->B* является прямой, во второй — обратной.

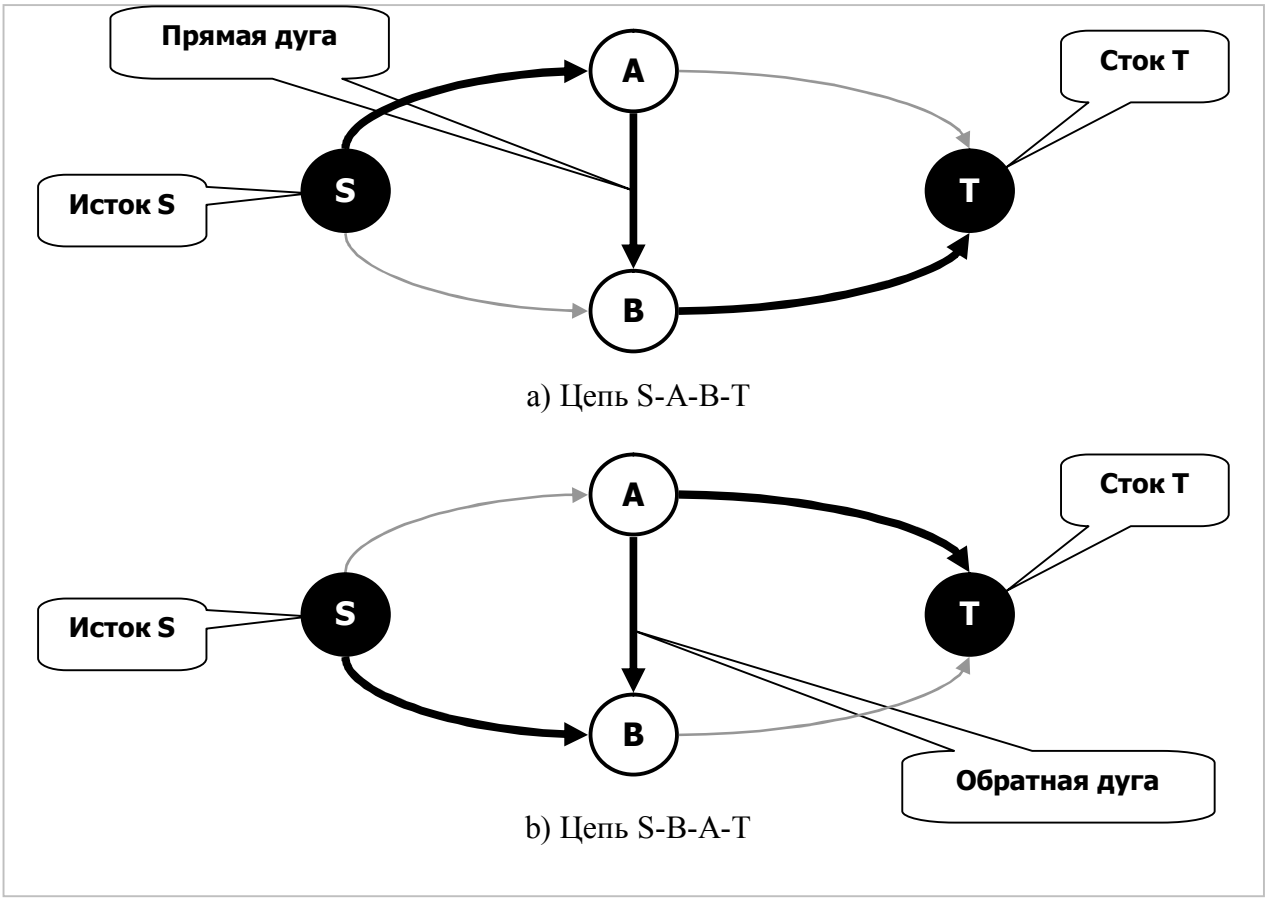


Рис. 24-3 — Изменение направления дуги в разных цепях

Далее класс дуги будем определять по отношению к конкретной вершине, инцидентной этой дуге. Причём к той вершине, которая расположена в данной цепи ближе к истоку *S*. В цепи *S-A-B-T* ближе к истоку находится вершина *A*, здесь дуга *A->B* является *исходящей* и *прямой*. В цепи *S-B-A-T* ближе к истоку находится вершина *B*, здесь дуга *A->B* является *входящей* и *обратной*.

Другим фактором, определяющим класс дуги, является протекающий в ней на данном этапе решения поток. Итоговая таблица классификации дуг представлена ниже.

Таблица 24-1 — Классы дуг

Класс дуги	Направление по отношению к данной цепи	Описание
I (Increase)	Прямая	Текущий поток ниже пропускной способности, через такую дугу можно увеличить общий поток.
R (Reverse)	Обратная	Текущий поток не нулевой, уменьшив эту утечку, можно увеличить общий поток.
N (Neutral)	Прямая	Текущий поток равен пропускной способности (насыщен), увеличить поток нельзя.
	Обратная	Текущий поток нулевой, уменьшить утечку нельзя.

Прямые дуги, поток через которые ниже предела пропускной способности дуги (не насыщен), отнесём к классу *I* (Increase). Через такую дугу можно увеличить поток от истока к стоку.

К классу *R* (Reverse) — отнесём *обратные* дуги с *ненулевым* потоком. Уменьшив утечку через такую обратную дугу, можно в целом увеличить достигающий сток поток.

Наконец к третьему, *нейтральному* классу *N*, отнесём прочие дуги, то есть *прямые* дуги с *насыщенным* потоком, и *обратные* с *нулевым* потоком. Любое изменение потока в них неминуемо уменьшит поток от источника к стоку, и потому нейтральные дуги будут игнорироваться при поиске увеличивающих поток цепей.

24.4. Увеличивающие поток цепи

Пусть удалось найти цепь, составленную только из *прямых* дуг класса *I* (рис. 24-4). Поток, текущий по дугам, показан стрелками. Ясно, что вдоль этой цепи поток можно увеличить лишь на **3** единицы, поскольку таким минимальным запасом обладает дуга *Y-Z* (**5−2=3**).

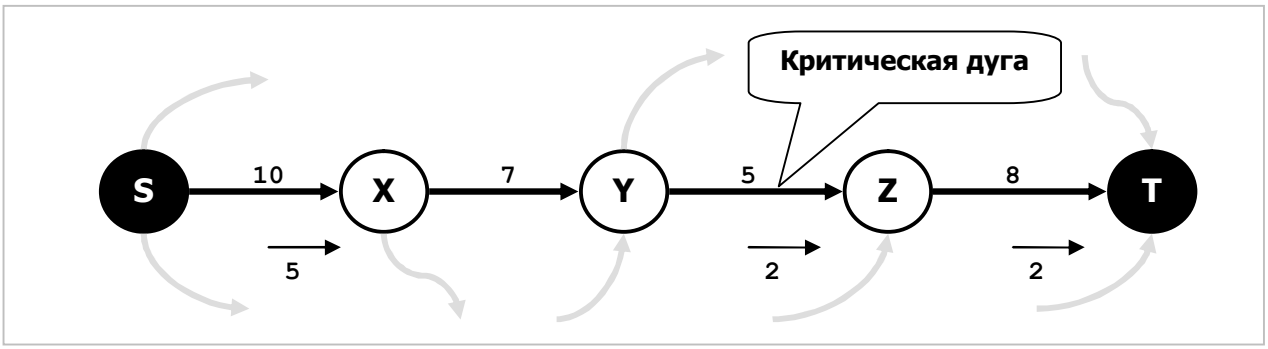


Рис. 24-4 — Увеличивающая цепь, составленная из прямых дуг класса I

Цепь на рис. 24-5 составлена только из *обратных* дуг, по которым «стекает назад» ненулевой поток. Снизить эту утечку можно не более чем на **2** единицы, и тем самым на **2** единицы увеличить потребление в стоке *T*.

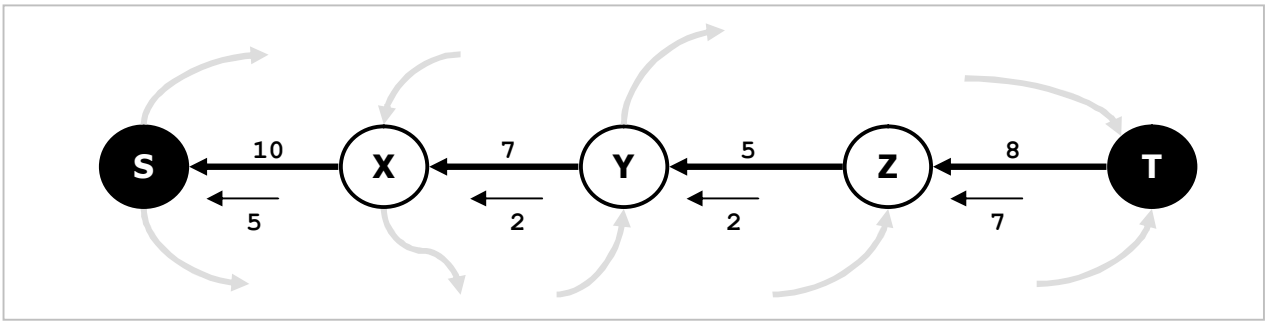


Рис. 24-5 — Увеличивающая цепь, составленная из обратных дуг класса R

Впрочем, цепей, состоящих только из обратных дуг, алгоритм не обнаружит. Скорее будут найдены *смешанные* цепи, содержащие как *прямые*, так и *обратные* дуги, вроде той, что показана на рис. 24-6.

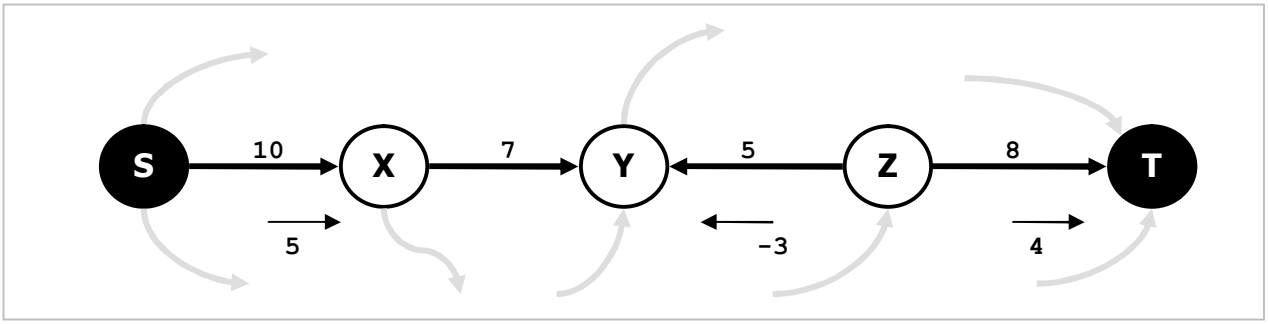


Рис. 24-6 — Увеличивающая цепь, составленная из прямых и обратных дуг

Обратим внимание на дугу *Z-Y*, по которой в обратном направлении движется поток в минус **3** единицы. Эти единицы потока создают местную циркуляцию, нагружая ею прямые дуги и затрудняя тем самым прямое движение потока. До какой степени можно увеличить поток вдоль такой цепи? Пройдём вдоль неё и найдём минимальное приращение потока, которое:

а) не превысит пропускной способности *прямых* дуг;
б) не превзойдет текущий в данный момент поток в *обратных* дугах (потоку в обратной дуге условно приписан знак минус).

Очевидно, что в данном примере поток можно увеличить на **3** единицы, после чего распределение потока вдоль цепи станет таким, как показано на рис. 24-7. Теперь поток в обратной дуге *Z-Y* иссяк, зато в прямых дугах он возрос. В конечном счете, общий поток от источника к стоку увеличился на **3** единицы. Отметим, что указанный поток обеспечивается течениями в других дугах сети, отмеченных на рисунке серым цветом, потоки в них для упрощения картины не показаны.

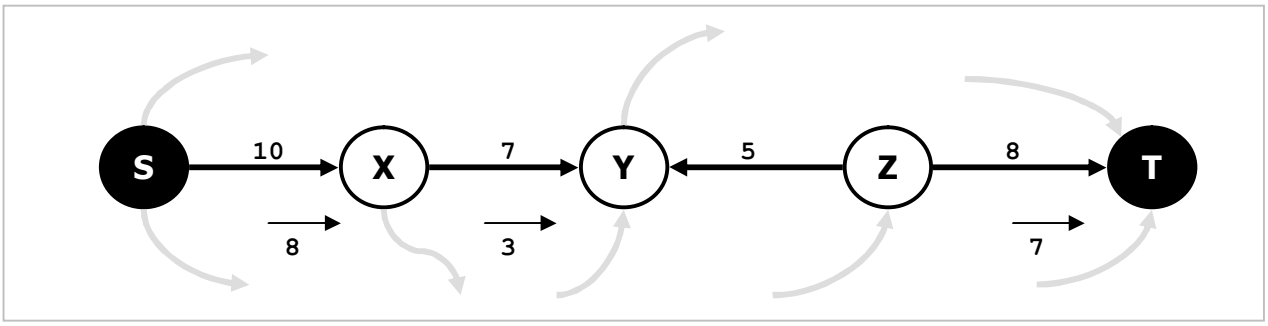


Рис. 24-7 — Поток вдоль цепи после его увеличения на 3 единицы

Из этих рассуждений вытекает принцип решения задачи, состоящий в следующем. Первоначально вдоль всех дуг графа пускаем нулевой поток. Далее отыскиваем увеличивающую цепь, состоящую только из дуг класса *I* и *R*, и наращиваем вдоль неё поток на предельно возможную для этой цепи величину. Если такую цепь найти не удаётся, попытки прекращаем, а иначе ищем следующую увеличивающую цепь с учётом уже текущего в дугах потока. После каждой такой итерации некоторые дуги будут переходить в нейтральный класс *N*, отчего будут исключаться при поиске цепей.

24.5. Поиск увеличивающей цепи

Можно составить много причудливых цепей между источником и стоком сети. Будем искать *кратчайшие* цепи, соединяющие источник и сток. То есть цепи, количество дуг в которых минимально. Подобная задача уже решалась в главе 15 при поиске кратчайших путей между вершинами без учёта длин рёбер.

Здесь поступим аналогично: в ходе поиска очередной цепи, начиная с истока *S*, будем расширять гамму вершин до тех пор, пока в этой гамме окажется вершина-сток *T* (вершины гаммы будем окрашивать). Попутно сформируем обратные ссылки и определим минимально допустимое увеличение потока вдоль цепи. По достижении стока пройдём по обратным ссылкам вдоль цепи назад в направлении истока, увеличивая поток вдоль прямых дуг, и уменьшая его вдоль обратных. В этом состоит одна итерация алгоритма, по завершении которой в графе появится, по меньшей мере, либо одна *насыщенная прямая* дуга, либо одна *истощённая обратная* дуга. Эти *нейтральные* дуги в последующих попытках игнорируются, и потому следующая итерация породит уже другую кратчайшую цепь, либо не породит её вовсе. В последнем случае алгоритм завершается.

24.6. Данные о потоке

Определившись с алгоритмом, снабдим нужными полями вершины и дуги графа. Для вершин такими полями будут **`mFlow`** — поток, протекающий через вершину в данный момент, и **`mLink`** — обратная ссылка в цепочке (дуга).

Поля дуг, используемые для решения потоковых задач, представлены ниже. Здесь два из них — **mLow** и **mTemp** — предстоит использовать в других потоковых задачах, они будут обсуждены позже.

```
// Линк -- ребро или дуга
TLink = class (TItem)
    ...
    // Для решения потоковых задач:
    mLow   : integer; // допустимые минимальный поток  mLow <= mHigh
    mHigh  : integer; // допустимые максимальный поток
    mFlow  : integer; // текущий поток
    mTemp  : integer; // для временного хранения mHigh
    mDirect: integer; // направление дуги (+1 -- прямая, -1 -- обратная)
    ...
end;
```

24.7. Ввод данных о потоке

Следующий шаг на пути создания потоковых методов — обеспечить ввод данных о потоке из текстового файла. Для решения настоящей и будущих потоковых задач необходимо ввести: а) предельную пропускную способность дуг; б) минимальную пропускную способность дуг, и в) стоимость дуг. Дабы не менять уже готовые процедуры ввода графа, для добавления потоковых данных создадим ещё один метод:

TGraphChars.LoadFlowData(const aName: String)

Он читает данные о потоке из указанного текстового файла, причём в качестве такового может служить файл, содержащий основные данные о графе. Начало потоковых данных пометим волшебным словом «FLOW», за которым последуют строки с данными, включающими дугу, а также её минимальную и максимальную пропускные способности, например:

```
FLOW:
AB= 1 2
AC= 1 3
BC= 0 1
BD= 1 4
```

Здесь в первой после метки строке даны параметры потока для дуги *A-B*, в следующей строке — для дуги *A-C*, и так далее. За подробностями реализации процедуры отсылаю к модулю **GrChars**.

24.8. Функция поиска максимального потока

Итак, разобравшись с хранением и загрузкой потоковых данных, приступим к главному — к методу поиска максимального потока между истоком **aSource** и стоком **aDest**. Поскольку алгоритм разобран уже достаточно подробно, приведём текст соответствующего метода, снабдив его лишь несколькими дополнительными пояснениями.

Листинг 24-1 — Поиск максимального потока при отсутствии нижней границы

```
function TGraph.CalcMaxFlow0(aSource, aDest: TNode): integer;
var Delta: integer; // очередное приращение потока
    Que: TBuffer;   // очередь вершин
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт начального потока
  // на случай, когда заданы минимальные потоки в сети (mLow>0)
  function CalcStartFlow: integer;
  var Link: TLink;    // исходящая дуга
  begin
    Result:= 0;
    Link:= aSource.OutLinkFirst;
    while Assigned(Link) do begin
      Inc(Result, Link.mFlow);
      Link:= aSource.OutLinkNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Построение увеличивающей цепи
  // с подсчётом соответствующего приращения потока
  function CalcDelta: integer;
  var Node: TNode;
      Link: TLink;    // исходящая или входящая дуга
  begin
    Result:=0;
    // установка mColor=0, mPred=nil, mDist= MaxInt, mFlow= MaxInt
    ResetNodes;
    Que.Clear;   // очистка очереди вершин
    // Пометку вершин начинаем с источника
    aSource.mColor:= CGray;
    Que.Put(aSource);
    while Que.GetCount>0 do begin
      Node:= Que.Get as TNode;
      // Выход из цикла, если помечен сток
      if Node=aDest then Break;
      // Обработка исходящих связей текущего узла:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Помечаем только белые вершины, достигаемые через ненасыщенные дуги
        if (Link.mDest.mColor=CWhite)            // если вершина не помечена
           and (Link.mFlow < Link.mHigh) then begin  // и дуга не насыщена
          Link.mDirect:= +1;  // это увеличивающаяся дуга
          // помечаем концевую вершину и ставим её в очередь:
          with Link.mDest do begin
            mColor:= CGray;    // цвет концевой вершины серый
            mPred:= Node;      // предшествующая вершина
            mLink:= Link;      // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mHigh - Link.mFlow); // поток
          end;
          Que.Put(Link.mDest); // в очередь приёмник дуги
        end;
        Link:= Node.OutLinkNext;
      end;
      // Обработка входящих связей текущего узла:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        // Помечаем только белые вершины,
        // из которых поступает ненулевой поток
        if (Link.mOwner.mColor=CWhite)          // если вершина не помечена
           and (Link.mFlow > Link.mLow) then begin // и поток можно уменьшить
          Link.mDirect:= -1;  // это уменьшающая дуга
```

```
          // помечаем вершину (источник дуги) и ставим её в очередь:
          with Link.mOwner do begin
            mColor:= CGray;     // цвет исходной вершины серый
            mPred:= Node;       // предшествующая вершина
            mLink:= Link;       // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
          end;
          Que.Put(Link.mOwner); // в очередь источник дуги
        end;
        Link:= Node.InLinkNext;
      end;  // while
    end;  // while

    // Если сток не помечен, то увеличивающая цепочка не найдена
    // и приращение потока равно нулю
    if aDest.mColor = CWhite then Exit;
    // Здесь  увеличивающая цепочка найдена
    Result:= aDest.mFlow; // допустимое увеличение потока
    // Обратное движение от стока к истоку с корректировкой потока
    Node:= aDest;  // сток
    while Node <> aSource do begin
      with Node.mLink do Inc(mFlow, mDirect * Result); // mDirect = +1 / -1
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.CalcMaxFlow0 }
  Result:= 0;
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
  Result:= CalcStartFlow; // начальный поток
  Que:= TBuffer.Create;   // рабочая очередь вершин
  // Повторяем поиск увеличивающих цепей, пока это возможно
  repeat
    Delta:= CalcDelta;    // подсчёт приращения потока
    Inc(Result, Delta);   // накопление результата
  until Delta=0;
  Que.Free;
end;
```

Вначале вызывается локальная функция **CalcStartFlow**, подсчитывающая текущий суммарный поток, исходящий из источника. Здесь эта функция будет возвращать ноль, поскольку никакого начального потока не существует. Она заготовлена для задачи, рассмотренной в главе 25, где поток в дугах ограничен не только сверху, но и снизу.

Далее циклически вызывается локальная функция **CalcDelta**, строящая очередную увеличивающую цепь и находящая максимальное приращение потока вдоль неё. Эта же функция изменяет текущий поток в дугах. Если увеличивающую цепь построить не удаётся, функция возвращает ноль и тогда метод завершается.

Внутри локальной функции **CalcDelta** вначале очищаются данные в вершинах: цвет, обратные ссылки, а предполагаемый поток через вершину назначается «бесконечным». Затем красится и ставится в очередь обработки вершина-источник дуги.

В цикле обработки очереди происходит расширение гаммы вершин и дуг. Из очереди выбирается очередная вершина, и обрабатываются сначала её исходящие (прямые) дуги, а затем входящие (обратные). Если *исходящая* дуга ведёт к неокрашенной вершине и поток в ней не насыщен, то дуга метится как прямая (**mDirect:=+1**), а в вершине-приёмнике этой дуги корректируются соответствующие данные: цвет, обратные ссылки и поток через вершину. Причём поток в вершине-приёмнике определяется как меньшее из двух: того потока, что вошёл в вершину, и того, что может дополнительно пропустить исходящая дуга:

```
    mFlow:= Minimum(Node.mFlow, Link.mHigh - Link.mFlow);
```

Схожим образом обрабатываются *входящие* дуги текущей вершины. Если входящая дуга принадлежит белой вершине, и поток в ней существует (точнее, это поток превышает минимальный), то дуга метится как обратная:

```
    mDirect:=-1;
```

после чего белая вершина окрашивается, в ней корректируются обратные ссылки и поток:

```
            mFlow:= Minimum(Node.mFlow, Link.mFlow);
```

Вновь окрашенные вершины ставятся в очередь, и цикл обработки очереди повторяется. Цикл завершается в двух случаях: а) при извлечении из очереди вершины-стока, — тут цепь построена; б) при опустошении очереди — здесь цепочка не может быть построена. В первом случае поле **mFlow** вершины-стока содержит то приращение потока, на которое можно увеличить его вдоль обнаруженной цепи. Далее следует завершающий аккорд: двигаясь обратно от *стока* к *истоку* по расставленным обратным ссылкам, наращивается поток вдоль обнаруженной цепи. Если же цепь не обнаружена, возвращается ноль.

24.9. Испытание

Ниже представлена программа для испытания метода поиска максимального потока для случая, когда нижняя граница для потока не задана.

Листинг 24-2 — Поиск максимального потока (нижняя граница не задана)

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CFile = 'Test.txt';
```

```
var Gr : TGraph;        // граф
    S, T : TNode;       // вершины источника и стока
    Flow: integer;      // максимальный поток
begin
    Gr:= TGraphChars.Load(CFile);
    Gr.Expo;
    // Ввод данных о потоках
    (Gr as TGraphChars).LoadFlowData(CFile);
    S:= (Gr as TGraphChars).GetNode('A');
    T:= (Gr as TGraphChars).GetNode('H');
    Flow:= Gr.CalcMaxFlow0(S, T);
    Writeln('Flow= ', Flow);
    Gr.ExpoLinksData;  // Вывод информации о потоках в дугах
    Readln;
    Gr.Free;
end.
```

Для испытания взят граф, показанный на рис. 24-8, данные для его ввода представлены в следующем файле:

```
Test.txt - максимальный поток
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
8 - количество вершин
A B C D E F G H
A -> B C
B -> E F
C -> D
D -> E
E -> H
F -> G
G -> H
H ->
FLOW:
AB= 0 5
AC= 0 4
BE= 0 2
BF= 0 4
CD= 0 4
DE= 0 4
EH= 0 3
FG= 0 4
GH= 0 4
```

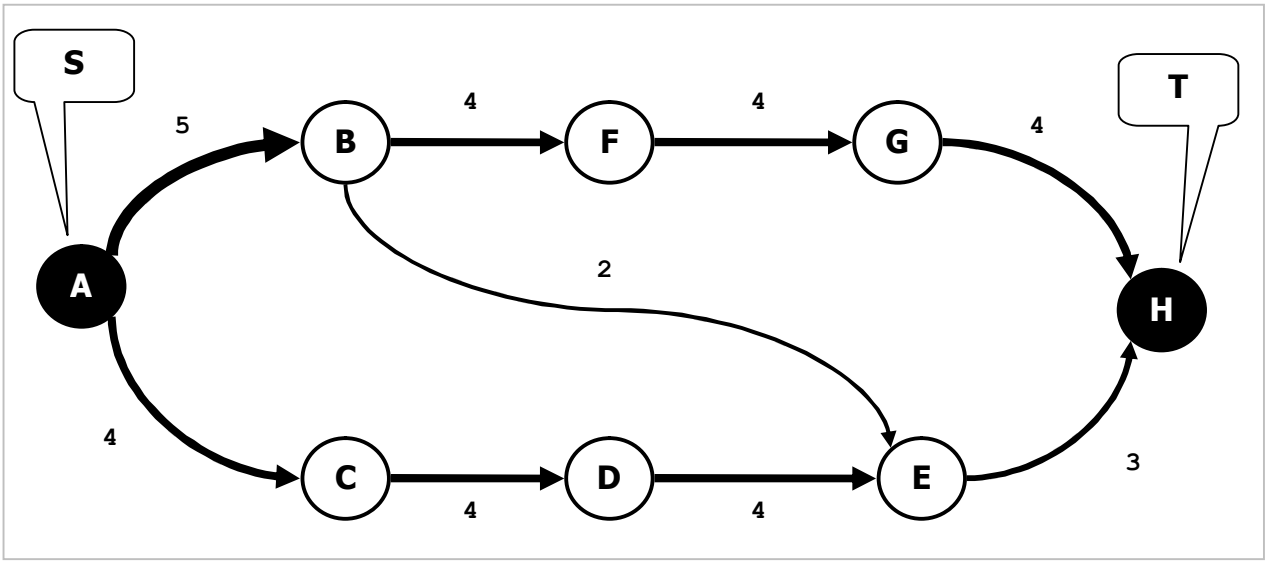


Рис. 24-8 — Граф для тестирования

Числа рядом с дугами показывают их пропускную способность. В ходе испытания метода, были последовательно найдены четыре увеличивающие цепи, показанные на следующих рисунках.

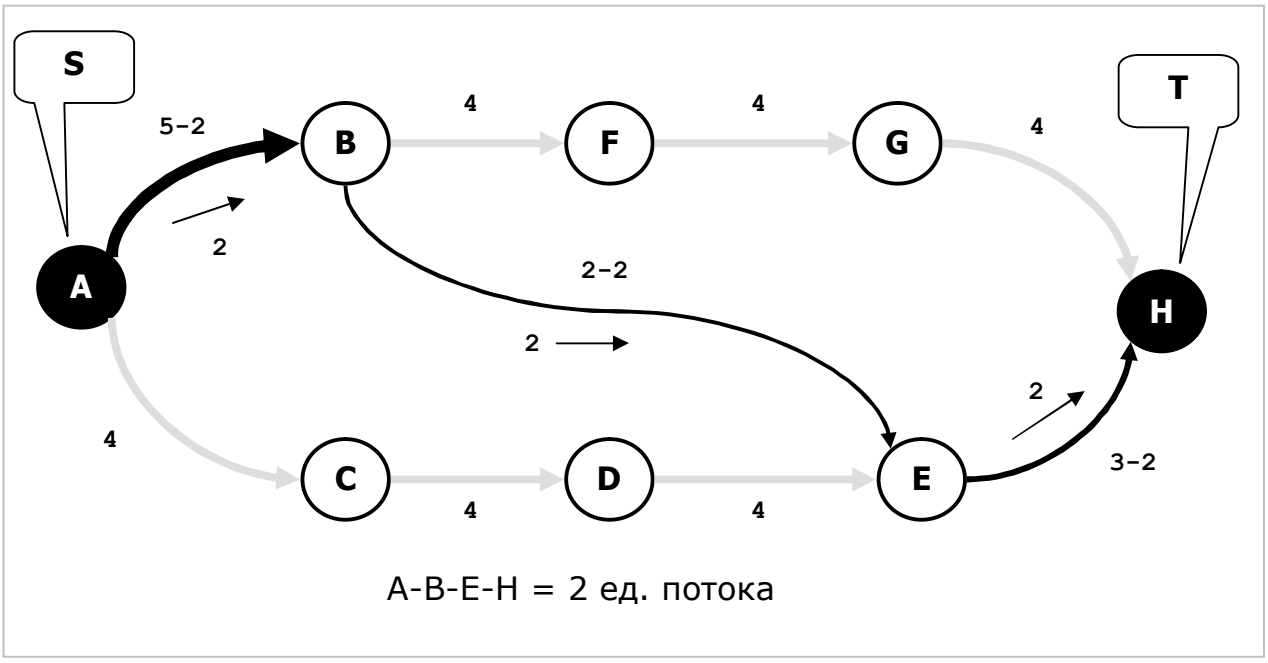


Рис. 24-9 — Пересылка двух единиц потока по цепочке A-B-E-H, насыщается дуга BE

В первой из обнаруженных цепей *A-B-E-H* насыщается прямая дуга *B-E*. Стрелки показывают направление и величину потока. Текущий запас пропускной способности дуг представлен разностями между пропускной способностью и текущим потоком.

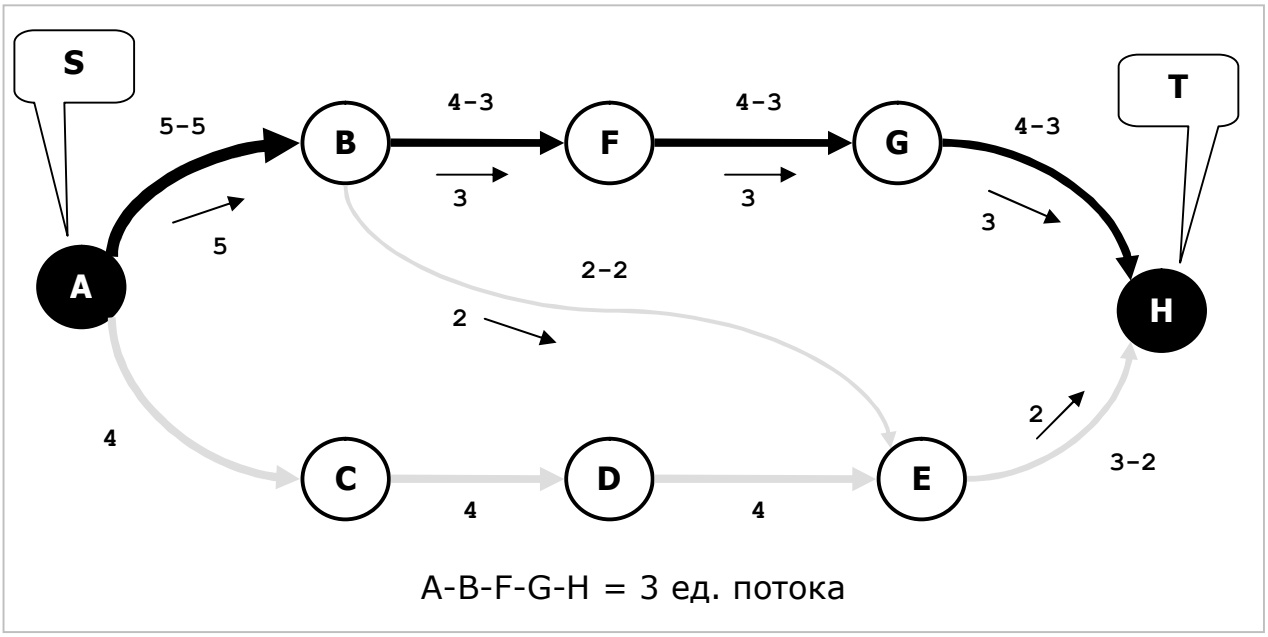


Рис. 24-10 — По цепочке A-B-F-G-H следуют три единицы потока, насыщается дуга A-B

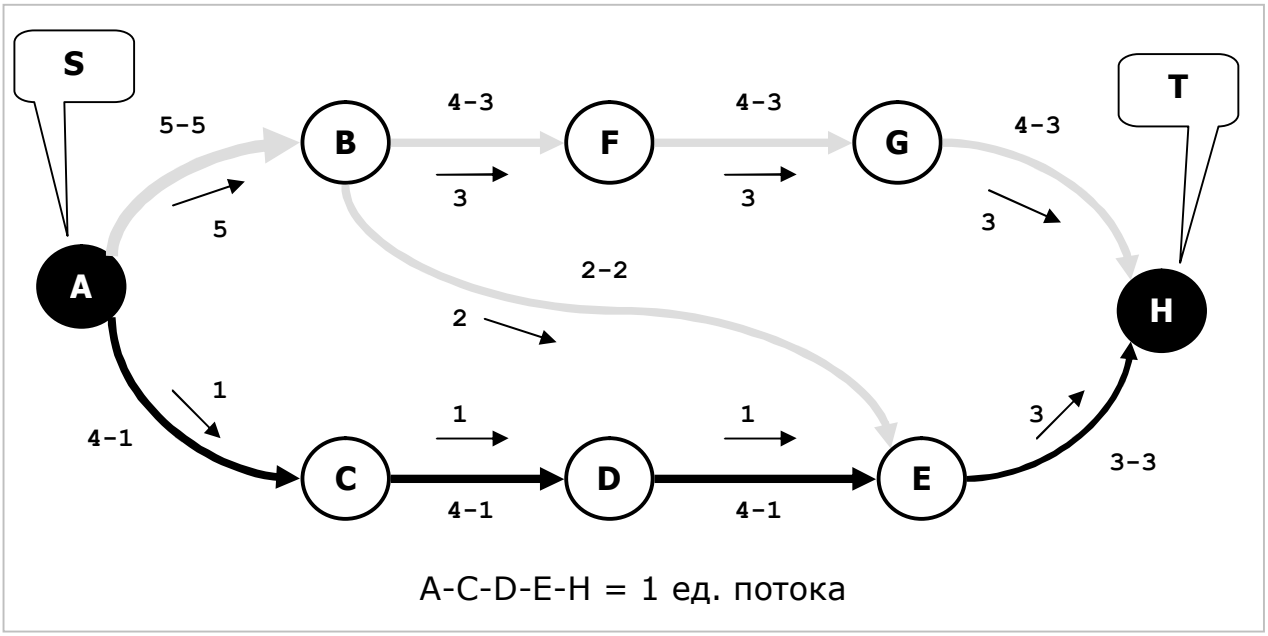


Рис. 24-11 — По цепочке A-C-D-E-H следуют три единицы потока, насыщается дуга E-H

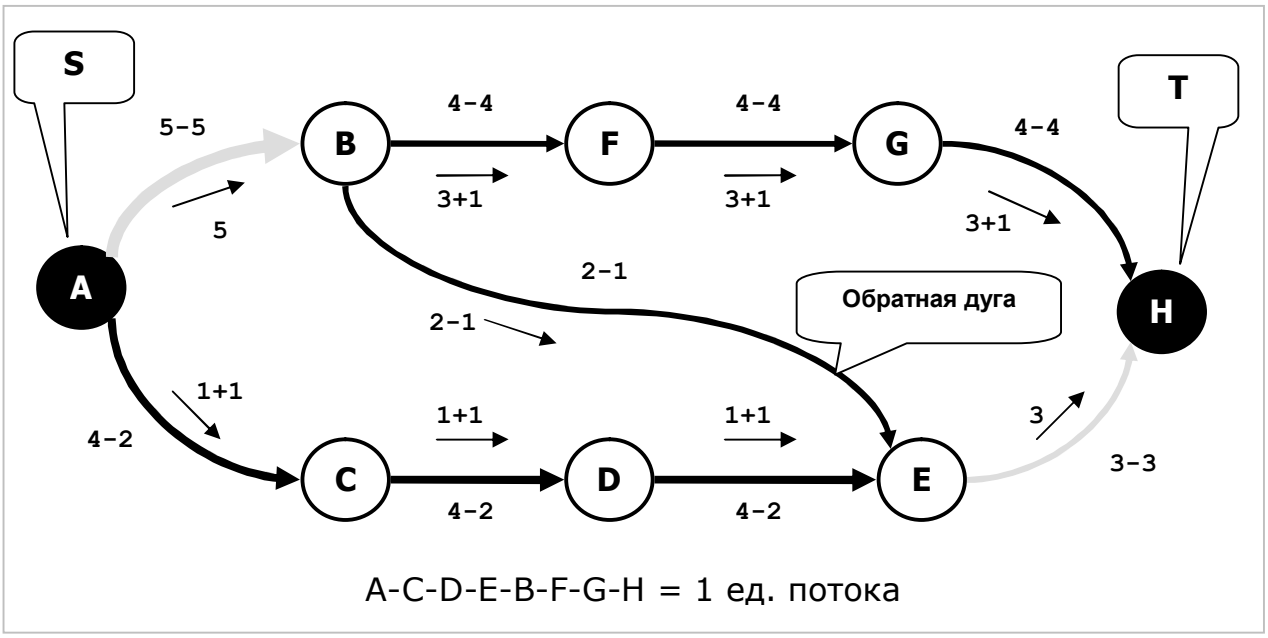


Рис. 24-12 — По цепочке A-C-D-E-B-F-G-H следуют одна единица потока, насыщаются дуги B-F, F-G, G-H

Обратите внимание на дугу *B-E*. В первой из обнаруженных цепей (рис. 24-9) она была прямой, а в последней (рис. 24-12) стала обратной. Уменьшение оттока в дуге *B-E* привело к увеличению потока вдоль цепи, но как он перераспределился? Одна единица, исходящая из вершины *B*, потекла по цепи *F-G-H*. При этом недостаток этой единицы потока в вершине *E* был компенсирован потоком из цепи *A-C-D*.

В итоге после четырёх итераций был получен максимально возможный поток в данной сети, составляющий **7** единиц.

24.10. Итоги

- Сетью называют ориентированный граф, дугам которого приписаны пропускные способности. Посредством сетей моделируют транспортировку материальных или информационных объектов.
- Основная потоковая задача состоит в поиске максимальной пропускной способности сети между истоком и стоком, и распределении потока по дугам графа с тем, чтобы достичь этого максимума.
- Принцип решения задачи состоит в поиске кратчайших увеличивающих цепей между источником и стоком с наращиванием потока вдоль этих цепей.

24.11. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 91 (пункт 4.2)
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 25
Поток, ограниченный сверху и снизу

В предыдущей главе решена базовая потоковая задача: найден *максимальный* поток в сети и его распределение по дугам. В некоторых ситуациях поток может быть ограничен не только *сверху*, но и *снизу*. Задачи этого рода порой не имеют решения, рассмотрим следующий пример.

Пусть пассажиры следуют из пункта *S* в пункт *T* с пересадкой их в промежуточном пункте *X* (рис. 25-1). На первом участке курсирует микроавтобус с вместимостью не более 11 человек, а на втором — большой автобус с вместимостью до 40 человек. Ввиду экономической целесообразности каждый транспорт имеет также минимальное ограничение на количество перевозимых людей. В этой связи можно поставить вопрос: сколько пассажиров можно перевезти между конечными пунктами так, чтобы каждый транспорт выполнил по одному рейсу?

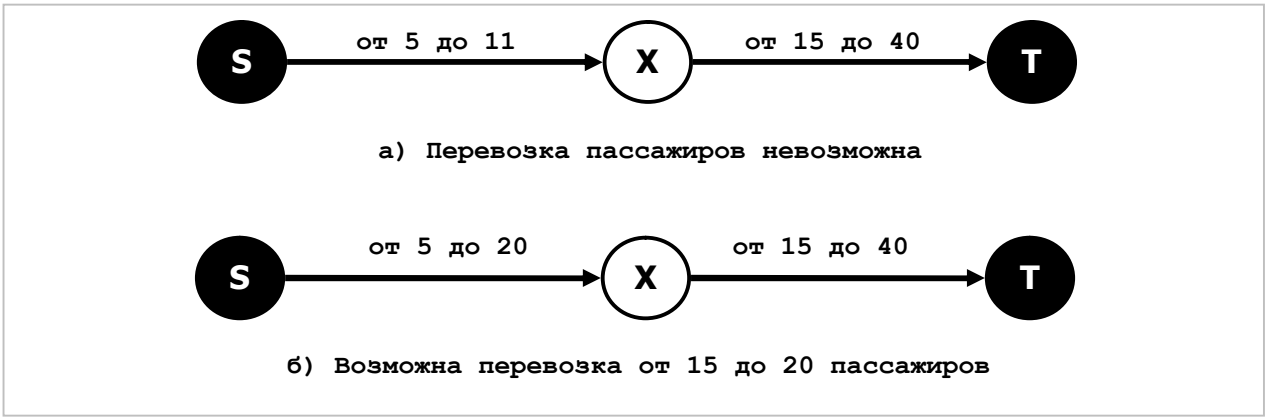


Рис. 25-1 — Примеры неосуществимых и осуществимых перевозок при разных ограничениях пропускных способностей

Очевидно, что в случае (а) такая перевозка в принципе невозможна. В случае (б) возможно перевезти от 15 до 20 пассажиров.

Итак, формулировка задачи такова: дана потоковая сеть с дугами, пропускная способность которых ограничена снизу и сверху. Необходимо найти а) минимальный поток, который можно пропустить по этой сети, и б) максимально возможный поток и его распределение по дугам.

Из приведенного примера видно, что далеко не всякое сочетание минимальных и максимальных пропускных способностей дуг даёт решение потоковой задачи. Потому поиск *максимального* потока надо предварить поиском *минимального* потока в сети (и найти его распределение по дугам). Если пропуск минимального потока осуществим, то максимальный поток можно найти ранее построенным алгоритмом, а иначе задача не имеет решения. В ходе решения

потребуются вспомогательные перестроения исходного графа с тем, чтобы воспользоваться готовыми методами, разработанными в предыдущей главе.

25.1. Эквивалентная схема

На рис. 25-2 представлен граф, дугам которого указаны (через двоеточие) нижний и верхний пределы пропускной способности. Истоком и стоком являются вершины *A* и *E* соответственно.

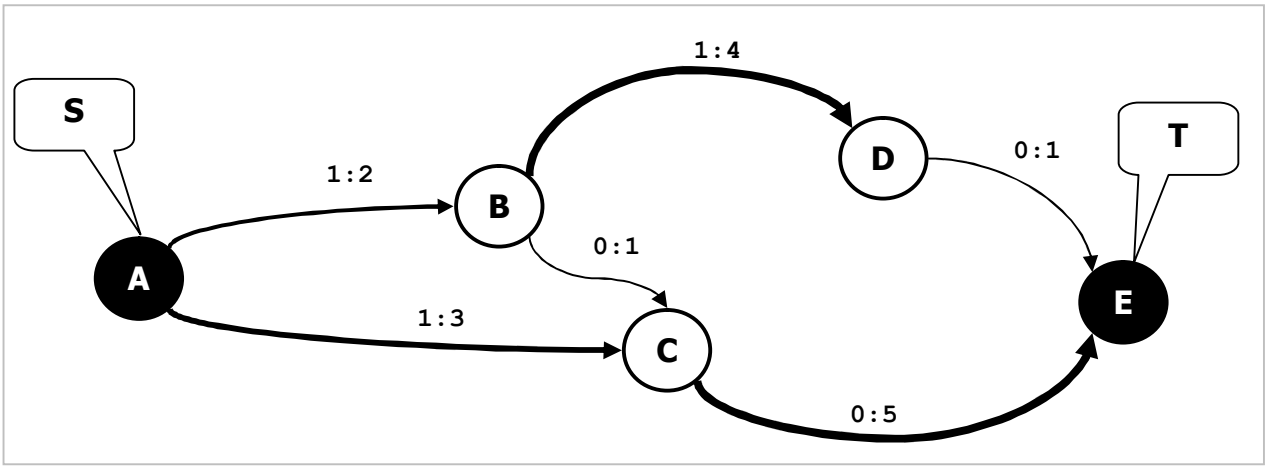


Рис. 25-2 — Граф с нижним и верхним ограничениями потока

Решим первую из поставленных задач — найдём и распределим по дугам минимальный поток. Сведём эту задачу к поиску максимального потока. Суть идеи в том, чтобы отделить «собственный» минимальный поток каждой дуги от минимальных потоков в других дугах. Для этого посредством вспомогательных построений исходный граф преобразуем в его эквивалент.

Возьмём дугу графа *X-Y* с нижней **L** и верхней **H** границами пропускной способности (рис. 25-3).

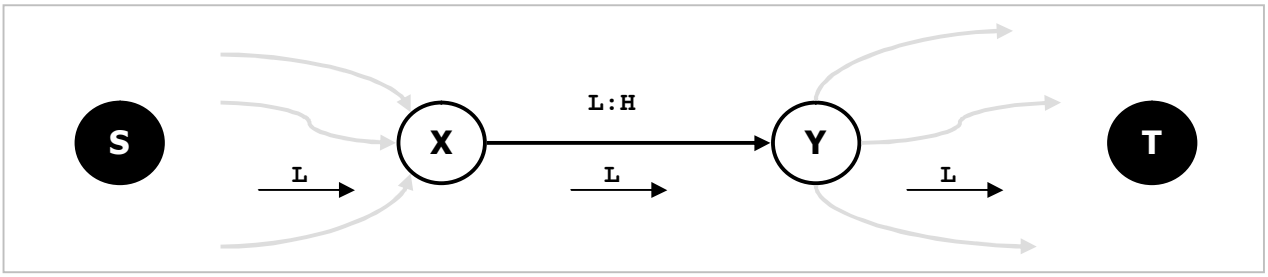


Рис. 25-3 — Дуга исходного графа с нижним (L) и верхним (H) пределами пропускной способности

Резерв пропускной способности дуги, вычисляемый как разность **D= H-L**, можно трактовать как верхний предел пропускной способности дуги, используемый для пропуска минимальных потоков *других* дуг. Понятно, что если минимальный поток в сети осуществим, то через каждую дугу должен протекать,

по меньшей мере, её собственный минимальный поток **L**. Поток может быть и несколько большим, но в пределах пропускной способности **H**. Этот поток **L** должен сначала пройти от источника *S* к началу дуги *X*, а затем от конца дуги *Y* к стоку *T*, пользуясь, если необходимо, *резервами* **D** в других дугах. Если такое продвижение осуществимо, то минимальный поток в графе возможен, и его можно найти.

Отделим «свой» минимальный поток дуги от потоков «чужих» дуг, и направим «свой» минимальный поток в «кругосветное путешествие» по чужим дугам. Для этого учредим вспомогательные исток *Sx* и сток *Tx*, а также соединим обратной дугой с неограниченной пропускной способностью сток *T* с истоком *S* сети (рис. 25-4). Теперь «свой» поток дуги *X-Y* пустим не по этой дуге, а окружным путём из истока *Sx* в сток *Tx* :

Sx --> Y ...-->... T --> S ...-->... X --> Tx

При этом пропускную способность дуги *X-Y* снизим до величины **D = H-L**.

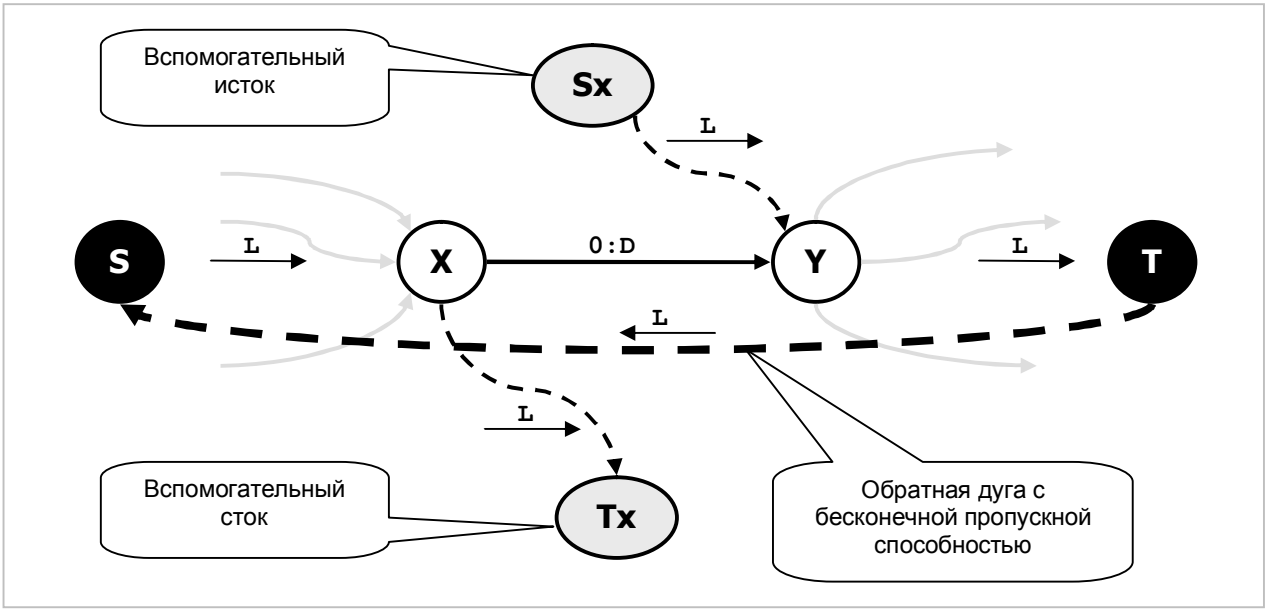


Рис. 25-4 — «Кругосветное путешествие» минимального потока L дуги X

Отделив подобным образом для всех дуг «свои» потоки от «чужих», получим новый граф с другими истоком и стоком, а также с дугами, *минимальная* пропускная способность которых уже не ограничена (равна нулю). Так задача о *минимальном* потоке свелась к уже решённой задаче о *максимальном* потоке.

Вновь введённым вспомогательным дугам (указаны пунктиром) назначим максимальные пропускные способности, равные **L**, после чего построим максимальный поток от истока *Sx* к стоку *Tx*. Затем проверим насыщенность вспомогательных дуг, для чего сравним сумму их пропускных способностей с суммой проходящих через них потоков. Если вспомогательные дуги окажутся насыщенными, то, все минимальные потоки успешно преодолели «кругосветку»,

наполнив каждую дугу по меньшей мере минимальным для неё потоком. Иное будет означать неспособность сети пропустить минимальный поток.

25.2. Дело техники

Обратимся к реализации вспомогательных дуг. По нашей задумке, каждую вершину надо связать со вспомогательным истоком *Sx* и стоком *Tx* таким количеством дуг, сколько их имеется у этой вершины. Однако в простом графе это невыполнимо. Поступим просто: совокупность из нескольких параллельных дуг заменим одной дугой с суммарной пропускной способностью этих дуг. Мало того: пару дуг от *истока* к вершине, и от вершины к *стоку* заменим одной дугой, а то и вовсе их отбросим, если совокупный поток от *истока* совпадает с таким же для *стока*. Поясним это на числовом примере и следующих рисунках.

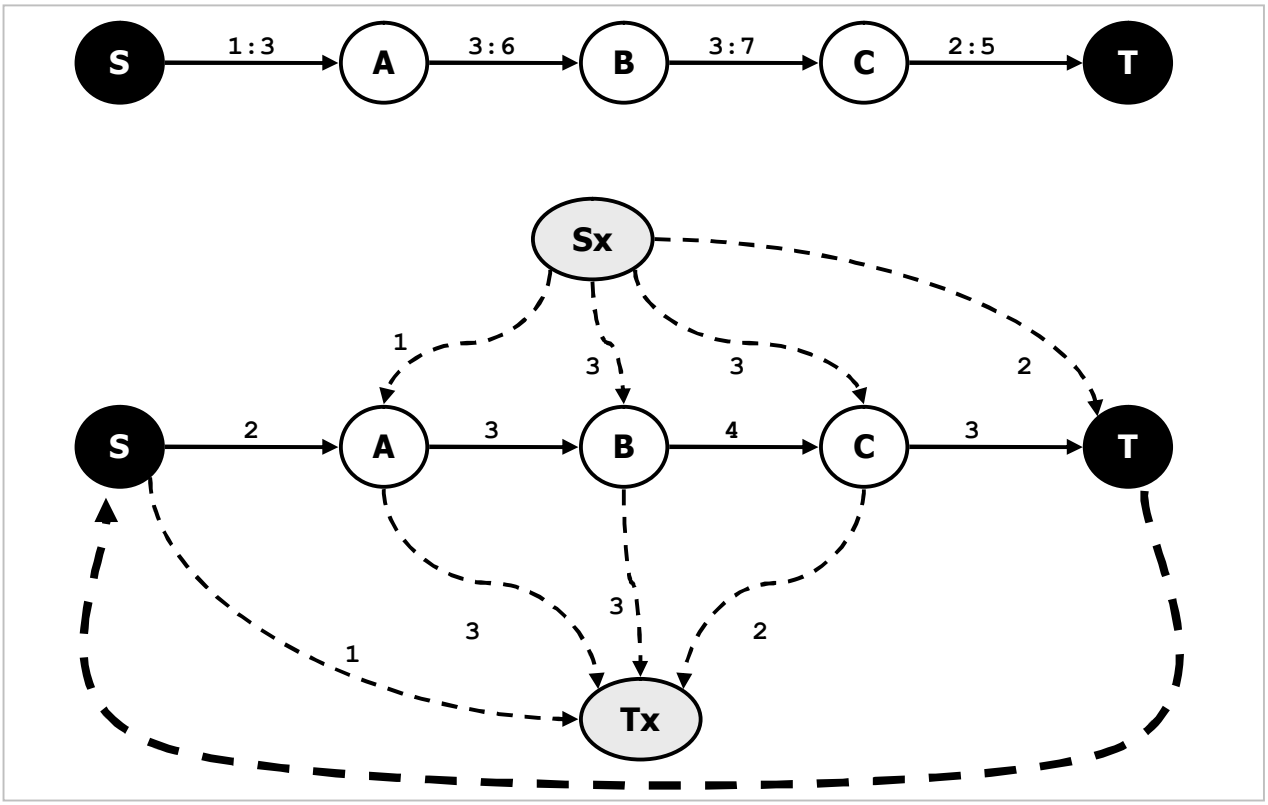


Рис. 25-5 — Исходная и вспомогательная сети

На рис. 25-5 вверху показана исходная сеть, где для всех дуг заданы минимальные и максимальны пределы, а ниже — соответствующая вспомогательная сеть. Видно, что часть потока пройдёт от *Sx* до *Tx* кратчайшими маршрутами, минуя все прочие дуги, и не влияя на потоки в них. Следовательно, эту часть потока можно отбросить, то есть: удалить дуги *Sx*→*B* и *B*→*Tx*, а вместо пары дуг *Sx*→*A* и *A*→*Tx* оставить дугу *A*→*Tx* с пропускной способностью **3-1=2**. Аналогично пару дуг *Sx*→*C* и *C*→*Tx* заменим одной дугой *Sx*→*C* с пропускной способностью **3-2=1**. Результат этих сокращений показан на рис. 25-6.

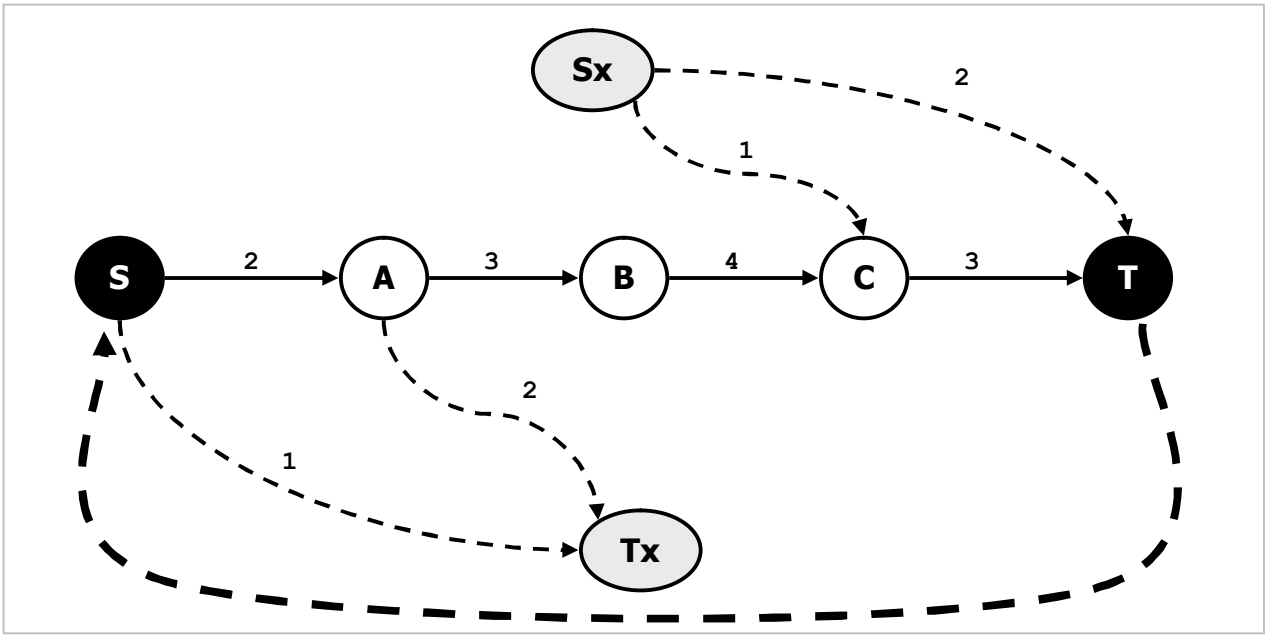


Рис. 25-6 — Сокращённая вспомогательная сеть

25.3. Алгоритм

Подведём промежуточный итог рассуждений и дадим описание алгоритма поиска *минимального* потока, реализованного в методе **`CalcMinFlow`**.

- Сток сети соединим с истоком обратной дугой с неограниченной пропускной способностью.
- Добавим в сеть две вспомогательные вершины: исток *Sx* и сток *Tx*.
- Для каждой исходной вершины определим разность между суммой минимальных потоков исходящих дуг и суммой входящих. При разности больше нуля, соединим вершину с вспомогательным стоком *Tx* дугой, пропускная способность которой равна этой разности. Если разность отрицательна, то соединим вспомогательный исток *Sx* с этой вершиной дугой с пропускной способностью, равной этой разности по модулю. При нулевой разности дополнительных дуг не создаём.
- Нижний предел пропускной способности всех исходных дуг графа назначим равным нулю, а верхний — разности **`D=H-L`**.
- Построим максимальный поток между вспомогательным истоком и стоком.
- Исследуем насыщенность вспомогательных дуг. Если все они насыщены, то минимальный поток в итоге построен и распределён по исходным дугам графа, а иначе пропуск минимального потока невозможен.
- Удалим вспомогательные дуги и вершины.

Метод вычисления минимального потока представлен ниже.

Листинг 25-1 — Вычисление минимального потока

```
function TGraph.CalcMinFlow(aSource, aDest: TNode): integer;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Восстановление данных о потоке после удаления искусственных вершин и дуг
  // arg = false -- минимальный поток не существует
  // arg = true  -- минимальный поток найден

  procedure RestoreFlowData(arg: boolean);
  var Node: TNode;
      Link: TLink;      // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        // Перебор исходящих линков вершины:
        Link:= OutLinkFirst;
        while Assigned(Link) do begin
          with Link do begin
            mLow:= mTemp;            // минимум
            mHigh:= mHigh + mLow;    // максимум
            // текущий минимальный поток:
            if arg
              then mFlow:= mFlow + mLow  // в случае успеха
              else mFlow:= 0;            // если поток не существует
          end; // with
          Link:= OutLinkNext;
        end; // while
      end; // with
      Node:= NodeNext;
    end; // while
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var Node: TNode;       // текущая вершина
    Link: TLink;       // текущий линк
    Reverse: TLink;    // обратный линк T->S во вспомогательном графе
    Flow: integer;     // максимальный поток
    SumLow: integer;   // сумма всех минимумов
    Sx: TNode;         // вспомогательный исток
    Tx: TNode;         // вспомогательный сток
    OldLink: TLink;    // линк T->S в исходном графе
    OldLow, OldHigh: integer; // сохранённые данные обратного линка

begin    { TGraph.CalcMinFlow }

  // Если не указаны вершины, то выход
  if not Assigned(aSource) or not Assigned(aDest) then begin
    Result:=-1;
    Exit;
  end;

  // Создаём:
  Sx:= TNode.Create(0, Self); // вспомогательный исток
  Tx:= TNode.Create(0, Self); // вспомогательный сток
  SumLow:=0;     // здесь накапливаем сумму всех минимальных потоков
  // Первый перебор вершин с подсчётом сумм исходящих минимальных потоков:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mFlow:=0; // сумма исходящих минимумов
    with Node do begin
```

```
      // Перебор исходящих линков вершины:
      Link:= OutLinkFirst;
      while Assigned(Link) do begin
        with Link do begin
          Inc(Node.mFlow, mLow);  // сумма исходящих минимумов
          mHigh:= mHigh - mLow;   // пересчёт максимума
          mTemp:= mLow;           // временно сохраняем минимум
          mLow:= 0;               // и обнуляем его
          mFlow:=0;               // обнуляем поток
        end;
        Link:= OutLinkNext;
      end;
    end; // with
    Node:= NodeNext;
  end; // while

  // Второй перебор вершин с подсчётом сумм входящих минимальных потоков:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      // Перебор входящих линков вершины:
      Link:= InLinkFirst;
      while Assigned(Link) do begin
        Dec(Node.mFlow, Link.mTemp); // минус сумма исходящих минимумов
        Link:= InLinkNext;
      end;
    end;
    // Если вершина является источником или стоком:
    if Node.mFlow <> 0 then begin
      // Если вершиная является источником:
      if Node.mFlow > 0 then begin
        // создаём дугу из вершины во вспомогательный сток
        Link:= Node.MakeLink(Tx, 0);  // создаём линк
        Link.mHigh:= Node.mFlow;      // с данными о потоке
      end else begin
        // создаём дугу из вспомогательного истока в вершину
        Link:= Sx.MakeLink(Node, 0);  // создаём линк
        Link.mHigh:= -Node.mFlow;      // с данными о потоке
      end
    end;
    Inc(SumLow, Abs(Node.mFlow)); // двойная сумма всех минимальных потоков
    Node:= NodeNext;
  end; // while
  SumLow:= SumLow div 2;  // одинарная сумма всех минимальных потоков

  // Если сумма минимальных потоков = 0, то выходим из процедуры
  if SumLow=0 then begin
    // Освобождаем:
    Sx.Free;     // вспомогательный исток
    Tx.Free;     // вспомогательный сток
    // Восстанавливаем данные о потоках:
    RestoreFlowData(false);
    Result:=0;
    Exit;
  end;

  // Вставляем вспомогательные вершины в граф:
  mNodes.Insert(Sx);   // исток
  mNodes.Insert(Tx);   // сток

  // Соединяем обратной дугой сток с истоком
  // (поток в обратной дуге не ограничен):
```

```
  // На случай отсутствия обратной дуги:
  OldLow:= 0; OldHigh:=0;

  // Обратная дуга уже существует?
  OldLink:= aDest.GetLink(aSource);
  if Assigned(OldLink) then begin
    // да, назначаем её в качестве реверсивной и запоминаем параметры
    Reverse:= OldLink;    // реверсивная дуга
    with OldLink do begin
      // да, запоминаем данные потока
      OldLow := mLow;     // нижний предел
      OldHigh:= mHigh;    // верхний предел
      // и устанавливаем новые
      mLow := 0;          // нижний предел отсутствует
      mHigh:= MaxInt;     // неограниченная пропускная способность
    end // with
  end else begin
    // здесь обратной дуги нет, поэтому создаём новую дугу
    Reverse:= aDest.MakeLink(aSource, 0); // дуга
    Reverse.mHigh:= MaxInt;   // с максимальной пропускной способностью
  end;

  // Во вновь построенном вспомогательном графе
  // вычисляем максимальный поток между вспомогательными истоком и стоком

  Flow:= CalcMaxFlow0(Sx, Tx);

  // После этого минимальный поток протекает через реверсную дугу:
  Result:= Reverse.mFlow;  // запоминаем
  Reverse.mFlow:= 0;       // и обнуляем

  // Удаляем обратную дугу, или восстанавливаем предыдущую, если она была

  // Обратная дуга существовала?
  if Assigned(OldLink) then with OldLink do begin
    // да восстанавливаем
    mLow := OldLow;
    mHigh:= OldHigh;
  end else begin
    // Дуга в исходном графе не существовала,
    // разрываем обратную связь между стоком и истоком
    aDest.RemoveLink(aSource);
  end;

  // Удаляем из графа и освобождаем:
  RemoveNode(Sx);  Sx.Free;     // вспомогательный исток
  RemoveNode(Tx);  Tx.Free;     // вспомогательный сток

  // Если вспомогательный максимальный поток (Sum)
  // равен сумме минимальных потоков (SumLow)
  // то минимальный поток в исходном графе существует

  if Flow = SumLow then begin
    // вспомогательные дуги насыщены
    RestoreFlowData(true); // формируем новые данные о потоках
  end else begin
    // вспомогательные дуги НЕ насыщены
    Result:=-1;             // минимального потока не существует
    RestoreFlowData(false); // восстанавливаем прежние данные о потоках
  end;
end;
```

Два слова об обратной дуге от стока к истоку. Предполагается, что в исходной сети такая дуга может как существовать, так и отсутствовать. В первом случае вначале запоминаются параметры этой дуги, а перед выходом из метода они восстанавливаются.

25.4. Метод вычисления максимального потока

Решив вопрос с минимальным потоком, найдём теперь *максимальный* поток в сети для рассматриваемого случая, когда заданы нижние пределы пропускной способности дуг. При *нулевых* нижних границах он вычислялся последовательным наращиванием потока вдоль увеличивающих цепей, начиная с *нулевого* потока. Здесь поступим точно также, однако начальным будет не нулевой, а уже вычисленный *минимальный* поток. Представленный ниже метод получился крайне простым, поскольку использует уже готовые процедуры.

Листинг 25-2 — Вычисление максимального потока для ненулевых нижних границ

```
function TGraph.CalcMaxFlow(aSource, aDest: TNode): integer;
begin
  // Сначала вычисляем минимальный поток:
  Result:= CalcMinFlow(aSource, aDest);
  // Если он не существует, то не существует и максимального:
  if Result < 0 then Exit;
  // После формирования минимального потока формируем максимальный:
  Result:= CalcMaxFlow0(aSource, aDest);
end;
```

Здесь уместно вспомнить о локальной функции **CalcStartFlow**, расположенной в методе **CalcMaxFlow0**. Теперь должно быть ясно, зачем она вычисляет начальное значение потока, текущего по дугам сети при старте алгоритма.

25.5. Испытание

Для испытаний методов вычисления минимальных и максимальных потоков воспользуемся следующей программой.

Листинг 25-3 — Программа для испытания потоковых методов

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CFile = 'Test.txt';
var Gr : TGraph;
```

```
    S, T : TNode;
    Flow: integer;
begin
    Gr:= TGraphChars.Load(CFile);
    (Gr as TGraphChars).LoadFlowData(CFile);
    S:= (Gr as TGraphChars).GetNode('A');
    T:= (Gr as TGraphChars).GetNode('E');
    Flow:= Gr.CalcMinFlow(S, T);
    Writeln('MinFlow= ', Flow);
    Gr.ExpoLinksData;
    Writeln('================');
    Flow:= Gr.CalcMaxFlow(S, T);
    Writeln('MaxFlow= ', Flow);
    Gr.ExpoLinksData;
    Readln;
    Gr.Free;
end.
```

Ниже представлены данные для ввода испытательного графа и сам этот граф.

```
Test.txt - минимальный и максимальный поток
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
5 - количество вершин
A B C D E
A -> B C
B -> C D
C -> E
D -> E
E ->
FLOW:
AB= 1 2
AC= 1 3
BC= 0 1
BD= 1 4
CE= 0 5
DE= 0 1
```

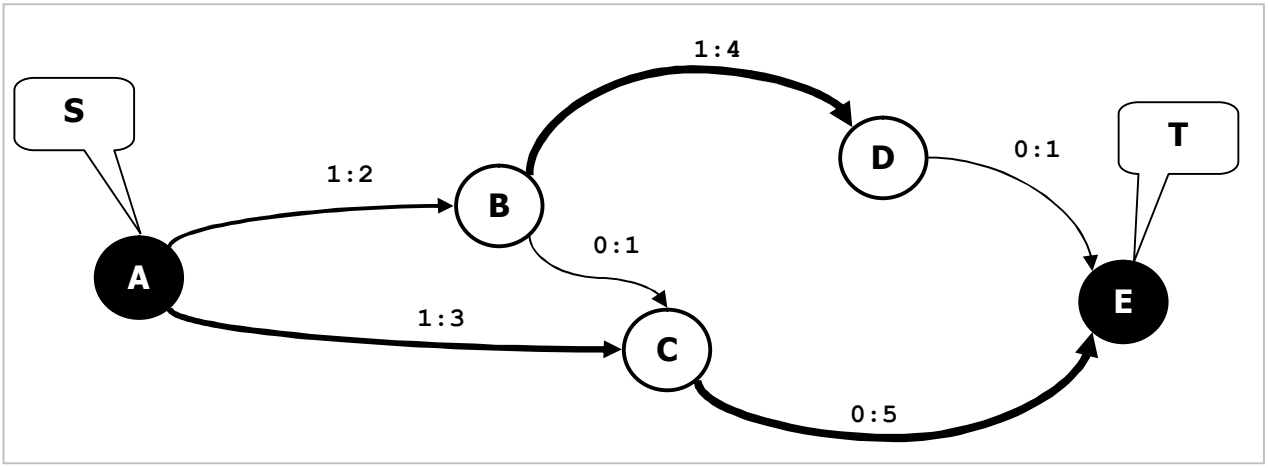


Рис. 25-7 — Граф с ненулевыми нижними границами

На рис. 25-1 представлена вспомогательная сеть, максимальный поток в которой составляет 2 единицы, а вспомогательные дуги насыщены.

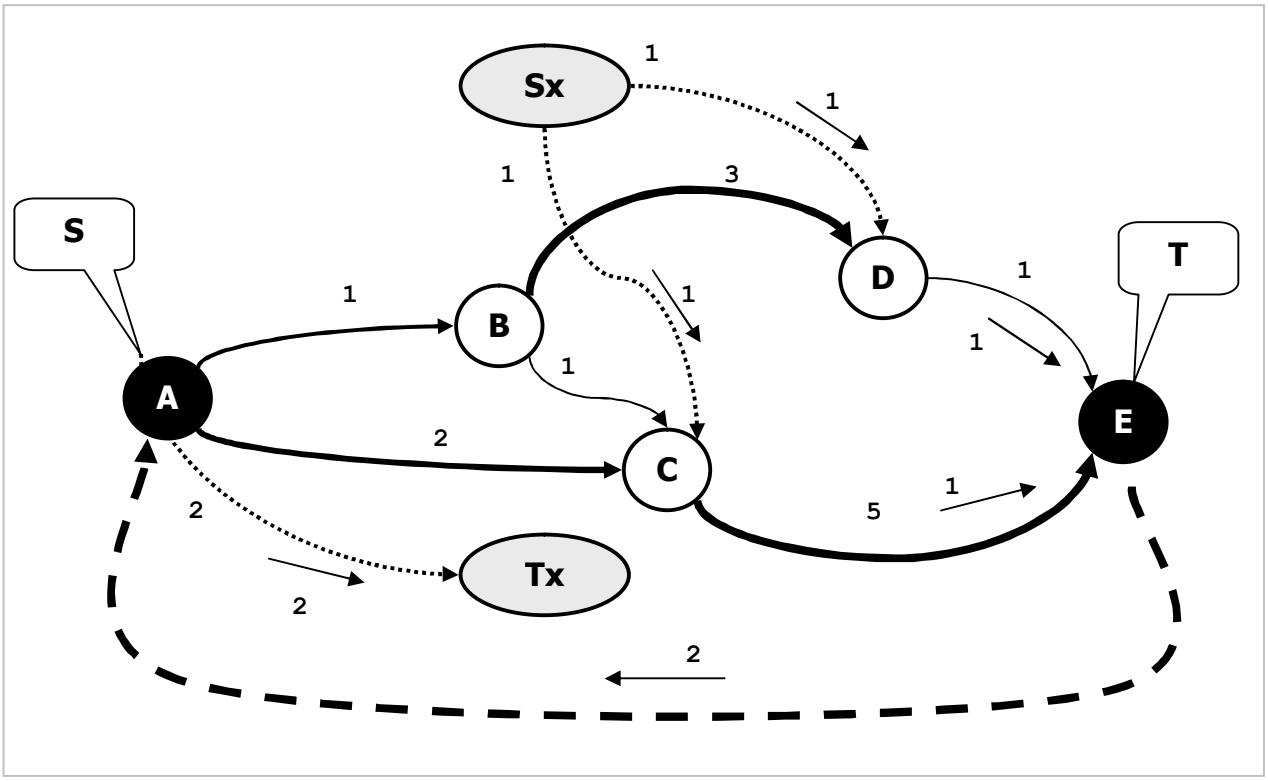


Рис. 25-8 — Вспомогательная сеть,
максимальный поток в которой составляет 2 единицы

На следующих рисунках дано распределение минимального и максимального потока по дугам графа.

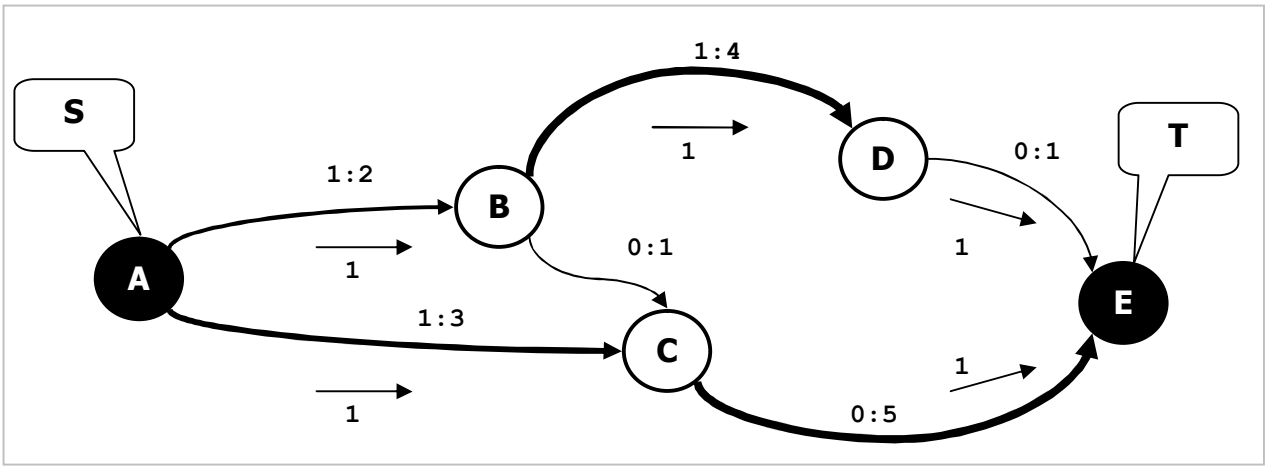


Рис. 25-9 — Минимальный поток в сети составляет 2 единицы

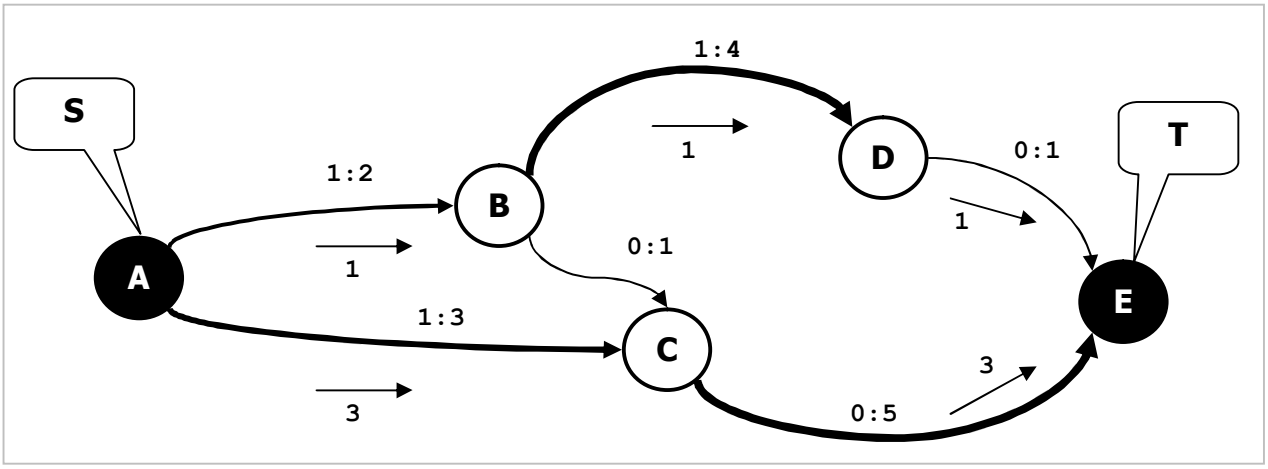


Рис. 25-10 — Максимальный поток в сети составляет 4 единицы

25.6. Итоги

- Ограничение на нижнюю и верхнюю пропускную способность дуг может повлечь невозможность пропуска потока в сети.
- Для поиска минимального потока в сети собственные минимальные потоки дуг отделяются от «чужих» минимальных потоков и соответственно уменьшаются их верхние пропускные способности. Путём вспомогательных преобразований строится новая сеть, на которой решается задача поиска максимального потока. Так поиск минимального потока сводится к поиску максимального.
- Нахождение минимального потока в сети (если он возможен), выполняется уже известным способом поиска максимального потока.
- Максимальный поток в сети ищется после успешного нахождения минимального потока уже известным способом.

25.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
√	2	Басакер Р. Саати Т.	Конечные графы и сети	Глава 7.8 стр. 327
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
	7	Кристофидес Н.	Теория графов. Алгоритмический подход	
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 26
Минимальная стоимость потока

26.1. Формулировка задачи

В главе 24 был дан пример с потоком машин, текущим по городским улицам. Пусть известно количество автомобилей, проезжающих город в единицу времени (поток), а также длина улиц между перекрёстками (дуг). И пусть стоимость проезда по дуге пропорциональна её длине. Отсюда следуют два вопроса: 1) как лучше распределить поток по улицам с целью его удешевления? и 2) какова будет минимальная стоимость этого потока?

Итак, задача о *минимальной стоимости потока* формулируется следующим образом. Дана сеть, где каждой дуге приписана пропускная способность и *цена* пересылки по ней единицы потока. Известен также *общий* поток в этой сети, и он не превышает предельно возможного. Надо найти минимальную *стоимость* пересылки этого потока и соответствующее *распределение* потока по дугам сети. Эту задачу называют ещё задачей о потоке *минимальной стоимости*.

26.2. Алгоритм

Принцип решения этой задачи схож с поиском максимального потока, рассмотренного в главе 24. Напомню, что там последовательно находили увеличивающие цепи дуг между истоком и стоком сети, начиная с кратчайшей цепи. Длину цепи трактовали как число дуг, её составляющих. Здесь поступим аналогично, но длину цепи будем трактовать как её стоимость. Таким образом, будем последовательно отыскивать наиболее *дешёвые* цепи. Стоимость цепи равна сумме цен составляющих её дуг (для цены дуги назначено поле **`TLink.mValue`**). В целом алгоритм решения задачи таков:

1. Начальный поток **`F`** и стоимость его транспортировки **`C`** принимаем равными нулю (**`F=0`**, **`C=0`**) .
2. «Покупаем», по крайней мере, одну, наименее дорогую увеличивающую цепь, и находим цену пересылки по ней единицы потока **`C1`**. Если ни одной цепи «купить» не удаётся, то возвращаем отрицательный результат — переслать заданный поток невозможно.
3. Находим допустимое приращение потока, текущего вдоль «купленной» цепи по цене **`C1`**. Если приращение потока невозможно, переходим к пункту 2, а иначе — к пункту 4.
4. Наращиваем поток **`F`**, распределяя его по дугам, а также стоимость его транспортировки. Если поток достиг заданного значения, то процесс завершается, а иначе переходим к пункту 3.

Итак, начинаем с «покупки» хотя бы одной наименее дорогой увеличивающей цепи (или нескольких равноценных). При отсутствии таких цепей задача не решается. Затем находим одно или несколько приращений потока вдоль найденных

цепей, и корректируем текущий *поток* и его *стоимость*. Если после приращения потока он достиг заданного значения, завершаем процесс, а иначе «покупаем» следующую, более дорогую, цепь. Определять приращение потока согласно п.3 будем почти так же, как делали это в задаче поиска максимального потока, и потому главное внимание уделим «покупке» цепей.

26.3. Торг уместен

Под «покупкой» понимается цена, выплачиваемая за пересылку единицы потока от истока к некоторой промежуточной, либо конечной вершине (стоку). Это похоже на покупку билета в пригородный поезд, цена которого зависит только от расстояния, но не от направления поездки. В сети может оказаться две или более увеличивающих цепей с одинаковой ценой **C**, и тогда по этой цене «приобретаются» сразу все эти цепочки. Стоимость проходящего по ним потока будет равна величине суммарного потока в «купленных» цепях, умноженной на цену одной цепочки.

Приняв это условное понимание покупки (далее упоминаем её без кавычек), обратимся к процессу, более похожему на аукцион. Речь идёт об оценке увеличивающих цепей. Рассмотрим аукцион на примере сети, представленной на рис. 26-1, где рядом с дугами показаны их цены (здесь цена отчасти соотносится с длиной дуги).

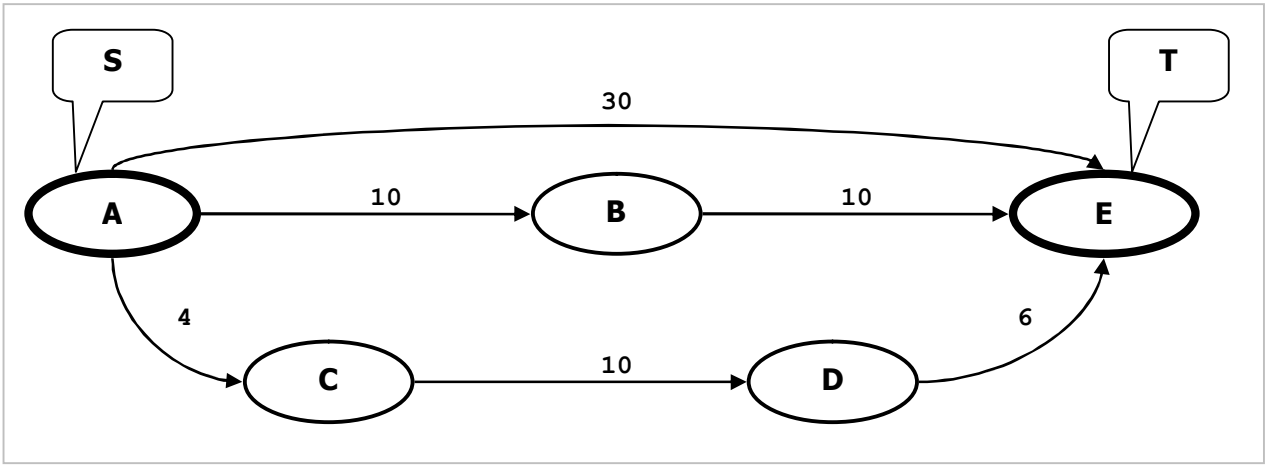


Рис. 26-1 — Сеть, где дугами назначены цены

Если бы мы искали *максимальный* поток без учёта цены, то первой была бы выбрана цепь *A-E*, затем *A-B-E* и далее *A-C-D-E*. Но теперь всё иначе: мы вынуждены платить за цепочки, а первая из них отнюдь не самая дешёвая.

В ходе покупки будем постепенно поднимать цену (аукцион!), отмечая по ходу потенциально купленные участки цепи, так называемыми *вершинными числами*, — об их формировании и применении сказано ниже.

На рис. 26-2 показано исходное состояние вершинных чисел перед началом аукциона (для этих чисел отведено поле **TNode.mDist**). Приобретённые участки

будем окрашивать серым, исток сети — это уже приобретённая вершина, и потому окрашен.

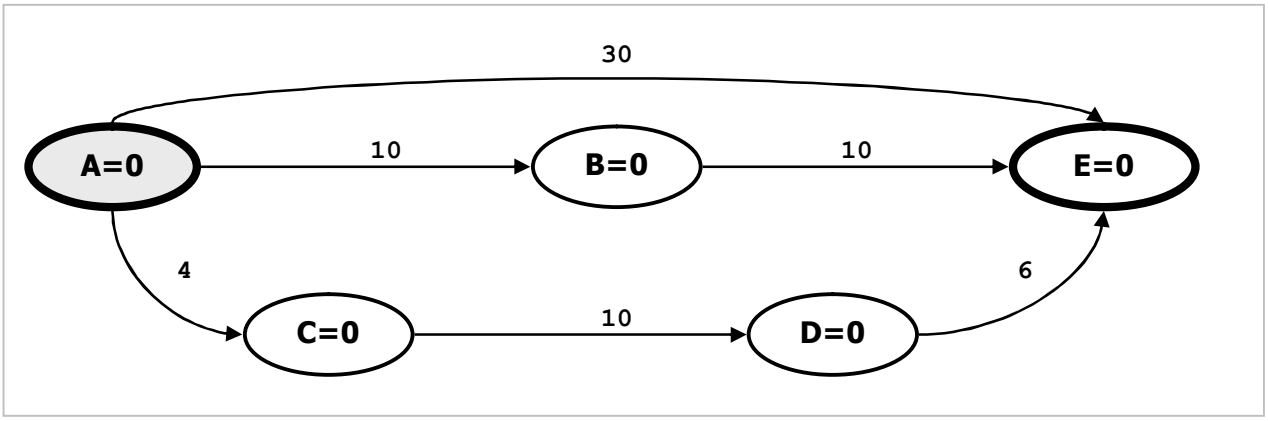


Рис. 26-2 — Исходные значения вершинных чисел

Будем постепенно набавлять цену: **1**, **2**, **3**... монеты, всякий раз наращивая вершинные числа *только в белых* вершинах. И после каждой прибавки проверять разность вершинных чисел соседних *разноцветных* вершин. Как только эта разность *сровняется* с ценой соединяющей их дуги, белая вершина окрашивается серым, и вместе с соединяющей дугой считается *приобретённой*. Поскольку купленная вершина окрашена серым, её вершинное число далее *не наращивается*. Процесс завершается приобретением стока сети *T*, и тогда вершинное число в стоке укажет цену всей приобретённой цепи (или нескольких равноценных).

На торг выставляют только *прямые* дуги с *ненасыщенным* потоком (дуги класса *I*), — *нейтральные* дуги *не рассматриваются*. Если посредством дуг класса *I* не удаётся купить увеличивающую цепь, процесс завершается с отрицательным результатом.

На рис. 26-3 показано состояние вершинных чисел в момент приобретения за **4** монеты дуги *A-C*. Обратите внимание: здесь, наряду с покупкой дуги *A-C* одновременно куплены части дуг *A-E* и *A-B*. Эта частичная оплата будет зачтена в ходе последующих торгов.

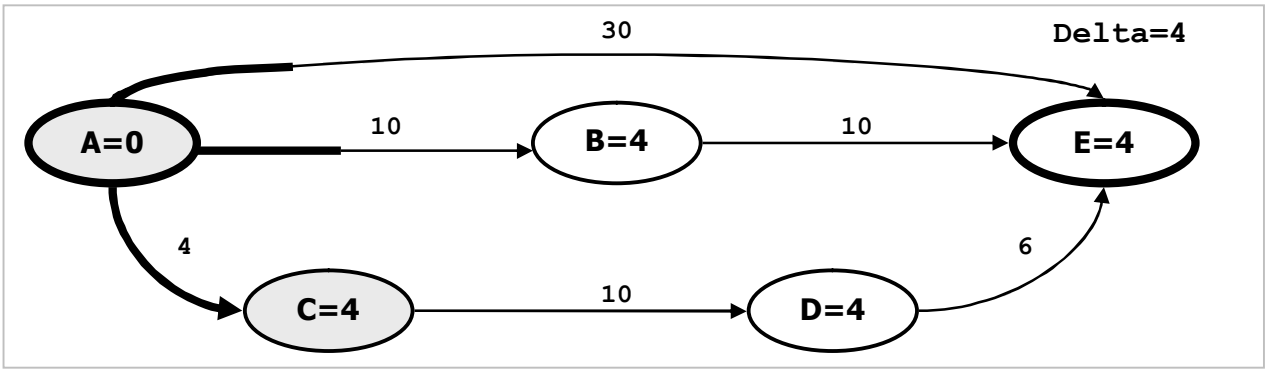


Рис. 26-3 — Участки сети, купленные за 4 монеты: приобретена дуга A-C

Осторожное набавление цены с единичным шагом — не самый быстрый способ торга. По разности чисел в соседних разноцветных вершинах и цене соединяющей их дуги можно сразу определить минимальную прибавку, необходимую для покупки очередной дуги. Перебрав все дуги, соединяющие серые и белые вершины, и определив минимальную из прибавок, можно сразу предложить нужную сумму и купить одну из этих дуг. Формула для вычисления прибавки очень проста:

```
Delta = Cost - (G - W)
```

где:

- **Delta** — прибавка, необходимая для приобретения дуги;
- **Cost** — цена приобретаемой дуги;
- **G**, **W** — вершинные числа соответственно в серой и белой вершинах.

На последующих рисунках показаны минимальные прибавки и приобретённые участки сети (здесь все приобретаемые дуги — увеличивающие).

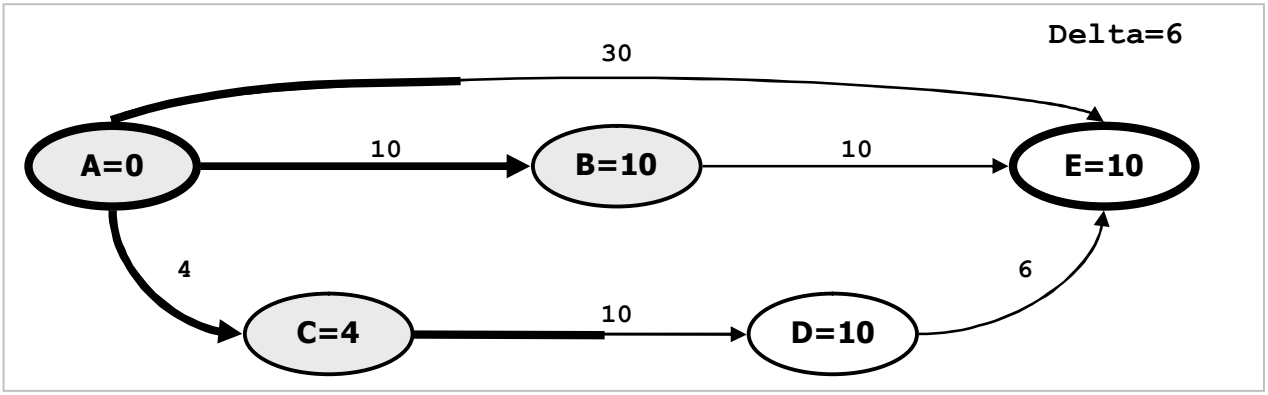


Рис. 26-4 — Добавлено 6 монет (всего 10), приобретена дуга A-B

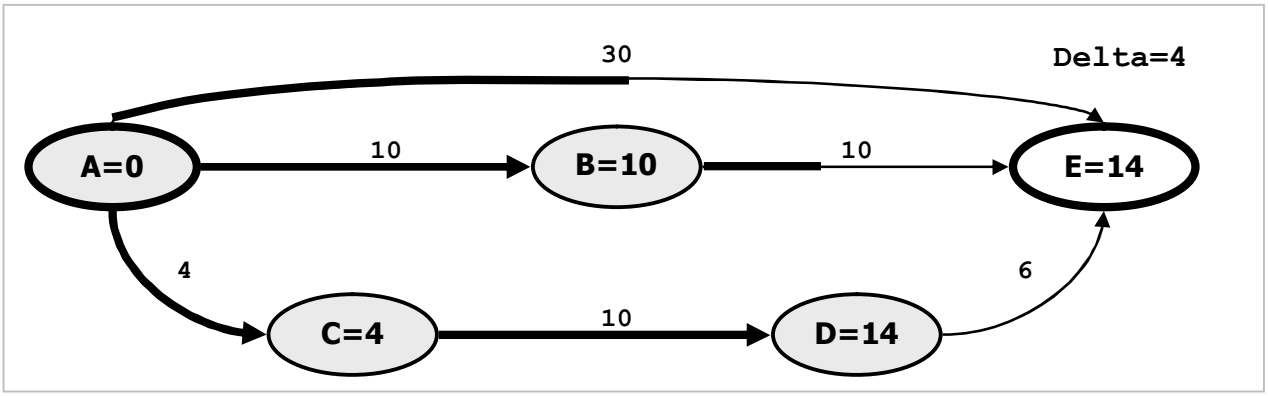


Рис. 26-5 — Добавлено 4 монеты (всего 14), приобретена дуга C-D

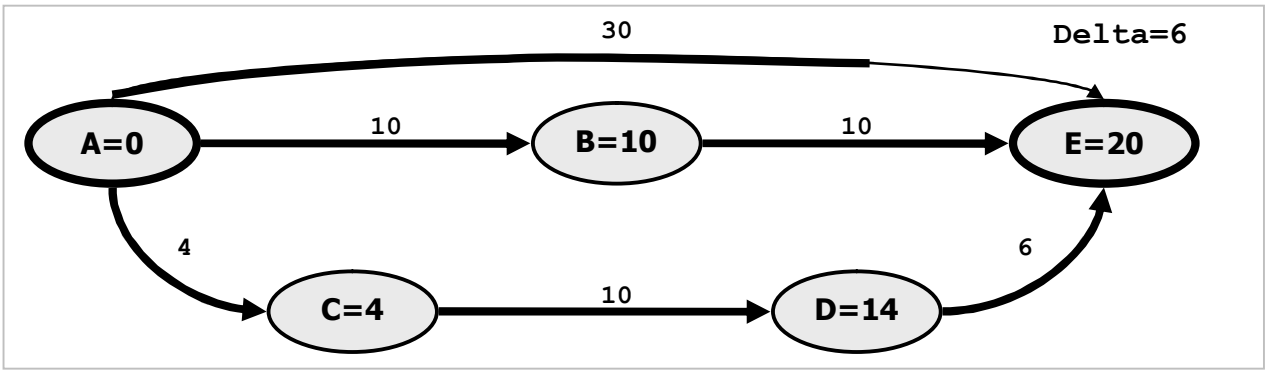


Рис. 26-6 — Добавлено 6 монет (всего 20), приобретены дуги B-E и D-E

Таким образом, после окраски стока у всех приобретённых в текущем торге дуг разности вершинных чисел сровняются с ценами этих дуг, что означает их приобретение. Этим признаком воспользуемся в процедуре поиска максимального потока вдоль купленных цепей. Напомню, что вершинное число в *стоке* покажет цену единицы потока вдоль очередной купленной цепи, — в процессе решения это число будет только возрастать.

В нашем примере приобретено две цепи с ценой пересылки 20 монет за единицу потока. Пустив максимальный поток вдоль этих цепей, вновь вернёмся на аукцион, но повторно приобрести эти цепи уже не сможем, поскольку они «закроются» насыщенными дугами. Торг пойдёт по иной траектории, в результате будет приобретена более дорогая дуга *A-E* по цене 30 монет за единицу потока. Таким образом, каждая следующая покупка будет обходиться всё дороже, пока не будут приобретены и насыщены цепи, пропускающие требуемый поток.

26.4. Метод вычисления стоимости потока

После обсуждения основных идей, обратимся к листингу, где представлены детали вычисления стоимости потока (напомню, что *цена дуги* содержится в поле **`TLink.mValue`**).

Листинг 26-1 — Вычисление минимальной стоимости заданного потока

```
function TGraph.CalcMinCostFlow(aSource, aDest: TNode; // источник и сток
                                aFlow: integer         // величина потока
                                ): integer;            // стоимость потока

var  Gray: TSet;     // множество серых (купленных) вершин
     Que: TBuffer;   // очередь вершин используется в CalcDeltaFlow
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка данных потока и вершинных чисел,
  // вызывается единожды в начале метода
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure ClearFlowData;
  var Node: TNode;
      Link: TLink;       // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
```

```
      Node.mDist:= 0;       // очищаем вершинные числа
      // Перебор исходящих линков вершины:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mFlow:= 0;           // текущий поток = 0
        Link.mColor:= CWhite;     // цвет дуги белый
        Link:= Node.OutLinkNext;
      end; // while
      Node:= NodeNext;
    end; // while
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка цвета (mColor) и окраска вершины-истока
  // Вызывается в начале поиска очередной увеличивающей цепочки
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure ClearColors;
  var Node: TNode;       // текущая вершина
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mColor:= CWhite;
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
      Node:= NodeNext;
    end;
    // Красим вершину-исток и помещаем в множество серых вершин:
    aSource.mColor:= CGray;
    Gray.Clear;
    Gray.Insert(aSource);
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Попытка приобретения очережной прямой дуги
  // Исходные данные:
  // - текущее множество серых вершин Gray
  // - текущий поток в дугах
  // - текущее состояние вершинных чисел
  // Возвращает true в случае покупки дуги, изменяя вершинные числа
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  function Buying: boolean;
  var Node: TNode;       // текущая вершина
      Link: TLink;       // текущий линк
      Delta: integer;    // очередное приращение суммы
      MinDelta: integer;// минимальное приращение суммы
  begin
    MinDelta:= MaxInt;
    // Перебор серых вершин:
    Node:= Gray.GetFirst as TNode;
    while Assigned(Node) do begin
      // Просмотр исходящих дуг:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do if mDest.mColor = CWhite then begin
          // Если поток не насыщен:
          if mFlow < mHigh then begin
            // Прицениваемся к увеличивающей дуге:
            // Link.mValue - цена увеличивающей дуги
            // mOwner.mDist - сумма оплаты цепочки вплоть до Node
            // mDest.mDist - частичная плата вплоть до mDest
            // Delta - то, что нужно доплатить за приобретение дуги Link
            Delta:= mValue - (mDest.mDist - mOwner.mDist);
            if Delta < MinDelta
              then MinDelta:= Delta;  // запоминаем минимальную доплату
          end; // if
```

```
        end;
        Link:= Node.OutLinkNext;
      end; // while
      Node:= Gray.GetNext as TNode; // Следующая вершина серого множества
    end; // while Assigned(Node)
    Result:= MinDelta < MaxInt;     // true, если дуга куплена
    if not Result or (MinDelta=0) then Exit;
    // Наращиваем mDist - вершинные числа белых вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mColor = CWhite then Inc(Node.mDist, MinDelta);
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск потока, проходящего по уже купленным дугам
  // Исходные данные:
  // - текущее множество серых вершин
  // - текущий поток
  // - текущие вершинные числа
  // Расширяет множество серых вершин.
  // Возвращает допустимое приращение потока (или ноль)
  // и стоимость единицы потока
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  function CalcDeltaFlow(var aCost: integer): integer;
  var
      Node: TNode;      // текущая вершина
      Link: TLink;      // исходящие и входящие дуги
  begin
    Result:=0; aCost:=0;
    Que.Clear;  // очищаем очередь вершин
    // Ставим в очередь серые вершины
    Node:= Gray.GetFirst as TNode;
    while Assigned(Node) do begin
      Que.Put(Node);
      Node:= Gray.GetNext as TNode;
    end;
    Que.Put(aSource);
    // Обработка вершин из очереди:
    while Que.GetCount>0 do begin
      Node:= Que.Get as TNode;
      // Обработка исходящих дуг текущего узла:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do if mDest.mColor=CWhite then begin
          // если вершина не помечена
          // Помечаем только белые вершины,
          // достигаемые через купленые ненасыщенные дуги
          // Link.mValue -- стоимость дуги
          // Link.mDest.mDist - mOwner.mDist -- разность вершинных чисел
          if (mValue = mDest.mDist - mOwner.mDist) // дуга куплена
             and (mFlow < mHigh) then begin  // и дуга не насыщена
            mDirect:= +1;  // отмечаем прямую дугу
            // помечаем концевую вершину и ставим в очередь:
            mDest.mColor:= CGray;     // цвет концевой вершины серый
            mDest.mPred:= Node;       // предшествующая вершина
            mDest.mLink:= Link;       // линк на предшествующую вершину
            mDest.mFlow:= Minimum(Node.mFlow, mHigh - mFlow); // поток
            Que.Put(mDest);       // в очередь приёмник дуги
            Gray.Insert(mDest);   // пополняем множество окрашенных
          end;
        end; // if
```

```
        Link:= Node.OutLinkNext;
      end;
      // Обработка входящих связей текущего узла:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        with Link do if mOwner.mColor=CWhite then begin
          // Помечаем только белые вершины,
          // из которых поступает ненулевой поток
          if mFlow > 0 then begin  // поток в дуге можно уменьшить
            mDirect:= -1;  // отмечаем встречную дугу
            // помечаем вершину и ставим в очередь:
            with Link.mOwner do begin
              mColor:= CGray;     // цвет исходной вершины серый
              mPred:= Node;       // предшествующая вершина
              mLink:= Link;       // линк на предшествующую вершину
              mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
            end;
            Que.Put(Link.mOwner);      // в очередь источник дуги
            Gray.Insert(Link.mOwner); // пополняем множество окрашенных
          end;
        end;
        Link:= Node.InLinkNext;
      end;  // while Assigned(Link)
      // Если помечен сток, то выход из цикла:
      if aDest.mColor <>  CWhite then Break;
    end;  // while Que.GetCount>0
    // Если сток помечен, то увеличивающая цепочка найдена
    // возвращаем допустимое увеличение потока и цену единицы потока
    if aDest.mColor <> CWhite then begin
      Result:= aDest.mFlow;
      aCost:= aDest.mDist;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Пересчёт потока вдоль предварительно найденной цепочки
  // aDelta - приращение потока
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure RecalcFlow(aDelta: integer);
  var Node: TNode;      // текущая вершина
  begin
    // Обратное движение от стока к истоку с корректировкой потока в дугах
    Node:= aDest;  // сток
    // Цикл, пока не достигнем истока:
    while Node <> aSource do begin
      // Node.mLink - линк из предшествующей вершины
      // Node.mLink.mDirect = +1 / -1
      with Node.mLink do Inc(mFlow, mDirect * aDelta); // поток
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Flow: integer;         // накопленный поток
  DeltaFlow: integer;    // приращение потока
  Cost1: integer;        // цена единицы потока
  OK: boolean;           // признак приобретения дуги

begin { TGraph.CalcMinCostFlow }

  Result:= 0;        // накопленная стоимость потока
  Flow:= 0;          // накопленный поток
  Gray:= CreateSet; // множество окрашенных (купленных) вершин
```

```
  Que:= TBuffer.Create;  // создаём очередь вершин
  ClearFlowData;     // очистка потока в дугах (mFlow=MaxInt)
  ClearColors;       // очистка цвета и начальная установка серого множества
  OK:= false;        // признак приобретения дуги

  // Пока поток не достиг заданного значения:
  while Flow < aFlow do begin
    // Пытаемся "купить" увеличивающую цепочку (дугу)
    // и формируем вершинные числа.
    // Если куплена хотя бы одна дуга, пытаемся провести поток:
    repeat
      // Пытаемся провести поток и определить цену единицы потока
      DeltaFlow:= CalcDeltaFlow(Cost1);
      if DeltaFlow > 0  then begin
        // Поток возможен, ограничиваем его приращение:
        if DeltaFlow > aFlow - Flow then DeltaFlow:= aFlow - Flow;
        // Распределяем поток по дугам и и определяем цену единицы потока:
        RecalcFlow(DeltaFlow);         // пересчёт потока в дугах
        Inc(Flow, DeltaFlow);          // суммарный поток
        Inc(Result, Cost1*DeltaFlow); // суммарная стоимость потока
        ClearColors;                   // очистка цвета и серого множества
      end else begin
        // Если поток не проведен, пытаемся купить хотя бы одну дугу
        // (функция Buying модифицирует вершинные числа)
        OK:= Buying;
        // Если цепочку купить нельзя, то заданный поток не существует
        if not OK then Break;
      end;
      // пока поток не достигнет нужного, либо не найдено приращение
    until Flow = aFlow;
    // Если цепочку купить нельзя, то заданный поток не существует
    if not OK then begin
      Result:=-1;  // стоимость = -1
      Break;
    end;
  end; // while Flow < aFlow
  // Очистка памяти:
  Gray.Free; // серое множество
  Que.Free;  // рабочая очередь
end;
```

Здесь функция **CalcDeltaFlow**, вычисляющая допустимое приращение потока вдоль очередной цепи, почти совпадает с аналогичной функцией в методе поиска *максимального* потока. Единственное отличие состоит в дополнительной проверке, куплена ли дуга:

```
          if (mValue = mDest.mDist - mOwner.mDist) // дуга куплена
             and (mFlow < mHigh) then begin        // и дуга не насыщена
```

Функция покупки очередной дуги **CalcCost1** в целом соответствует обсуждённому выше алгоритму.

26.5. Тестирование

Программа для проверки метода вычисления минимальной стоимости потока представлена ниже.

Листинг 26-2 — Минимальная цена потока

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CFile = 'Test.txt';

var Gr : TGraph;
    S, T : TNode;
    Flow: integer;
    Cost: integer;
begin
    Gr:= TGraphChars.Load(CFile);
    (Gr as TGraphChars).LoadFlowData(CFile);
    S:= (Gr as TGraphChars).GetNode('A');
    T:= (Gr as TGraphChars).GetNode('G');
    Flow:= Gr.CalcMaxFlow0(S, T);
    Writeln('MaxFlow= ', Flow);
    repeat
      Write('Flow = '); Readln(Flow);
      Cost:= Gr.CalcMinCostFlow(S, T, Flow);
      Writeln('Cost= ', Cost);
      Gr.ExpoLinksData;
    until Flow=0;
    Gr.Free;
end.
```

Для тестирования взят граф, изображённый на рис. 26-7, данные для его ввода представлены ниже (стоимость дуги задана полем **mValue**).

```
Test.txt - cost of flow (Kristofides, 342)
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
7 - количество вершин
A B C D E F G
A -> B=7 D=13 F=28
B -> C=25 D=4 E=10
C -> G=5
D -> C=6 F=5
E -> G=12
F -> E=3 G=7
G ->

FLOW:
AB= 0 16
AD= 0 11
AF= 0 13
BC= 0 17
BD= 0 18
BE= 0 16
CG= 0 22
DC= 0 12
DF= 0 19
EG= 0 16
FE= 0 10
FG= 0  5
```

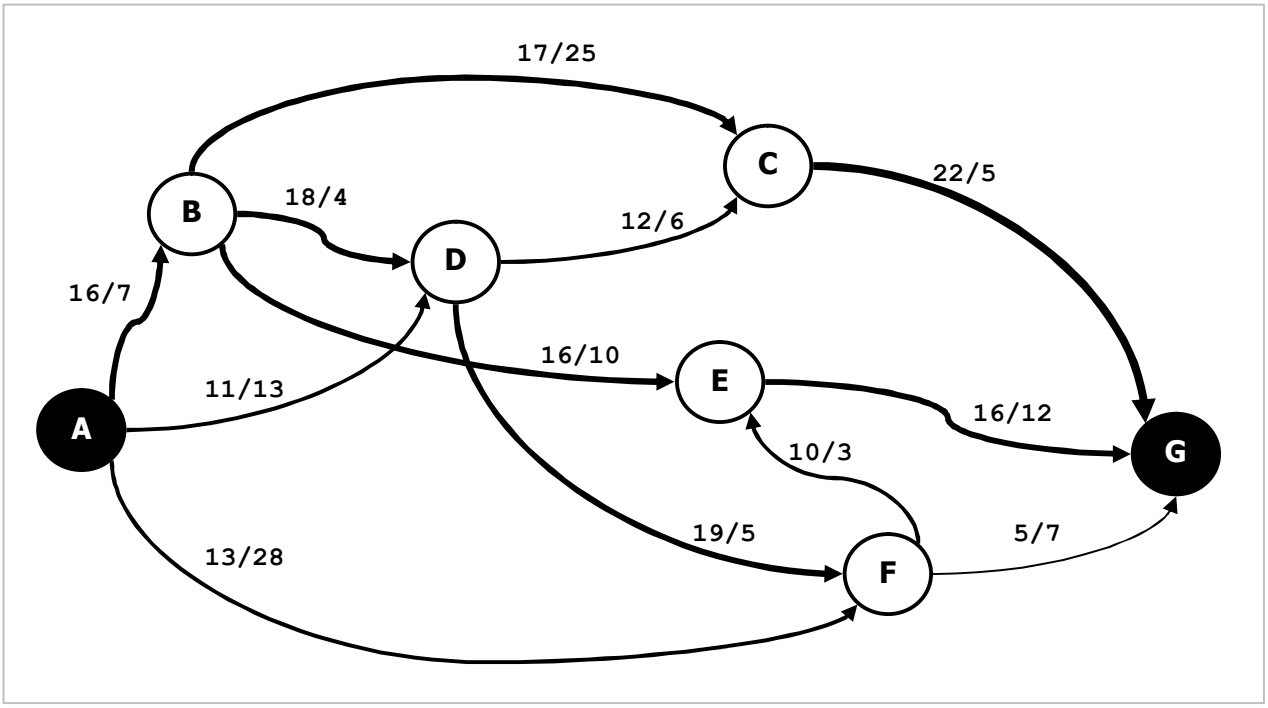


Рис. 26-7 — Граф для испытания алгоритма
вычисления минимальной стоимости потока
(числитель — пропускная способность, знаменатель — цена дуги)

На следующих рисунках показаны найденные цепочки, их цены и пропускные способности, а также вершинные числа и распределение потока в дугах. Каждый рисунок соответствует покупке очередной цепи, увеличивающей поток.

Обратите внимание на изменение потока в обратных дугах. Так, например, пущенный на первых итерациях по дуге *D-F* поток в **5** единиц (рис. 26-11), позже перераспределяется так, что полностью уходит по другим маршрутам, и дуга *D-F* остаётся в итоге «сухой» (рис. 26-12).

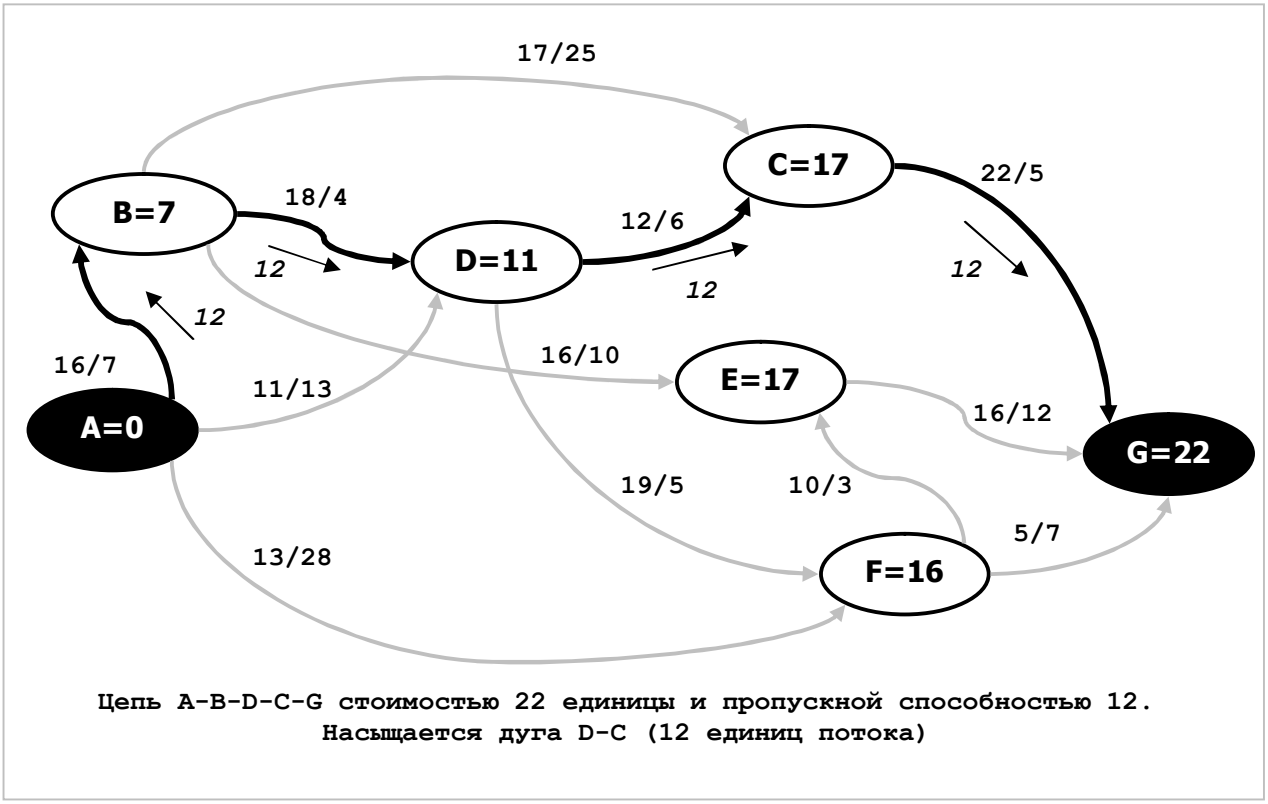
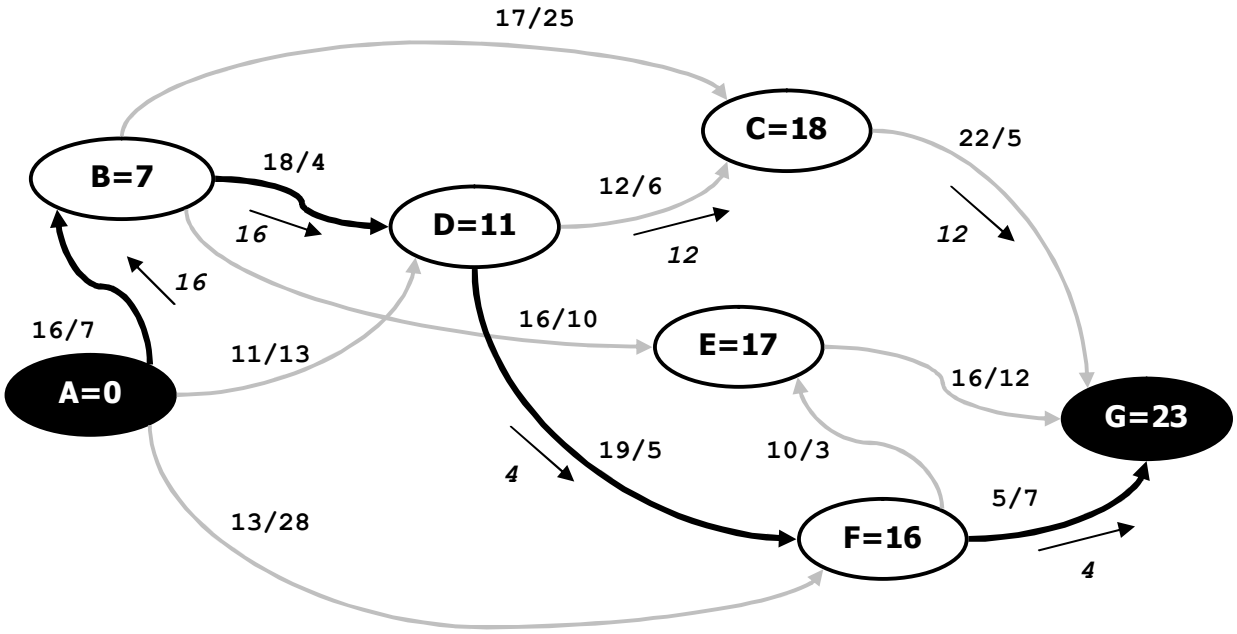
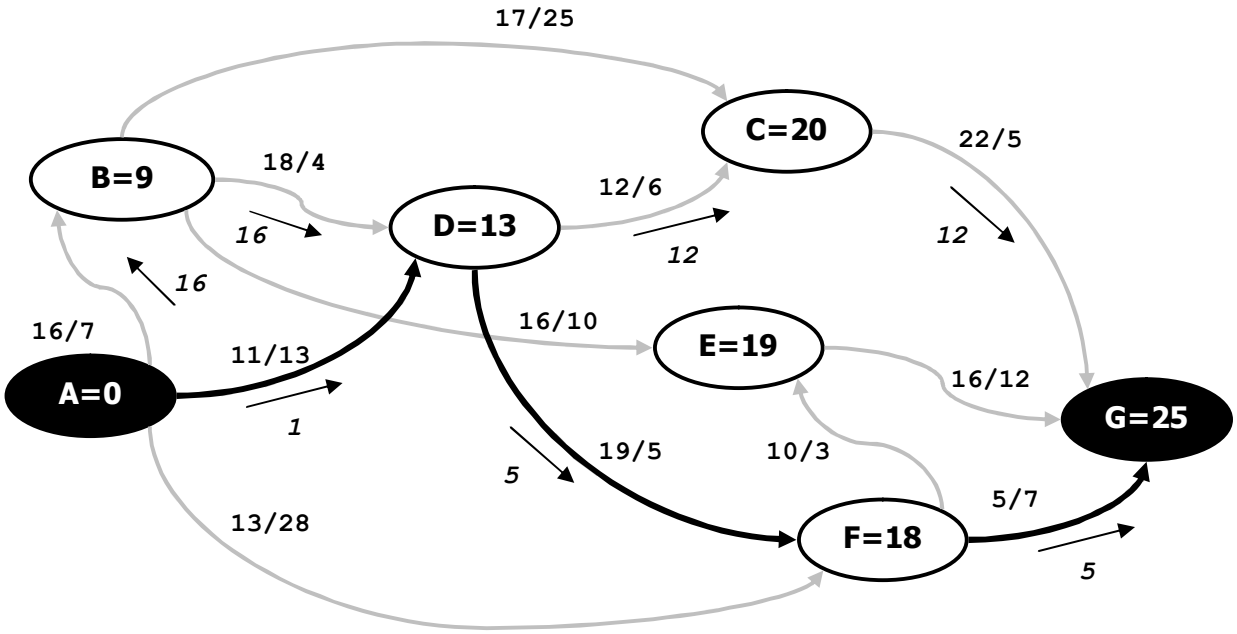


Рис. 26-8 — Первая цепочка стоимостью 22 единицы



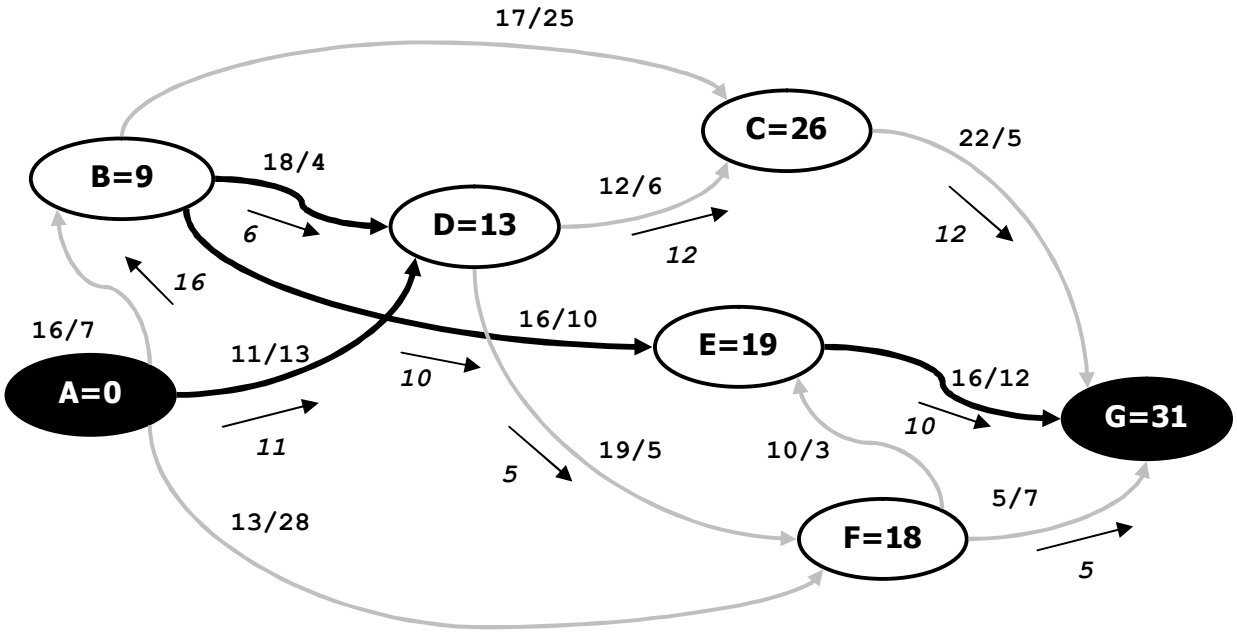
Цепь A-B-D-F-G стоимостью 23 единицы и пропускной способностью 4.
Насыщается дуга A-B (12+4 = 16 единиц потока)

Рис. 26-9 — Вторая цепочка стоимостью 23 единицы



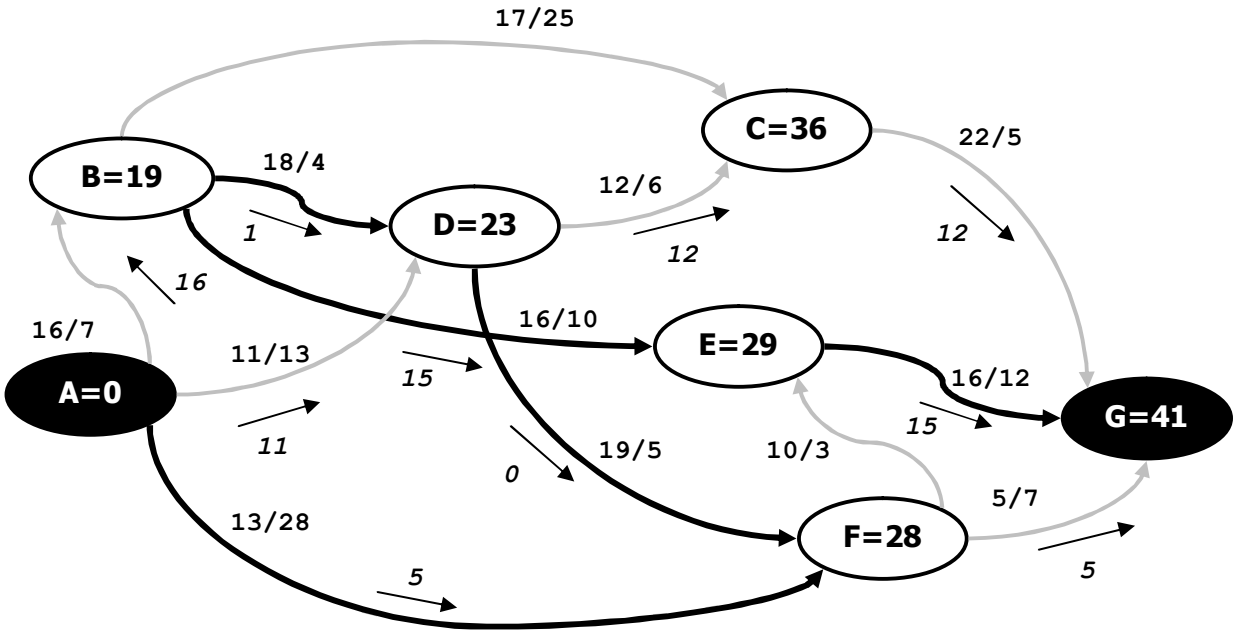
Цепь A-D-F-B стоимостью 25 единиц и пропускной способностью 1.
Насыщается дуга F-G (4+1 = 5 единиц потока)

Рис. 26-10 — Третья цепочка стоимостью 25 единиц



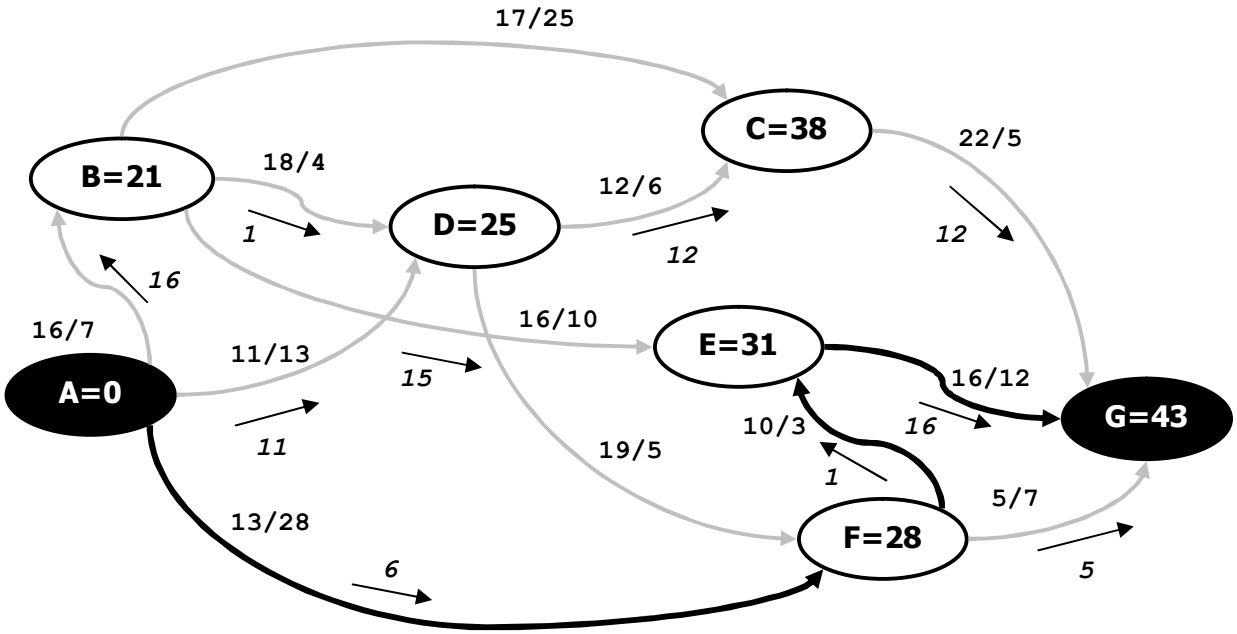
Цепь A-D-B-E-G стоимостью 31 единица и пропускной способностью 10.
Насыщается дуга A-D (1+10 = 11 единиц потока).
Поток в обратной дуге B-D уменьшается с 16 до 6 единиц.

Рис. 26-11 — Четвёртая цепочка стоимостью 31 единица



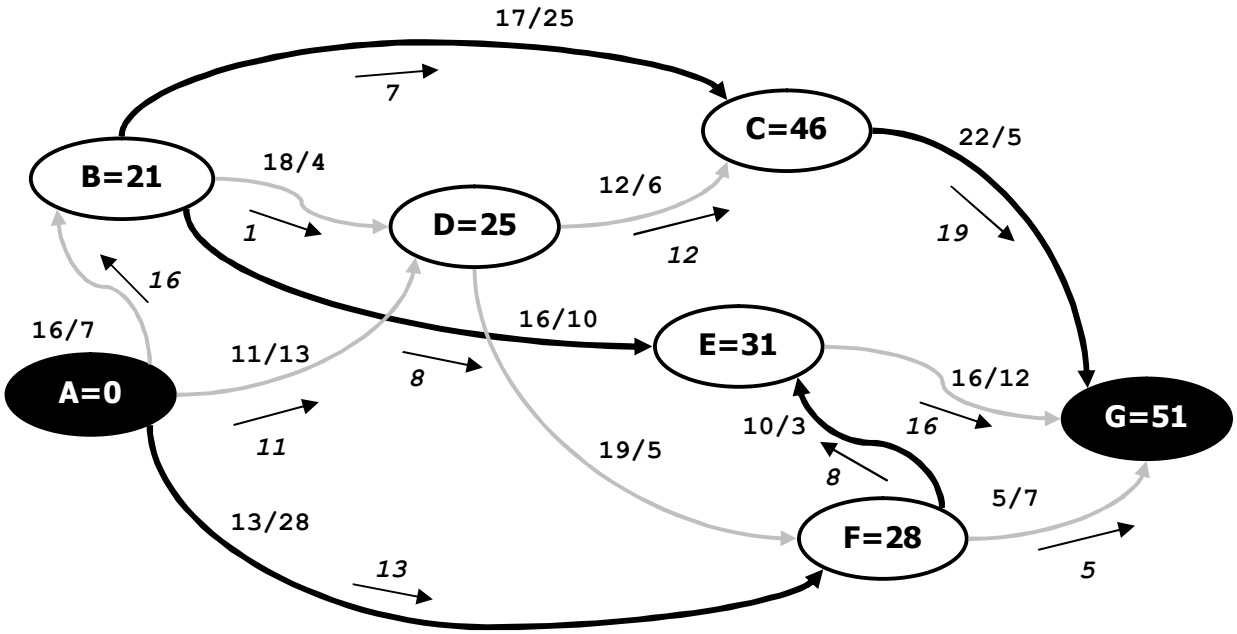
Цепь A-F-D-B-E-G стоимостью 41 единица и пропускной способностью 5.
Исчезает поток в обратной дуге D-F (5-5 = 0 единиц потока).
Поток в обратной дуге B-D уменьшился с 6 до 1 единицы.

Рис. 26-12 — Пятая цепочка стоимостью 41 единица



Цепь A-F-E-G стоимостью 43 единицы и пропускной способностью 1.
Насыщается дуга E-G (15+1 = 16 единиц потока).

Рис. 26-13 — Шестая цепочка стоимостью 43 единицы



Цепь A-F-E-B-C-G стоимостью 51 единица и пропускной способностью 7.
Насыщается дуга A-F (6+7 = 13 единиц потока).
Поток в обратной дуге B-E уменьшился с 15 до 8 единиц.

Рис. 26-14 — Седьмая цепочка стоимостью 51 единица

Приведенные выше рисунки с этапами и результатами расчёта ещё раз демонстрируют следующие утверждения:

- Сумма потоков, исходящих из истока S, равна сумме потоков, поглощаемых стоком T.
- Сумма потоков, входящих в промежуточную вершину, и исходящих из неё, равна нулю.
- Разность вершинных чисел смежных вершин вдоль вновь приобретённой цепи равна цене инцидентных им дуг.
- Вершинное число в стоке T после покупки цепи равно стоимости вновь приобретённой цепи.
- Максимальный поток вдоль очередной цепи либо насыщает прямую дугу, либо истощает обратную. И те, и другие дуги в ходе следующей итерации будут трактоваться как нейтральные, и потому не влияют на дальнейшее распределение потока.

Итоги вычислений сведены в табл. 26-1. В левом столбце показана очередная найденная цепь и её цена, в среднем — текущее значение потока (увеличивается с единичным шагом), в правом столбце — накопленная стоимость потока, как сумма произведений цены цепи на величину пропускаемого по ней потока.

Табл. 26-1 — Сводная таблица с результатами расчёта минимальной стоимости потоков

Цена единицы потока и цепь	Величина потока	Стоимость потока суммарная	Цена единицы потока и цепь	Величина потока	Стоимость потока суммарная
22 A-B-D-C-G	1	22	31 A-D-B-E-G	21	505
	2	44		22	536
	3	66		23	567
	4	88		24	598
	5	110		25	629
	6	132		26	660
	7	154		27	691
	8	176	41 A-F-D-B-E-G	28	732
	9	198		29	773
	10	220		30	814
	11	242		31	855
	12	264		32	896
23 A-B-D-F-G	13	287	43 A-F-E-G	33	939
	14	310	51 A-F-E-B-C-G	34	990
	15	333		35	1041
	16	356		36	1092
25 A-D-F-G	17	381		37	1143
31 A-D-B-E-G	18	412		38	1194
	19	443		39	1245
	20	474		40	1296

26.6. Итоги

- Задача вычисления минимальной стоимости потока состоит в том, чтобы при заданных пропускных способностях дуг и их стоимостях наиболее рационально распределить поток по дугам сети, и попутно определить стоимость этого потока.
- Задача решается «покупкой» увеличивающих цепей, начиная с самых «дешёвых». Через приобретённые цепи пропускают допускаемый ими поток, после чего некоторые прямые дуги сети насыщаются либо истощаются обратные. Те и другие не рассматриваются в ходе последующих «торгов».

26.7. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 339, Стр. 351
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 104
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 27
Паросочетание в двудольном графе

Поиск пары — задача, решаемая природой каждой весной. Существует много других примеров: оптимальная расстановка рабочих по станкам, формирование спортивных пар и т.д.

27.1. Двудольные графы

Задачи этого рода моделируют *двудольными* графами. *Двудольным* называют граф, все вершины которого разбиты на два подмножества — назовём их условно левым и правым. При этом все вершины левого подмножества связаны рёбрами или дугами *только* с вершинами правого подмножества, и наоборот. Примеры: покупатели и товары, рабочие и станки. Рассмотрим ещё пример.

Торговец недвижимостью хочет продать с наибольшей выгодой ряд домов и подыскал нескольких покупателей. Каждый покупатель оговорил приемлемую для него цену нескольким приглянувшимся ему домам, намереваясь в итоге купить один из них. Таким образом, каждый покупатель заинтересован несколькими домами, и каждый дом приглянулся нескольким покупателям, назначившим за него каждый свою цену. Теперь продавцу надо распределить дома между покупателями так, чтобы сумма всех сделок оказалась максимальной. Такова задача поиска паросочетания максимальной стоимости на двудольном графе, где левую долю составляют покупатели, а правую — товары. Здесь надо *максимизировать* сумму, в других задачах требуется *минимизировать* что-то (расходы, убытки), — оба варианта задачи решаются одним алгоритмом, поскольку взаимно преобразуются друг в друга.

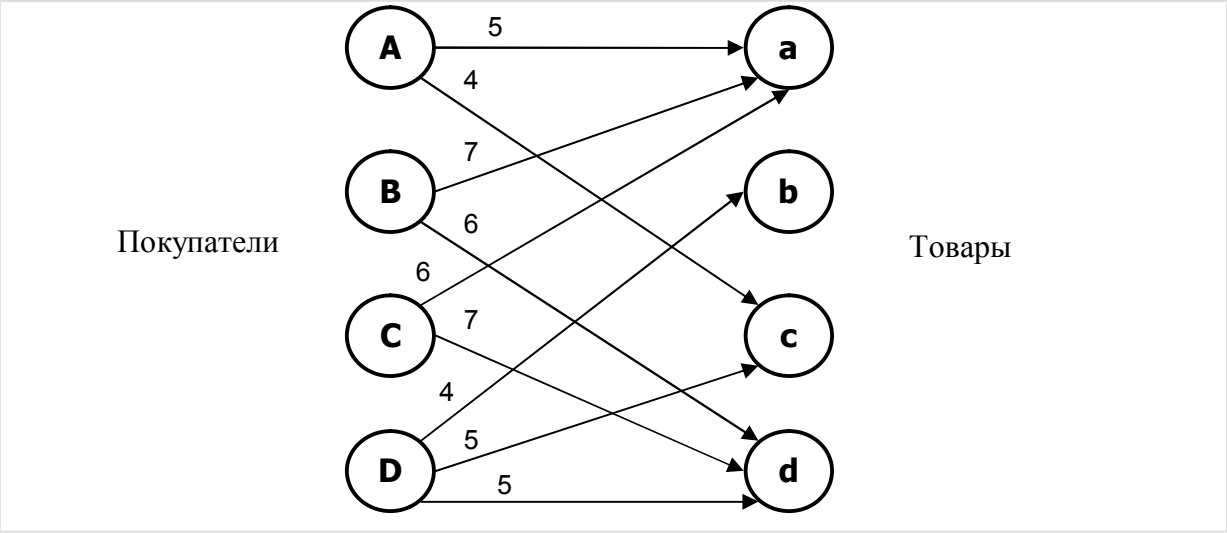


Рис. 27-1 — Покупатели (слева) и товары (справа)

Двудольный граф на рис. 27-1 отражает ситуацию с покупателями (A, B, C, D) и домами (a, b, c, d). Числа у стрелок означают цену, которую готов уплатить покупатель за тот или иной дом. Здесь домом a заинтересовались три покупателя,

а домом *b* лишь один. Количество покупателей и товаров может не совпадать, и тогда либо кто-то останется без дома, либо часть домов окажется не проданной.

Той же схемой можно описать ситуацию, когда вершинам левой доли соответствуют источники товара (фабрики или оптовые базы), а вершинам правой доли — его потребители (розничные магазины). Стрелки показывают возможные пути доставки и стоимость доставки товара. Пусть каждый источник товара готов обеспечить лишь один магазин. Тогда хозяину розничной сети для минимизации расходов на доставку надо найти *наибольшее* паросочетание *минимальной* стоимости.

Итак, решаемая задача формулируется следующим образом. Дан двудольный граф, вершины которого разбиты на два подмножества: левое и правое. Левое подмножество соединяется с правым взвешенными дугами или рёбрами. Необходимо найти наибольшее множество дуг (рёбер), суммарная стоимость которых окажется минимальной (или максимальной).

27.2. Термины, определения и очевидные закономерности

Прежде, чем обсуждать пути решения задачи о паросочетаниях, ознакомимся с терминами, принятыми в этой сфере.

Что такое *паросочетание* ? Это подмножество попарно несмежных рёбер графа, то есть рёбер, не имеющих общих вершин. Отметим, что *пустое* множество рёбер тоже является паросочетанием.

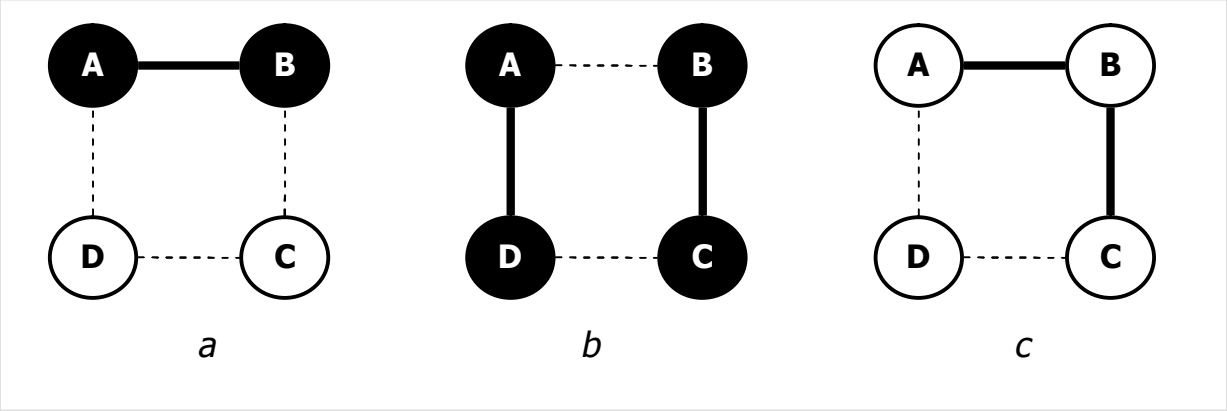


Рис. 27-2 — Паросочетания (a) и (b)

Обратимся к рис. 27-2. Здесь рёбра, не входящие в паросочетание, изображены тонким пунктиром и условно считаются *белыми*, а рёбра, входящие в него, выделены сплошной жирной линией и условно считаются *чёрными*. Таким образом, на рисунке (*a*) показано паросочетание из одного ребра *A-B*, а на рисунке (*b*) — паросочетание из двух рёбер: *A-D* и *B-C*. На рисунке (*c*) выделено жирным два ребра, но они *не составляют* паросочетания, поскольку имеют одну общую вершину *B*.

Вершины, не инцидентные рёбрам паросочетания, называют *свободными* или *экспонированными*. На рисунке (*a*) это вершины *D* и *C*, будем красить их *белым*. Вершины, соединённые ребром паросочетания, окрасим чёрным.

Из всего сказанного следует:

- у чёрного ребра (входящего в паросочетание) обе смежные вершины чёрные;
- у белого ребра хотя бы одна из двух инцидентных вершин белая;
- у чёрной вершины хотя бы одно смежное ребро чёрное;
- у белой вершины все смежные рёбра белые.

Паросочетание, которое невозможно расширить добавлением рёбер, не удалив из него уже имеющееся, называется *максимальным*, а паросочетание, содержащее наибольшее количество рёбер, — *наибольшим*. На рис. 27-3 показаны максимальное паросочетание (a), и два наибольших (b, c). И тех, и других, в графе может быть несколько. Отметим, что к паре *A-C* невозможно присоединить другие рёбра, и потому такое паросочетание является максимальным, но не наибольшим. *Наибольшее* паросочетание является одновременно и *максимальным*, но не наоборот.

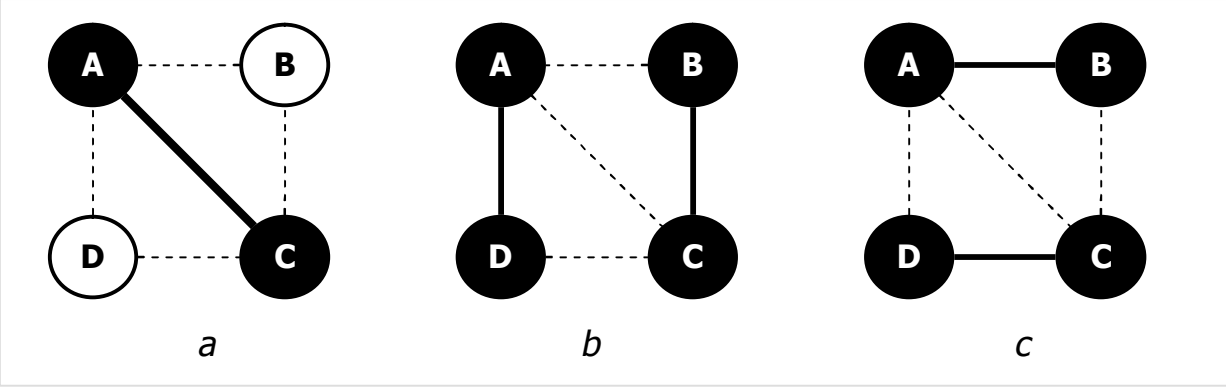


Рис. 27-3 — Максимальное (a) и два наибольших (b, c) паросочетания

Совершенным называют паросочетание, в которое вовлечены все вершины, — такой граф должен содержать чётное количество вершин. *Совершенное* паросочетание одновременно является и *наибольшим*.

Оптимальным назовём такое *наибольшее* паросочетание на взвешенных рёбрах, сумма весов рёбер которого либо *минимальна*, либо *максимальна* (зависит от постановки задачи). Таким образом, в упомянутых выше задачах ищется одно из *оптимальных* паросочетаний: либо минимального, либо максимального веса.

27.3. Увеличивающие цепи и деревья

Подход, принятый в алгоритмах поиска *наибольших* паросочетаний состоит в постепенном добавлении рёбер к текущему паросочетанию, начиная с пустого. Для

описания процесса введём ещё два термина: чередующаяся (*альтернирующая*) цепь, и увеличивающая (*аугментальная*) цепь.

Альтернирующая цепь состоит из попарно смежных рёбер, *чередующихся* так, что за ребром не входящим в паросочетание следует ребро, входящее в него, и т.д.; примеры представлены на рис. 27-4. *Альтернирующие* цепи в свою очередь могут оказаться:

- a) увеличивающими (*аугментальными*);
- b) уменьшающими;
- c) нейтральными (*венгерскими*).

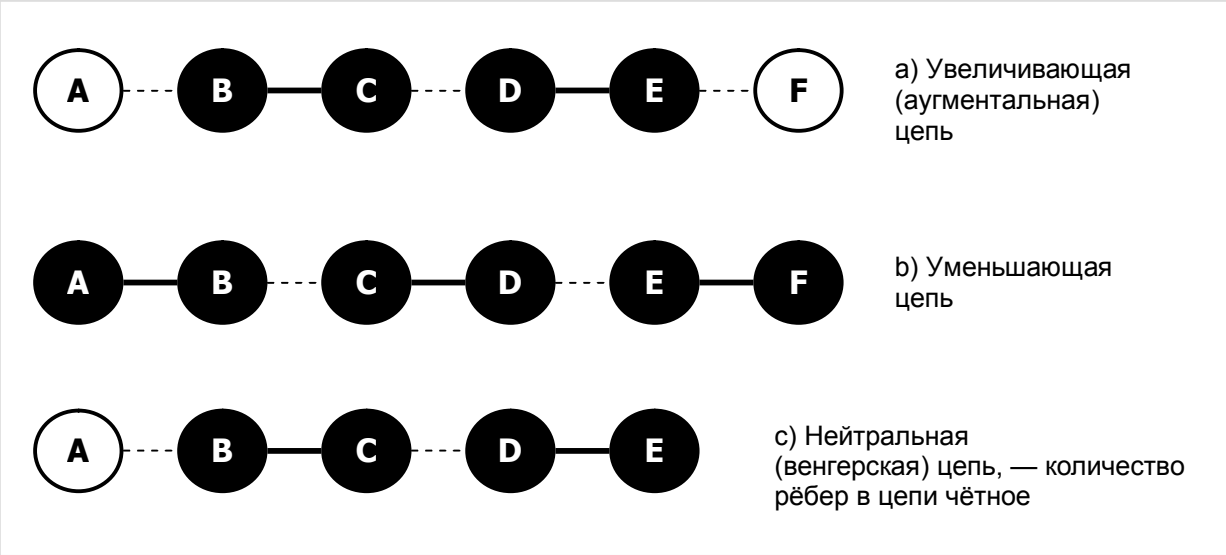


Рис. 27-4 — Альтернирующие (чередующиеся) цепи

В *аугментальной* цепи два её крайних ребра и две крайние вершины — белые (свободные). Если рёбра такой цепи инвертировать — перекрасить в противоположные цвета (белые в чёрные, а чёрные в белые), — то в паросочетании станет на одно чёрное ребро больше. Этим свойством не обладает нейтральная (венгерская) цепь. Отсюда следует, что для последовательного расширения паросочетания надо отыскивать *аугментальные* цепи.

27.4. Поиск увеличивающих (аугментальных) цепей

Поиск исходит из того, что *увеличивающая* цепь всегда начинается с белой (экспонированной) вершины и замыкается такой же. Стало быть, увеличивающую цепь ищут, начиная с *белой* вершины; этот поиск может завершиться:

- a) успешно — обнаружением другой белой вершины, либо
- b) неуспешно — после посещения всех доступных вершин графа.

Поскольку из белой вершины в общем случае можно следовать по нескольким направлениям, в ходе поиска будет порождаться дерево с корнем в этой вершине.

Дерево можно формировать обходом либо в глубину, либо в ширину. Рассмотрим обход в ширину на примере графа, представленного на рис. 27-5.

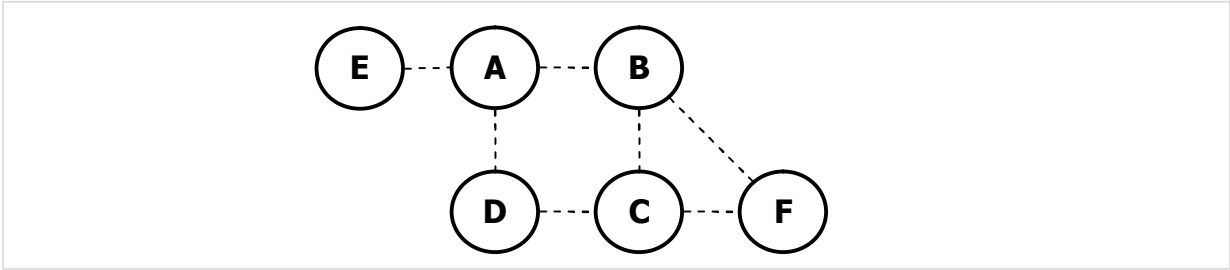


Рис. 27-5 — Исходный граф для поиска паросочетания

Будем обрабатывать вершины в алфавитном порядке, и начнём строить дерево из вершины *A*. Отсюда сразу попадаем в белую вершину *B*, и на этом построение дерева завершится, поскольку найдена увеличивающая цепь *A-B*. Инвертировав её, добавим в паросочетание первое ребро (рис. 27-6).

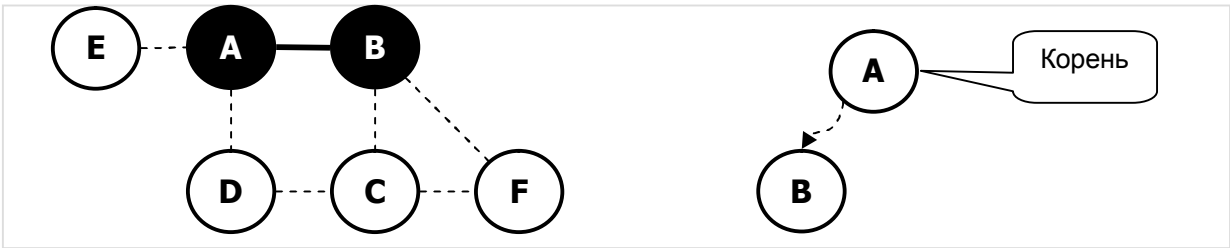


Рис. 27-6 — Построение цепи из вершины A, найдено ребро A-B

На следующей итерации дерево строится из белого корня *C*, и к множеству добавляется ребро *C-D* (рис. 27-7), а в графе остаются лишь две белые вершины.

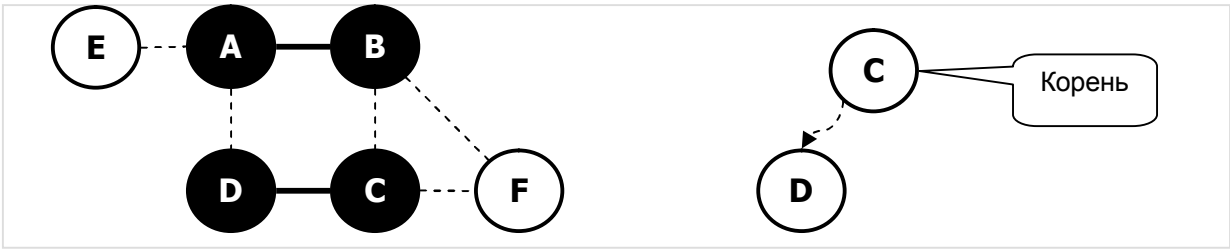


Рис. 27-7 — Построение цепи из вершины C, найдено ребро C-D

Начав строить дерево из белого корня *E*, сразу попадаем в чёрную вершину *A*. Казалось бы, отсюда можно следовать далее в вершины *B* и *D*. Однако белые рёбра здесь не дадут чередующейся цепи. Поэтому из первой чёрной вершины *A* (её называют *внутренней*) можно следовать далее только по чёрному ребру в чёрную вершину *B* (её называют *внешней*), соблюдая тем самым чередование рёбер. Из *внешней* чёрной вершины *B* можно следовать далее по нескольким белам рёбрам, так попадаем в две вершины: чёрную *C* и белую *F*. Вершиной *F* постройка дерева завершается (рис. 27-8), поскольку очередная увеличивающая цепь найдена.

Здесь введены понятия *внутренней* и *внешней* вершин цепи. Из двух вершин парного ребра *внутренней* будем называть ту его вершину, которая находится ближе к корню дерева. Соответственно другая вершина будет *внешней*.

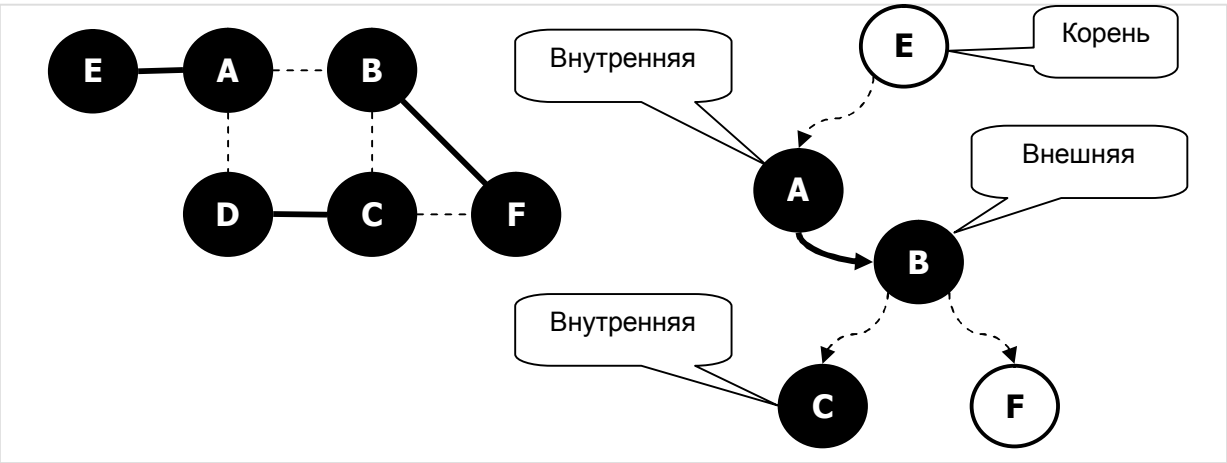


Рис. 27-8 — Построение цепи из вершины E,
найдена увеличивающая цепь E-A=B-F

Инвертировав аугментальную цепь *E-A=B-F*, добавим в паросочетание два ребра и удалим одно. Так образуется цепь *E=A-B=F* (здесь знаком «=» отмечено чёрное ребро паросочетания). Теперь в графе не осталось белых вершин, — процедура завершена.

Рассмотрим обработку графа, показанного на рис. 27-9. Здесь количество вершин нечётное, стало быть, одна из них после построения паросочетания неминуемо останется белой.

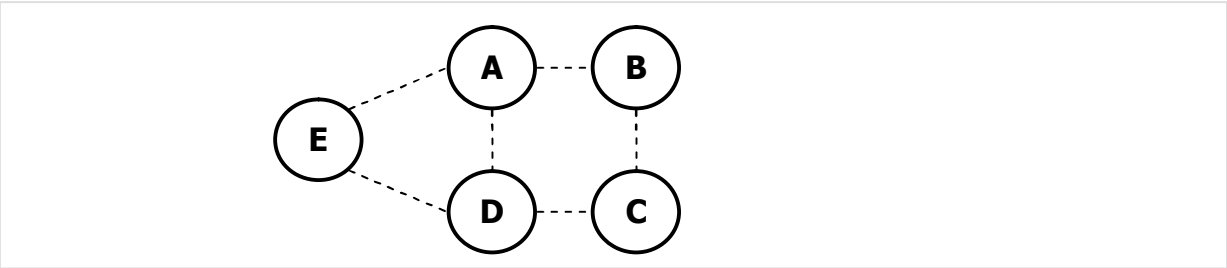


Рис. 27-9 — Граф с нечётным количеством вершин

Рёбра *A=B* и *C=D* будут добавлены в паросочетание точно так же, как описано выше. При построении дерева из вершины *E* получим картину, показанную на рис. 27-10, — здесь из вершины *E* протянулись две нейтральные *венгерские* цепи, соответствующее дерево тоже называют *венгерским*. В этой ситуации расширить паросочетание невозможно.

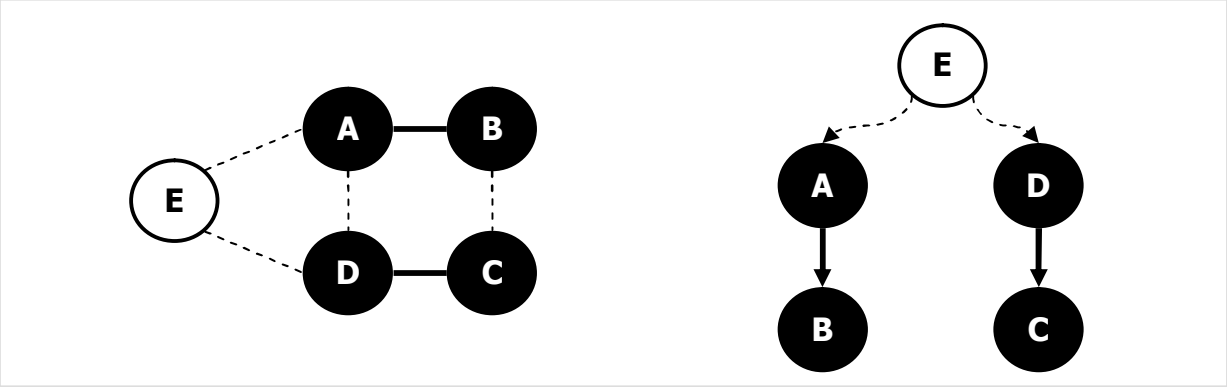


Рис. 27-10 — Дерево с корнем в C оказалось венгерским

Дадим теперь алгоритм построения *наибольшего* паросочетания в графе:

- Красим все вершины и рёбра белым.
- Для всех белых вершин:
  - Берём очередную белую вершину и обходом в ширину строим дерево с корнем в этой вершине. Постройка дерева завершается либо нахождением увеличивающей цепи, либо построением венгерского дерева.
  - Если на предыдущем шаге найдена увеличивающая цепь, инвертируем цвета рёбер вдоль этой цепи, и две крайние вершины цепи красим чёрным.

При постройке дерева учитываем, что ближняя к корню чёрная вершина пары является *внутренней*, из неё по *чёрному* ребру можно следовать только в следующую чёрную вершину — *внешнюю*. Из *внешней* чёрной вершины по белым рёбрам открыты пути в другие вершины. Так достигается чередование белых и чёрных рёбер в создаваемых цепях.

Этот алгоритм применим и к двудольным графам. Отметим, что увеличивающая цепь всегда содержит нечётное количество рёбер или дуг. Поэтому, выбрав корнем белую вершину *левой* доли, успешную постройку можно завершить только в *белой* вершине *правой* доли. Если же цепь окажется *венгерской,* её постройка закончится в *чёрной* вершине *левой* доли.

27.5. Венгерский алгоритм Куна

Описанный выше алгоритм даёт *наибольшее* паросочетание, однако не учитывает вес рёбер. Для построения *оптимального* (минимального или максимального по весу) паросочетания на *двудольном* графе применяют *венгерский* алгоритм *Куна*, названый так в память о венгерских математиках, внёсших вклад в его разработку. Алгоритм Куна включает в себя описанный выше поиск *наибольшего* паросочетания, но вдобавок выполняет ещё ряд действий, смысл которых поясним следующим упрощённым примером.

Представим страну, где на берегах Дуная приютилось *чётное* количество городов, — по **N** городов на каждом берегу. Причём каждый город одного берега соединён мостом с каждым городом другого, как показано на рис. 27-11.

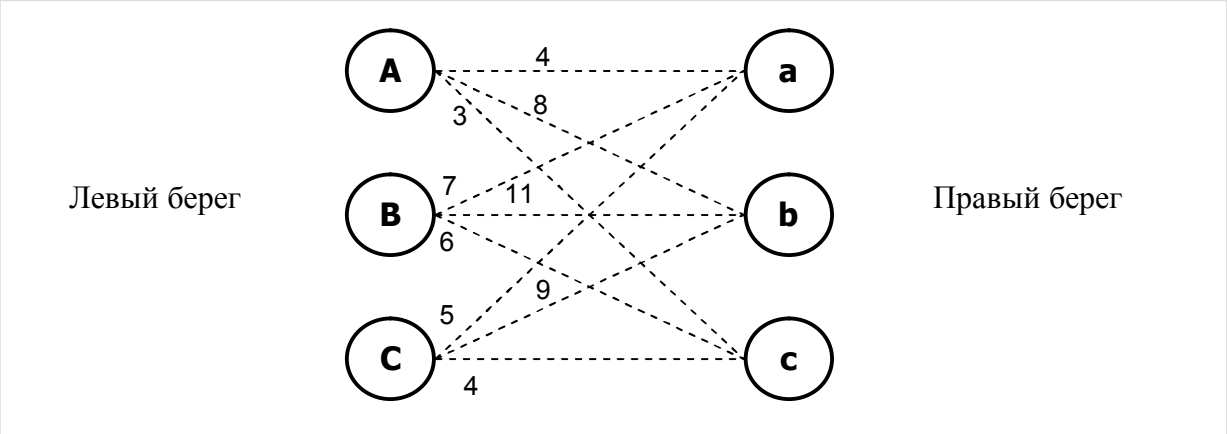


Рис. 27-11 — Схема городов с мостами (полный двудольный граф)

Здесь числа рядом с рёбрами относятся не к длине мостов, а характеризуют иное свойство — степень *наклона* моста, его *крутизну*. Дело в том, что правый берег реки выше левого, и все опоры правого берега выше левобережных, как это показано на рис. 27-12. Числа рядом с опорами указывают их высоту, а разность высот определяет *крутизну* моста, условно выражаемую *весом* ребра графа.

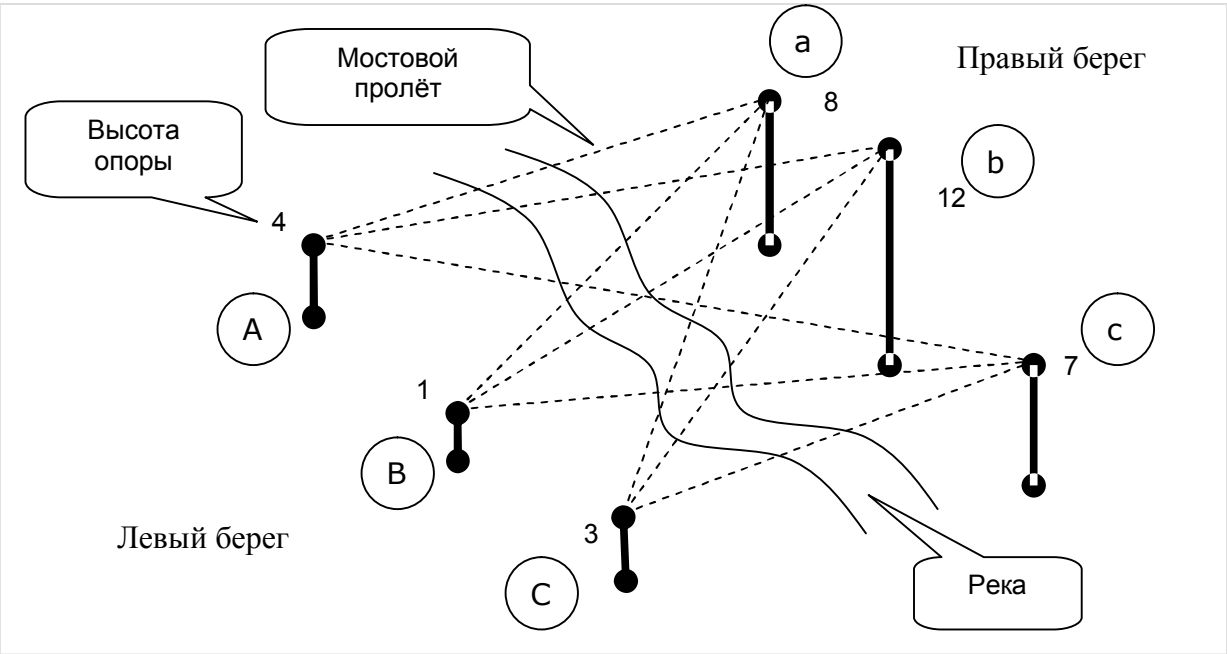


Рис. 27-12 — Левые и правые опоры мостов (правые опоры выше)

Технический прогресс привёл правительство к мысли соединить города железными дорогами. В целях экономии решили ограничиться минимум дорог с тем условием, чтобы каждый левобережный город соединялся путями только с одним правобережным, и наоборот. Стало быть, мосты с путями здесь составят *наибольшее* паросочетание.

Но к чему тут уклон мостов? Дело в том, что путейцы могут прокладывать рельсы только по выровненным в горизонтальное положение мостам. Выравнивание мостов поднятием опор требует средств. Решили, что будут выравнивать мосты в минимальной степени, лишь бы на выровненных мостах построить *наибольшее* паросочетание, а остальные не трогать. В итоге приходим к заключению, что мосты с путями здесь составят *наибольшее* паросочетание *минимального* веса.

Теперь следим за действиями строителей. Обследовав все опоры левого берега (белые вершины), выяснили, что найти здесь хотя бы одну увеличивающую цепь невозможно ввиду отсутствия горизонтальных мостов. Зато в ходе обследования обнаружили, что, приподняв все левые опоры на 3 метра, можно гарантированно выровнять, по крайней мере, один мост. Из рисунка понятно, что для этого достаточно приподнять лишь опору *A*, но по своему опыту строители знали, что прочие опоры в будущем поднимать тоже придётся. Итак, приподняв на три метра опоры *A*, *B* и *C*, строители выровняли мост *A-c*, и после второй попытки поиска паросочетания добавили к нему чёрное ребро *A=c*. Это состояние отражено на рис. 27-13. Здесь отметим, что крутизна всех прочих мостов после поднятия опор тоже уменьшилась (вот что значит опыт!).

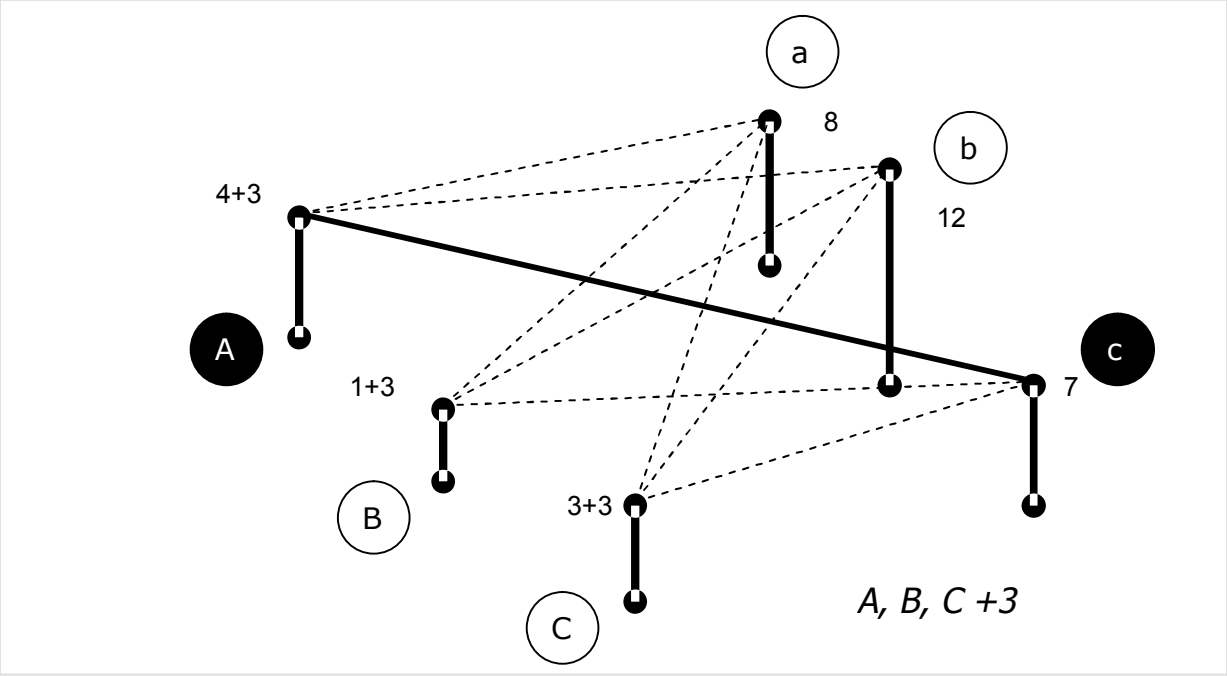


Рис. 27-13 — Поднятие опор на 3 метра
и добавление ребра A=c в паросочетание

Следующая попытка найти аугментальную цепь, следуя из белых вершин *B* и *C*, вновь окончилась неудачно. Зато выяснилось, что для выравнивания следующего моста достаточно приподнять эти вершины на один метр, что и было сделано. Результат показан на рис. 27-14, — здесь оказался выровненным мост *C-c* (горизонтальные мосты обозначены сплошной линией).

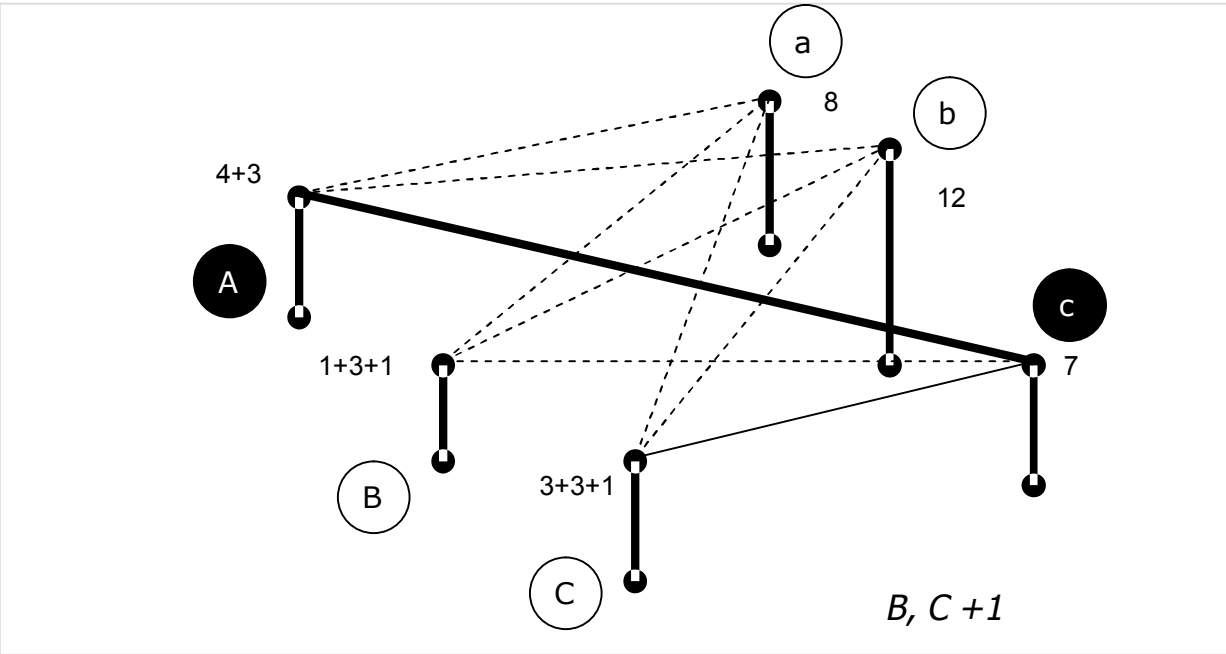


Рис. 27-14 — Поднятие опор B и C и на 1 метр и выравнивание моста C-c

Казалось бы, теперь можно проложить рельсы по выровненному мосту *C-c*, однако не будем забывать, что город *c* уже связан мостом с правым берегом, и второй ему не положен. Другими словами, из вершины *C* можно проложить только венгерскую цепь *C-c=A* (знаком «=» метим чёрные ребра паросочетания).

Введём здесь понятие *платформы* — так кратко назовём множество вершин, посещённых при очередной попытке расширить паросочетание. Поначалу платформу составляли только *белые* вершины *левого* берега, — от них пытались строить увеличивающие цепи. Но сейчас их география расширится: в попытках строить цепи будут посещены: сначала вершина *B*, затем из вершины *C* вершины *c* и *A*. И тогда на текущий момент платформа составит множество вершин *A-B-C-c*.

Очередная неудачная попытка открыла строителям новое знание: для выравнивания следующего моста надо приподнять некоторые опоры ещё на метр. Но какие именно? Если только две белые на левом берегу (*B* и *C*), то нарушится горизонтальность моста *C-c*, следовательно, опора *c* тоже нуждается в подъёме. Однако поднятием опоры *c* нарушится горизонтальность моста *A-c*. В итоге выясняется, что поднять на метр надо всю *платформу A-B-C-c*, что и было сделано (рис. 27-15). В результате выровнялись ещё два моста: *A-a* и *C-a*, что позволило в следующей *удачной* попытке добавить в паросочетание ребро *C=a*.

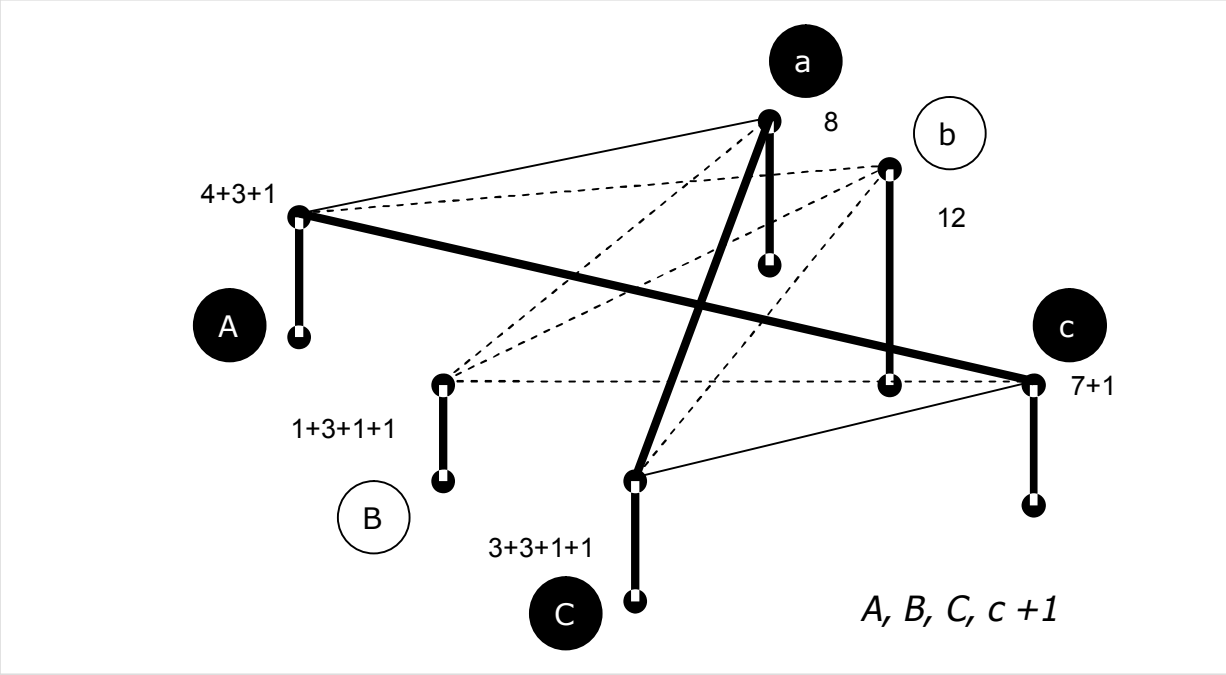


Рис. 27-15 — Поднятие на 1 метр платформы A-B-C-c
и выравнивание мостов A-a и C-a, добавилось ребро C=a

Однако конечная цель пока не достигнута — вершина *B* всё ещё белая, и потому строители попытались (неудачно) найти увеличивающую цепь из этой вершины. Оказалось, что для выравнивания мостов *платформу* (а её составляет сейчас только вершина *B*) надо поднять ещё на **2** метра. В итоге выровнялись ещё два моста: *B-a* и *B-c* (рис. 27-16).

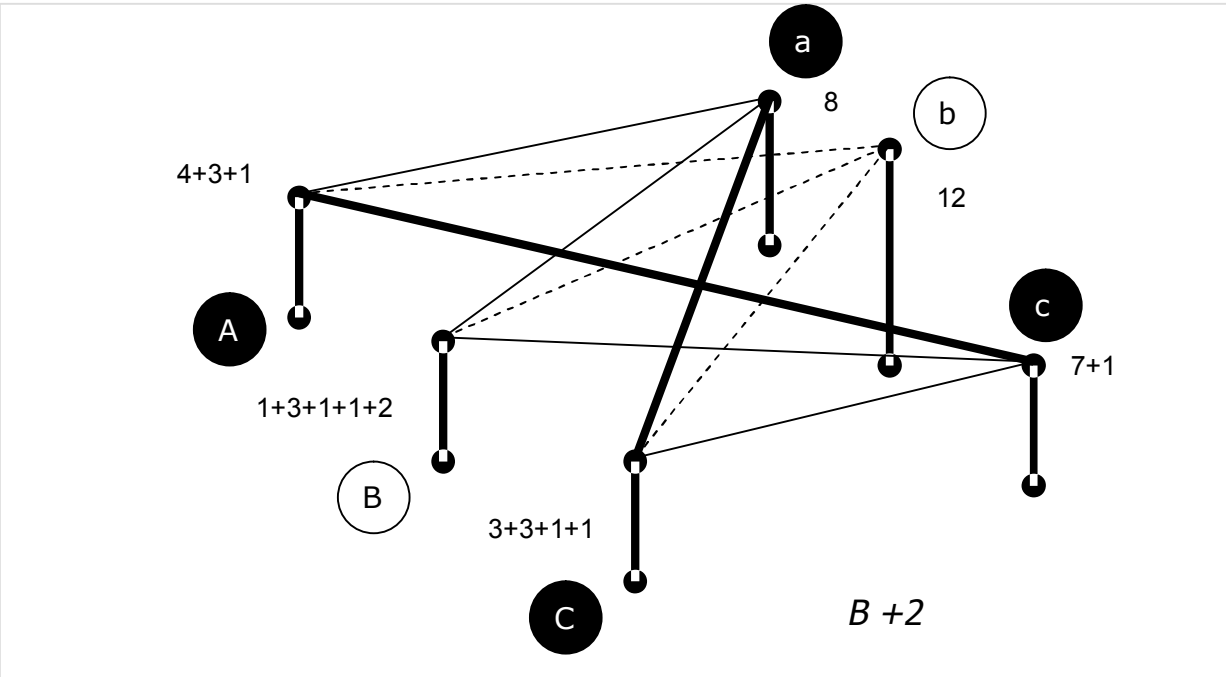


Рис. 27-16 — Поднятие опоры B и выравнивание мостов B-a и B-c

Однако попытка расширить паросочетание вновь оказалась неудачной, но теперь по иной причине: из вершины *B* протянулись две нейтральные *венгерские*

цепи: *B-a=C-c=A* и *B-c=A-a=C*. Отметим, что теперь платформу (множество посещённых вершин) составили все вершины, за исключением вершины *b.* Здесь строители вынуждены были приподнять всю *платформу* ещё на **4** метра, что добавило горизонтальный мост *B-b*. Последняя попытка расширить паросочетание оказалась успешной (рис. 27-17).

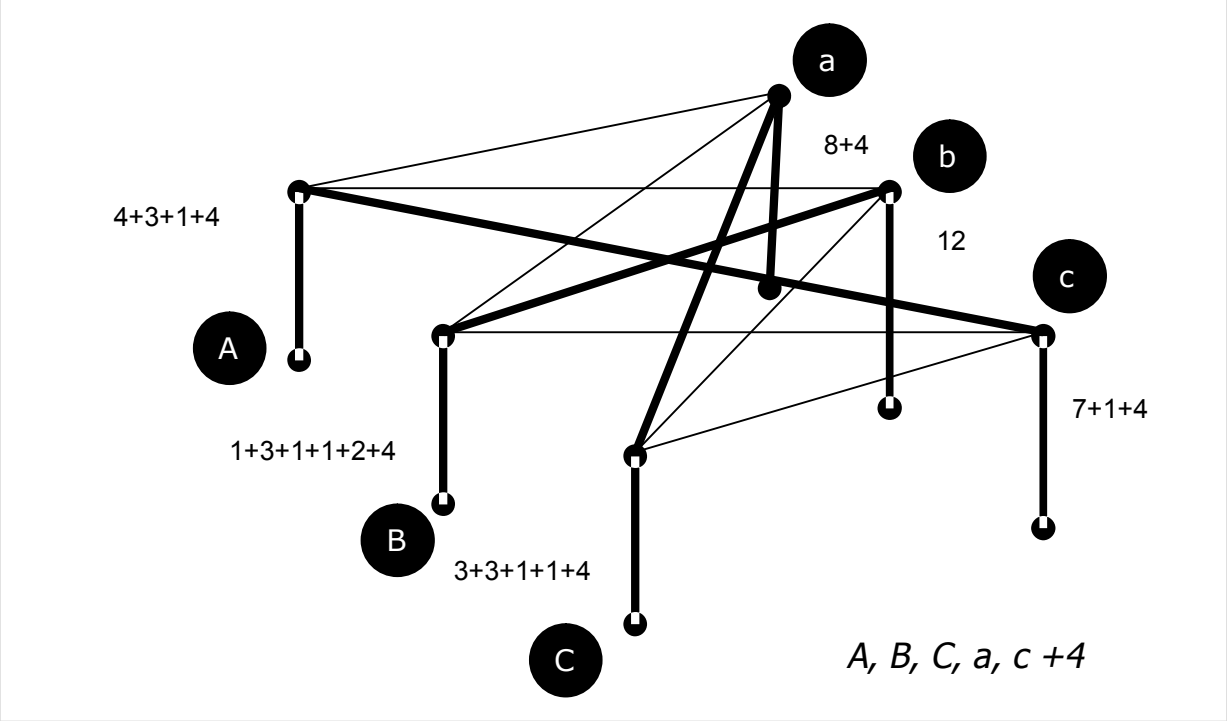


Рис. 27-17 — Поднятие платформы A-B-C-a-c на 4 метра
и выравнивание моста B-b

Пусть расходы на поднятие опор пропорциональны высоте их поднятия, тогда принятый строителями порядок работ должен минимизировать эти расходы. Надо признать, что исходные данные здесь подобраны так, чтобы сработал наглядный мостовой пример, а он не охватывает всё разнообразие графов, многие из которых нельзя интерпретировать мостами. Тем не менее, алгоритм *Куна* применим к любым взвешенным двудольным графам, потому удостоим его здесь формального описания.

От мостов снова перейдём к графу, и заменим крутизну мостов весом рёбер. Вершины уподобим опорам, и припишем им числовое поле **D,** обозначающее их добавочное возвышение относительно исходного положения (в программе для этого служит поле **TNode.mDist**). Тогда текущая «крутизна моста» между вершинами **i** и **j** определится формулой:

Vij = V0ij – (Di-Dj), где

- **V0ij** — исходная крутизна пролёта (поле **TLink.mValue**);
- **Di** — добавочное возвышение левой вершины **i;**
- **Dj** — добавочное возвышение правой вершины **j**.

Напомню, что только рёбра с текущей *нулевой* крутизной участвуют в построении увеличивающих цепей. С учётом всего сказанного, алгоритм Куна можно изложив так.

* Установить исходное состояние: все вершины и рёбра белые, поля возвышения **D** равны нулю.
* Пока не построено *наибольшее* паросочетание, повторять:
  - Снять признаки посещения всех вершин (очистить платформу).
  - Добавить в паросочетание все возможные рёбра, последовательно отыскивая увеличивающие цепи, начиная из белых вершин левой доли. При поиске цепей отметить все *посещённые* вершины (сформировать платформу) и запомнить минимальное возвышение **Delta**, достаточное для выравнивания очередного моста.
  - Если на предыдущем шаге паросочетание не расширилось, поднять платформу на высоту **Delta**, то есть, прибавить это число к полям возвышения **D** всех *посещённых* вершин (платформы).

Здесь вспомогательную процедуру поиска увеличивающих цепей нагрузим двумя дополнительными обязанностями, что практически не усложнит её. Первая состоит в пометке всех посещённых вершин, а вторая — в запоминании минимальной крутизны тех рёбер, которые ещё не участвуют в построении цепей (имеют не нулевую текущую крутизну). В промежуточном итоге дадим список новых терминов этой главы.

Табл. 27-1 — Основные термины для описания алгоритмов поиска паросочетаний

Термин	Пояснение
Паросочетание	Подмножество рёбер такое, что никакие два из них не имеют общей (смежной) вершины.
Максимальное паросочетание	*Паросочетание*, которое нельзя расширить без удаления из него других рёбер. В графе может быть несколько максимальных паросочетаний.
Наибольшее паросочетание	*Паросочетание* максимальной мощности, т.е., включающее в себя наибольшее количество рёбер. Наибольшее паросочетание является и максимальным.
Совершенное паросочетание	*Паросочетание*, в которое входят все вершины графа
Оптимальное паросочетание	*Наибольшее* паросочетание на взвешенных рёбрах минимального либо максимального веса.
Экспонированная (свободная) вершина	Вершина, не принадлежащая паросочетанию.
Альтернирующая цепь (чередующаяся)	Простая цепь, ребра которой попеременно лежат и не лежат в некотором паросочетании.
Аугментальная цепь (увеличивающаяся)	*Альтернирующая* (чередующаяся) цепь, начальная и конечная вершины которой экспонированы.
Венгерская цепь (нейтральная)	*Альтернирующая* (чередующаяся) цепь, только начальная вершина которой экспонирована.
Альтернирующее дерево (чередующееся)	Дерево, построенное из корня, который является *экспонированной* (свободной) вершиной.

27.6. Реализация алгоритма Куна

Дадим назначение полей вершин и рёбер в контексте венгерского алгоритма Куна:

- **TNode.mDist** — показывает степень дополнительного «возвышения» вершины (опоры моста);
- **TNode.mColor** — текущий цвет вершины, чёрные вершины входят в паросочетание;
- **TNode.mFlow** — признак посещения вершины, то есть, признак принадлежности её к платформе (0 или 1);
- **TNode.mRoot** — обратная ссылка (линк) в построенном дереве (ссылка в направлении корня);
- **TNode.mLink** — чёрный линк паросочетания для этой вершины (**nil**, если вершина белая);
- **TLink.mValue** — исходная крутизна ребра (мостового пролёта);
- **TLink.mColor** — текущий цвет ребра, чёрные входят в паросочетание;
- **TLink.mLow** — остаточная крутизна ребра по завершении метода;
- **TLink.mHigh** — признак участия ребра в данном алгоритме, 0 — ребро активно, а иначе считается отключенным.

Последние два элемента перечня будут использоваться алгоритмом Куна в интересах алгоритма поиска кратчайшего Гамильтонова цикла, — это тема 33-й главы. Там от Куна потребуются дополнительные данные, а именно остаточная крутизна ещё не выправленных мостов. Эту крутизну сохраним по выходе из метода в полях **TLink.mLow**. Кроме того, некоторые рёбра (дуги) будут временно изыматься из графа, отключаться. С этой целью поле **TLink.mHigh** будет использовано как признак отключения ребра: 0 — ребро активно, используется; 1— ребро пассивно, отключено. По умолчанию это поле очищено, и все рёбра активны. Всё сказанное влияет на реализацию метода в минимальной степени.

Основной метод, реализующий венгерский алгоритм Куна, назван **MarkMinPairsDicoty**, — он метит чёрным рёбра и вершины *наибольшего* паросочетания *минимальной* стоимости, вычисляет эту стоимость, а также сохраняет остаточную крутизну рёбер в полях **TLink.mLow**.

Листинг 27-1 — Основной метод, реализующий венгерский алгоритм Куна

```
function TGraph.MarkMinPairsDicoty(var aPairs: integer): integer;
var    Que: TBuffer;  // очередь вершин
    NodesL: TSet;     // множество левых вершин
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Выполняется начальная установка двудольного графа:
  // все вершины и дуги белые
  // Формируется вспомогательное подмножество левых вершин
  // Возвращает максимально возможное количество пар

  function  InitNodes: integer;
  var Node: TNode;
```

```
    Link: TLink;
    L, R: integer; // счётчики левых и правых вершин
begin
  // Обнуление счётчиков вершин
  L:= 0; R:= 0;
  // Перебор всех вершин:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mColor:= CWhite;   // цвет вершины белый
    if Node.IsLeft then begin
      Inc(L);               // подсчёт левых вершин
      Node.mDist:= MaxInt;  // вершинное число ищем вторым перебором
      NodesL.Insert(Node);  // и вставка в множество левых
    end else if Node.IsRight then begin
      Inc(R);               // подсчёт правых вершин
      Node.mDist:=0;        // вершинное число = 0
    end;
    Node:= NodeNext;
  end; // while
  // Количество ожидаемых паросочетаний равно меньшему из двух:
  if L > R then Result:= R else Result:= L;
  // Перебор левых вершин:
  Node:= NodesL.GetFirst as TNode;
  while Assigned(Node) do begin
    // Перебор исходящих линков
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do if mHigh=0 then begin
        // это активный линк
        Link.mColor:= CWhite;
        // запоминаем вес минимальной дуги
        if Node.mDist > Link.mValue  then
          Node.mDist:= Link.mValue; // начальное "приподнятие опоры"
      end;
      Link:= Node.OutLinkNext;
    end; // while
    Node:= NodesL.GetNext as TNode;
  end; // while
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск аугментальной (улучшающей) цепи:
// aLeft - корневая экспонированная левая вершина дерева (белая)
// Возвращает экспонированную правую вершину (белую),
// при условии, что найдена аугментальная цепь, а иначе NIL.
// Формирует ссылки в полях TNode.mRoot, ведущие к корню дерева.
// Если экспонированная вершина не найдена,
// то параметр aDelta содержит минимальное приращение цены,
// необходимое для покупки ещё хотя бы одного линка.

function FindAugmenting(aLeft : TNode; var aDelta: integer): TNode;
var Node: TNode;
    Link: TLink;
    Cost: integer;
begin
  Result:= nil;        // ссылка на правую вершину
  Que.Clear;           // очистить очередь вершин
  Que.Put(aLeft);      // и поместить туда корневую вершину
  // Пока очередь не пуста и не обнаружена аугментальная цепь
  while (Que.GetCount > 0) and not Assigned(Result) do begin
    Node:= Que.Get as TNode;
    Node.mFlow:= 1;    // отмечаем посещение вершины
    if Node.IsLeft then begin
```

```
        // Это вершина левой доли
        // Ищем белую вершину в правой доле, а чёрные ставим в очередь
        Link:= Node.OutLinkFirst;
        while Assigned(Link) do begin
          // обрабатываем только активные белые линки
          with Link do if (mHigh=0) and    // активный линк
                          (mColor=CWhite)  // и линк белый
            then begin
              // Вычисляем приведенную (пониженную) стоимость дуги
              Cost:= mValue - (mOwner.mDist - mDest.mDist);
              if Cost = 0 then begin
                // Это достижимая правая вершина (Cost=0)
                if  mDest.mColor = CWhite then begin
                  // Если она белая,
                  // то найдена белая вершина и аугментальная цепь
                  Result:= mDest;       // экспонированная белая вершина
                  Result.mRoot:= Link; // ссылка в направлении корня дерева
                  Break;               // выход из цикла и процедуры
                end else if mDest.mFlow = 0 then begin
                  // Найдена чёрная вершина,
                  // она ещё не посещалась, и потому ставим её в очередь
                  mDest.mRoot:= Link; // ссылка в направлении корня дерева
                  Que.Put(mDest);     // ставим в очередь
                end; // else
              end else begin
                // Это недостижимая правая вершина (Cost > 0),
                // через aDelta возвращаем минимальное приращение цены
                if aDelta > Cost then aDelta:= Cost;
              end;
          end; // with Link
          Link:= Node.OutLinkNext;
        end; // while - конец перебора линков
      end else begin
        // Это внутренняя чёрная вершина в правой доле (Node.mIsLeft = FALSE)
        Link:= Node.mLink;         // извлекаем из неё линк паросочетания
        Link.mOwner.mRoot:= Link; // это ссылка в направлении корня дерева
        // Ставим в очередь найденную внешнюю чёрную вершину в левой доле
        Que.Put(Link.mOwner);
      end;
    end;
  end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Инверсия аугментальной (увеличивающей) цепи.
// Вызывается при условии, что такая цепь обнаружена
// в процедуре FindAugmenting.
// aLeft, aRight -- левая и правая экспонированные (белые) вершины.
// Использует поля Node.mRoot, сформированные процедурой FindAugmenting,
// и формирует из них линки паросочетаний Node.mLink.

procedure Inverse(aLeft, aRight : TNode);
var Node: TNode;
    Link: TLink;
    Flag: boolean;
begin
  // Проходим аугментальную цепь в обратном порядке
  // инвертируя цвета дуг
  Flag:= true;      // инвертируемый флаг определяет цвет ребра (дуги)
  Node:= aRight;
  repeat
    Link:= Node.mRoot as TLink;
    if Flag then begin
      // Переход справа налево, окраска дуги чёрным
```

```
        // Красим белым старый линк паросочетания
        with Node do if Assigned(mLink) then mLink.mColor:= CWhite;
        Node.mLink:= Link;     // запоминаем новый линк паросочетания
        Link.mColor:= CBlack; // и красим его чёрным
        Node:= Link.mOwner;    // переход в сторону корня дерева
        Node.mLink:= Link;     // и здесь запоминаем новый линк паросочетания
      end else begin
        // Переход слева направо, окраска дуги белым
        Link.mColor:= CWhite;
        Node:= Link.mDest;      // переход в сторону корня дерева
      end;
      Flag:= not Flag; // инвертируемый флаг определяет цвет ребра (дуги)
    until Node = aLeft;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка у всех вершин полей TNode.mFlow -- признаков посещения.
  // Вызывается перед обходом свободных (белых) левых вершин

  procedure ClearFlags;
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= 0;    // Это признак посещения при постройке деревьев
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обход свободных (белых, экспонированных) левых вершин
  // с целью построения улучшающих аугментальных цепей.
  // Возвращает количество обнаруженных цепей (и соответственно пар).
  // Если ни одна такая цепь не будет обнаружена,
  // то через параметр aDelta возвращает минимальное "поднятие опоры",
  // необходимое для выпрямления хотя бы одного моста

  function FindPairs(var aDelta: integer): integer;
  var LNode, RNode : TNode;
  begin
    Result:= 0;        // счётчик обнаруженных пар
    aDelta:= MaxInt;   // минимальное приращение цены
    ClearFlags;        // очистка признаков посещения вершин TNode.mFlow
    // Перебор левых вершин
    LNode:= NodesL.GetFirst as TNode;
    while Assigned(LNode) do begin
      if LNode.mColor = CWhite then begin
        // Это открытая (экспонированная) левая вершина
        // Ищем открытую вершину справа и формируем aDelta
        RNode:= FindAugmenting(LNode, aDelta);
        if Assigned(RNode) then begin
          // Аугментальная цепь найдена
          // Инвертируем цвета дуг от RNode к LNode
          Inverse(LNode, RNode);
          // Две крайние вершины цепи метим чёрным
          // как вошедшие в паросочетание
          LNode.mColor:= CBlack;
          RNode.mColor:= CBlack;
          Inc(Result);  // и наращиваем счётчик пар
        end;
      end;
      LNode:= NodesL.GetNext as TNode;
    end;
  end;
```

```
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура корректирует вершинные числа Node.mDist на величину aDelta
  // (приподнимает платформу)
  // Вызывается после обхода экспонированных (белых) вершин
  // после того, как ни одна новая пара не обнаружена.

  procedure Correct(aDelta: integer);
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Если вершина посещалась, то нарастить вершинное число
      if Node.mFlow <> 0 then begin
        Inc(Node.mDist, aDelta);
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт стоимости паросочетания по сумме чёрных линков
  // и формирование пониженной стоимости (крутизны) дуг TLink.mLow
  // с учётом текущих значений вершинных чисел TNode.mDist.
  // (поля TLink.mLow используются при поиске цикла Гамильтона)

  function CalcCost: integer;
  var Node: TNode;
      Link: TLink;
  begin
    Result:= 0;
    // Перебор вершин левой доли
    Node:= NodesL.GetFirst as TNode;
    while Assigned(Node) do begin
      // Node.mLink - это чёрный линк паросочетания
      Inc(Result, Node.mLink.mValue);
      // Перебор исходящих линков
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // если это активный линк, вычисляем редуцированную стоимость
        with Link do
          if mHigh=0 then mLow:= mValue - (mOwner.mDist - mDest.mDist);
        Link:= Node.OutLinkNext;
      end;
      Node:= NodesL.GetNext as TNode;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
   P: integer;      // количество новых пар на очередной итерации
   Delta: integer; // приращение, необходимое для приподнятия платформы
   Count: integer; // счётчик осташихся не найденных пар

begin  { TGraph.SetPairsDicoty }

  Que:= TBuffer.Create; // очередь вершин
  NodesL:= CreateSet;   // множество левых вершин

  Count:= InitNodes; // возвращает максимально возможное число пар
  aPairs:= Count;

  repeat     // повторять до формирования всех паросочетаний
    repeat   // повторять, пока возможно приращение паросочетания
      P:= FindPairs(Delta); // P - количество обнаруженных новых пар
```

```
                                // Delta - миним. приращение для покупки дуги
      Dec(Count, P);            // Count - счётчик оставшихся пар
    until (P=0) or (Count=0);
    // Если паросочетание не полно, то скорректировать вершинные числа
    // на величину Delta (приподнять платформу),
    // "выровняв" тем очередной мост или несколько мостов
    if Count > 0 then Correct(Delta);
  until Count=0;

  // Здесь максимальное паросочетание достигнуто,
  // подсчитать стоимость паросочетания и остатки стоимостей дуг
  Result:= CalcCost;
  // Очистка памяти:
  NodesL.Free;  // множество левых вершин
  Que.Free;     // очередь вершин
end;
```

Отметим несколько моментов. В функции инициализации **InitNodes** формируется вспомогательное множество левых вершин, что в последующем ускоряет их перебор. Принадлежность вершины к левой доле графа определяется методом **TNode.IsLeft**:

```
function TNode.IsLeft: boolean;
begin
  Result:= mLnkOut.GetCount <> 0;
end;
```

То есть, к вершинам левой доли причислены те, из которых исходит хотя бы одна дуга. Разумеется, что это работает только в ориентированном двудольном графе, в иных случаях потребуется другой признак принадлежности к долям. Затем определяем количество пар, которое составит *наибольшее* паросочетание. Им будет меньшее из двух: число вершин в левой, либо в правой доле.

Там же — при инициализации — производим начальный «подъём мостовых опор»: вершинным числам **mDist** в левой доле присваиваем минимальные значения так, чтобы гарантировано «выровнять» хотя бы по одному «мосту», исходящему из левых вершин. Начальное «выравнивание» уменьшает количество последующих итераций.

Следующая функция формирует объект, содержащий множество дуг *наибольшего* паросочетания *минимальной* стоимости, а также вычисляет его стоимость. Здесь предварительно вызывается упомянутый выше метод **MarkMinPairsDicoty**.

Листинг 27-2 — Метод, формирующий множество дуг паросочетания и его стоимость

```
function TGraph.GenMinPairsDicoty: TCostSet;
var Cost: integer;    // стоимость паросочетания
    Pairs: integer;   // количество найденных пар
    Node: TNode;
    S : TSet;
begin
  Cost:= MarkMinPairsDicoty(Pairs);
  S:= CreateSet;
```

```
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft
      then S.Insert(Node.mLink);
    Node:= NodeNext;
  end;
  Result:= TCostSet.Create(Cost, S, true);
end;
```

Испытание метода отложим до знакомства с ещё одним алгоритмом, решающим ту же задачу поиска паросочетания через поток.

27.7. Альтернативное решение через поток

Простота этого метода поиска паросочетания обусловлена использованием уже известного алгоритма построения насыщенного потока минимальной стоимости (см. главу 26). Идея метода в следующем.

Дополним двудольный граф двумя вершинами: искусственным истоком *S*, и искусственным стоком *T*, и соединим их соответственно с левой и правой долями графа дугами *нулевой* стоимости так, как показано на рис. 27-18. Всем дугам полученного графа припишем неделимую *единичную* пропускную способность.

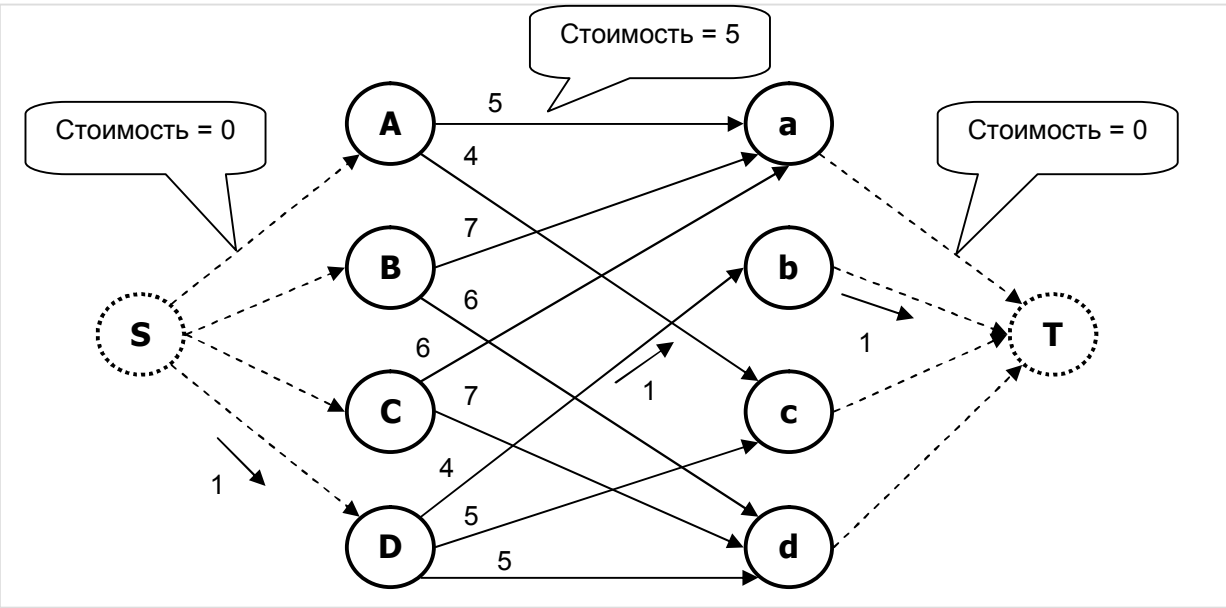


Рис. 27-18 — Дополнение графа вспомогательным истоком S и стоком T, пропускная способность всех дуг равна единице

Найдём здесь распределение наибольшего потока минимальной стоимости. Понятно, что каждая вершина *левой* доли сможет принять от источника *S* и передать дальше только единицу потока (поток не дробится). То же относится и к вершинам *правой* доли: исходящие из них дуги способны отправить в сток *T* лишь по одной единице потока. В дугах исходного графа поток распределится по «дешёвым» дугам, — они то и составят оптимальное в данном случае паросочетание. Отметим, также, что наибольший поток насытит вспомогательные

дуги либо истока, либо стока, либо того и другого сразу (если число вершин в долях одинаково).

В следующем листинге дан метод поиска наибольшего паросочетания минимальной стоимости на двудольном графе с применением потокового алгоритма.

Листинг 27-3 — Поиска наибольшего паросочетания минимальной стоимости через поток

```
function TGraph.GenMinPairsDicotyFlow: TCostSet;
var  S, T : TNode;
     Node : TNode;
     Link : TLink;
     Cost : integer;
      Cnt : integer;
      Res : TSet;
begin
  // Всем дугам графа назначаем единичную пропускную способность
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do begin
        mTemp:= mHigh;  // временно сохраняем
        mHigh:= 1;      // и устанавливаем единичную пропускную способность
      end;
      Link:= Node.OutLinkNext;
    end;
    Node:= NodeNext;
  end;
  // Создаём искусственные исток S и сток R:
  S:= TNode.Create(0, Self);
  T:= TNode.Create(0, Self);

  // Перебираем вершины, соединяя левую долю с истоком, а правую со стоком
  // дугами со стоимостью =0, пропускной способностью =1
  // Одновременно подсчитываем вершины в левой доле
  Cnt:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft then begin
      // Дуга из истока в левую долю,
      Link:= S.MakeLink(Node, 0);
      Link.mHigh:= 1;
      Inc(Cnt);
    end  else begin
      // Дуга из правой доли в сток,
      Link:= Node.MakeLink(T, 0);
      Link.mHigh:= 1;
    end;
    Node:= NodeNext;
  end;  // while

  // Из количества вершин в левой и правой долях выбираем меньшее:
  if Cnt > mNodes.GetCount div 2 then Cnt:= mNodes.GetCount - Cnt;

  // Вставляем в граф:
  InsertNode(S);   // исток
  InsertNode(T);   // сток
```

```
  // Находим поток минимальной стоимости:
  Cost:= CalcMinCostFlow(S, T, Cnt);

  // Удаляем и уничтожаем искусственные исток и сток:
  RemoveNode(T);
  RemoveNode(S);
  T.Free; S.Free;

  Res:= CreateSet;
  if Cost>=0 then begin
    // Здесь паросочетание существует
    // В паросочетание включаем дуги с ненулевым потоком (Link.mFlow <> 0)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do begin
          if mFlow <> 0 then Res.Insert(Link);
          mHigh:= mTemp; // восстанавливаем исходные значения
        end;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
  end; // if
  // Формируем множество со стоимостью:
  Result:= TCostSet.Create(Cost, Res, true);
end;
```

27.8. Испытание методов

Прежде всего, сравним результаты, выдаваемые двумя рассмотренными выше методами поиска паросочетания минимальной стоимости на двудольном графе, этой цели служит следующая программа.

Листинг 27-4 — Программа для сравнения результатов двух методов поиска паросочетания минимальной стоимости

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res : TCostSet;    // результат
    Nodes : integer;   // количество вершин

// Вывод результата в текстовый файл

procedure ResultExpo(const aFile: String);
var Pair : TItem;
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
```

```
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('-----------------------');
  with Res do begin
    if mCost<0 then Exit;
    Pair:= mSet.GetFirst;
    while Assigned(Pair) do begin
      if Pair is TLink then
        with Pair as TLink do Writeln(mOwner.GetName + '-'
                                    + mDest.GetName, ' =', mValue:3)
        else begin Pair.Expo; Writeln end;
      Pair:= mSet.GetNext;
    end;
    Writeln('Cost= ', mCost);
  end;
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;

const CFile = 'Out.txt';

begin
    repeat
      Write('Nodes= '); Readln(Nodes);
      if Nodes < 3 then Break;
      // Создаётся случайный двудольный граф
      Gr:= THugeGraph.GenDicoty(99, Nodes);
      { Либо вводится из текстового файла
      Gr:= TGraphChars.Load('Dicoty-8.txt');
      }
      // Решение через поток:
      Res:= Gr.GenMinPairsDicotyFlow;
      ResultExpo('');  ResultExpo(CFile);
      Res.Free;

      // Решение алгоритмом Куна:
      Res:= Gr.GenMinPairsDicoty;
      ResultExpo('');  ResultExpo(CFile);
      Res.Free;
      Gr.Free;
    until false;
end.
```

На графе, текстовое представление которого дано ниже,

```
Dicoty-8.txt (Kristofides, 408)
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
16 - количество вершин
A B C D E F G H a b c d e f g h
A -> a=13 b=21 c=20 d=12 e= 8 f=26 g=22 h=11
B -> a=12 b=36 c=25 d=41 e=40 f=11 g= 4 h= 8
C -> a=35 b=32 c=13 d=36 e=26 f=21 g=13 h=37
D -> a=34 b=54 c= 7 d= 8 e=12 f=22 g=11 h=40
E -> a=21 b= 6 c=45 d=18 e=24 f=34 g=12 h=48
F -> a=42 b=19 c=39 d=15 e=14 f=16 g=28 h=46
G -> a=16 b=34 c=38 d= 3 e=34 f=40 g=22 h=24
H -> a=26 b=20 c= 5 d=17 e=45 f=31 g=37 h=43
```

обоими методами получено паросочетание стоимостью **76** единиц:

```
A-a = 13
B-h =  8
C-g = 13
D-e = 12
E-b =  6
F-f = 16
G-d =  3
H-c =  5
```

Отметим, что в общем случае методы могут дать разные паросочетания *одинаковой* стоимости.

Теперь сравним быстродействие методов, воспользовавшись следующей программой.

Листинг 27-5 — Программа для сравнения быстродействия методов поиска паросочетания

```
{$APPTYPE CONSOLE}
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
var
  Gr : TGraph;
  Res : TCostSet;    // результат (паросочетание)
  Start: TDateTime; // для засечки времени
  Time : integer;
  Nodes : integer;   // количество вершин
  Retry : integer;   // количество повторений
  i : integer;
  Sum1, Sum2 : integer;

// Вывод результатов

procedure ResultExpo(const aFile: String);
var n: integer;
```

```
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('-----------------------------------------------------');
  Writeln('Nodes=', Nodes:4, ' Retry=', Retry:4);
  Writeln('Flow= ', Sum1/Retry:7:1, ' ms');
  Writeln('Hung= ', Sum2/Retry:7:1, ' ms');
  Writeln('F/H = ', (Sum1+0.1)/(Sum2+0.1) :7:1);
  Writeln('-----------------------------------------------------');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;

begin
  repeat
    Write('Nodes= '); Readln(Nodes);
    if Nodes<3 then Break;
    Write('Retry= '); Readln(Retry);
    if Retry=0 then Break;
    Sum1:= 0; Sum2:= 0;
    for i:= 1 to Retry do begin
      Gr:= THugeGraph.GenDicoty(99, Nodes);
      Write('.');
      // Через поток:
      Start:= Now;
      Res:= Gr.GenMinPairsDicotyFlow;
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum1, Time);
      Res.Free;
      // Венгерским
      Start:= Now;
      Res:= Gr.GenMinPairsDicoty;
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum2, Time);
      Res.Free;
      Gr.Free;
    end;
    Writeln(#7);  ResultExpo(''); ResultExpo('Out.txt');
  until false;
end.
```

В эксперименте порождены по сотне случайных графов размерностью 50, 100, 150 и 200 пар вершин. В табл. 27-2 даны средние времена их обработки.

Табл. 27-2 — Результаты сравнения быстродействия двух методов поиска паросочетания

Количество пар вершин	Среднее время обработки, ms		Отношение времён
	Через поток	Методом Куна	
50	5,7	0,8	6,9
100	32,4	2,9	11,4
150	99,1	5,9	16,8
200	229,6	10,6	21,7

Рис. 27-19 показывает линейный рост преимущества венгерского метода Куна в сравнении с потоковым методом. Быстрый алгоритм Куна будет использован позднее при поиске кратчайшего Гамильтонова цикла.

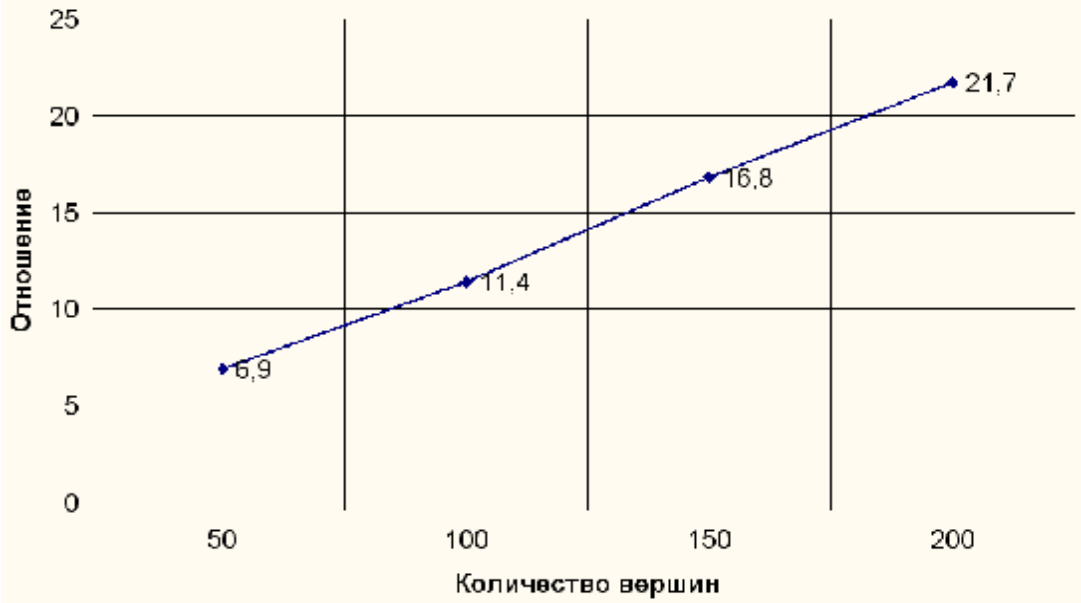


Рис. 27-19 — Соотношение быстродействия потокового метода и метода Куна в зависимости от размера двудольного графа

27.9. Итоги

- Задачи поиска паросочетаний состоят в том, чтобы найти либо наибольшее паросочетание в графе — без учёта веса рёбер, либо какое-то из оптимальных: минимального или максимального веса.
- Быстрые алгоритмы поиска паросочетаний основаны на поиске увеличивающих (аугментальных) цепей.
- Венгерский алгоритм Куна ищет оптимальные паросочетания на двудольных графах. Он использует принцип поиска увеличивающих цепей в комбинации с накоплением особых вершинных чисел.
- Эта же задача на двудольном графе может решаться потоковым алгоритмом, но выполняется он существенно медленней.
- Ввиду эквивалентности двух методов поиска паросочетания на двудольном графе, поиск оптимального потока в нём предпочтительно выполнять более быстрым алгоритмом Куна.

27.10. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 405
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 178
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 28
Паросочетание в произвольном графе

Предыдущая глава познакомила читателя с быстрым алгоритмом Куна для поиска паросочетаний в двудольных графах. Такими графами моделируются асимметричные связи между двумя группами разнородных объектов, но это не исчерпывает всех отношений в нашем мире. Рассмотрим пример, где создаются пары из однородных равноправных элементов.

Пусть имеются **N** сотрудников, и технология производства требует их парной работы. Причём эффективность этой работы зависит от совместимости конкретной пары и может быть оценена числом. Ситуация моделируется графом, где вершины соответствуют сотрудникам, а рёбра — возможным парам, причём вес ребра характеризует эффективность пары. Необходимо создать максимально возможное количество пар, причём так, чтобы добиться *наибольшего* суммарного эффекта. Искомое паросочетание максимальной мощности и *максимального* веса принято называть *оптимальным* или *совершенным*.

В иных ситуациях целью может быть *совершенное* паросочетание *минимального* веса (предположим, для минимизации некоторого ущерба). Иногда бывает достаточно найти паросочетание максимальной мощности без учёта веса рёбер. Отмечу, что все три рода задач решаются одним алгоритмом.

Указанным выше задачам посвятим эту и следующую главы. Конечной целью наших усилий является реализация быстрого алгоритма Эдмонса. Однако этот изощрённый алгоритм изобилует массой тонкостей, и потому нуждается в тщательном тестировании на тысячах случайных графов. С этой целью создадим относительно простой, хотя далеко не самый эффективный алгоритм, основанный на поиске разбиений множества. Его достоинствам является и некая универсальность в том смысле, что при соответствующей модификации он позволяет находить не только 2-сочетания, но и сочетания из **M** связанных вершин, то есть **M**-сочетания (скажем, для создания бригад из **M** человек).

28.1. Постановка задачи

Ещё раз уточним интересующие нас рёбра графа. Рассмотрим рис. 27-1.

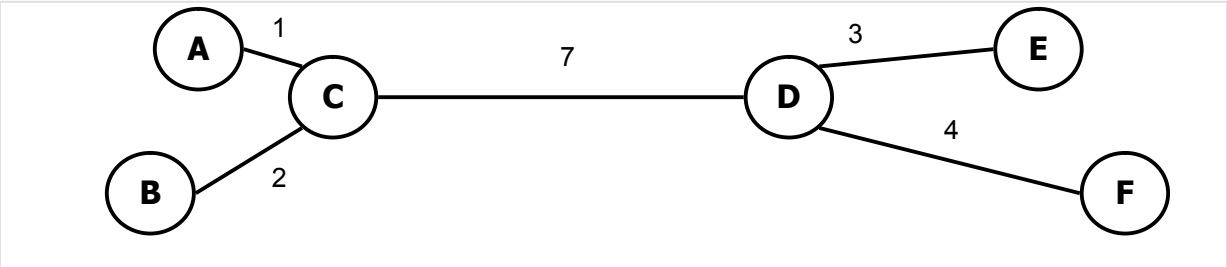


Рис. 28-1 — Граф с взвешенными рёбрами

Если не учитывать веса рёбер, здесь можно найти четыре паросочетания максимальной мощности по два ребра в каждом:

```
1) A-C, D-E
2) A-C, D-F
3) B-C, D-E
4) B-C, D-F
```

Одно из них является паросочетанием *минимального* веса, другое — паросочетанием *максимального* веса (см. рис. 28-2 и рис. 28-3).

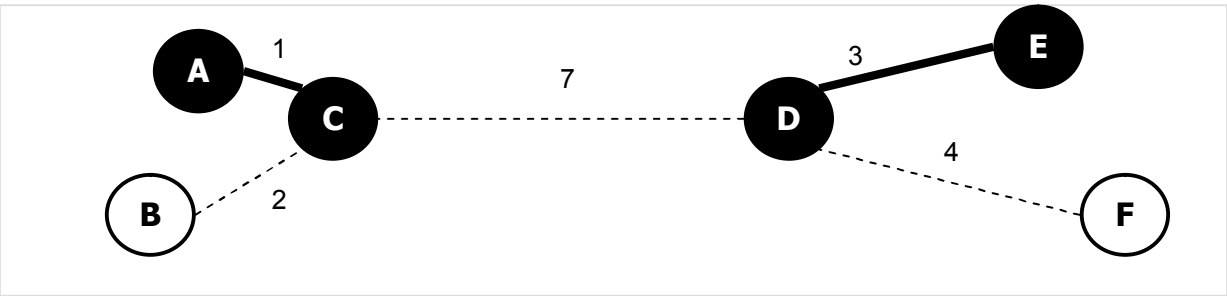


Рис. 28-2 — Наибольшее паросочетание минимального веса (1+3=4)

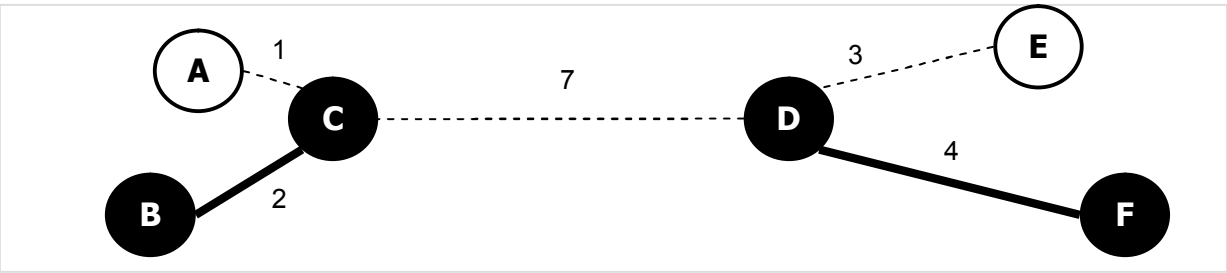


Рис. 28-3 — Наибольшее паросочетание максимального веса (2+4=6)

Отметим, что отдельное ребро *C-D* тоже является паросочетанием, причём вес его максимален (7 против 6). Однако это паросочетание *не является совершенным*, и потому не будет целью будущего алгоритма.

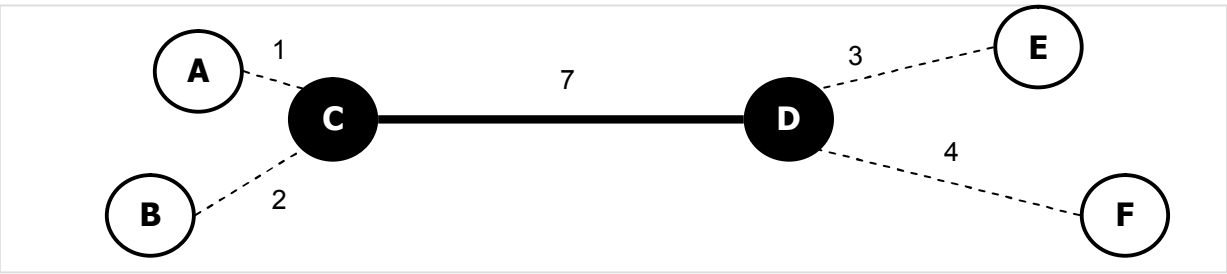


Рис. 28-4 — Паросочетание максимального веса (не наибольшее)

Итак, предстоящая задача формулируется следующим образом. Дан граф со взвешенными рёбрами. В зависимости от заданного параметра поиска, надо найти в нём одно из паросочетаний: a) *максимальной мощности* без учёта веса рёбер или дуг; b) *совершенное минимального* веса; c) *совершенное максимального* веса. Если существует несколько решений одинаковой стоимости, алгоритм должен выдать любое из них.

28.2. Основные идеи

Алгоритм, основанный на разбиении множества, исходит из того, что пара — это подмножество паросочетания. Каждому подмножеству приписан вес соединяющего пару ребра. В паросочетании эти подмножества не должны взаимно пересекаться, стало быть, можно подобрать нужную комбинацию *перебором* всех возможных подмножеств, составленных из пар вершин (или подмножеств из **м** вершин — в общем случае). В итоге решение сводится к операциям с множествами.

Известно, что перебор подмножеств — экспоненциально сложная задача. В 10-й главе для ускорения перебора применена группировка подмножеств в блоки и досрочный возврат в ходе их перебора. Подобным образом поступим и здесь.

Сначала сгруппируем все рёбра (т.е. возможные пары) в блоки так, чтобы в каждом блоке оказались только рёбра, инцидентные одной из вершин, эту вершину назовём *меткой* блока. Так, к примеру, в первый блок с меткой *A* поместим все рёбра, инцидентные вершине *A*. В следующий блок попадут все рёбра, инцидентные вершине *B*, за исключением тех, что уже содержатся в первом блоке. В блоке с меткой *C* соберутся рёбра, инцидентные этой вершине, за исключением тех, что содержатся в предыдущих двух блоках, и т.д. Таким образом, общее количество рёбер в блоках будет равно их количеству в графе.

Далее в ходе перебора из каждого блока будет отбираться лишь по одному ребру, что на порядки снизит трудоёмкость. Для ускорения перебора упорядочим рёбра внутри блоков и сами блоки некоторым образом. Так, для поиска паросочетания минимального веса (сведём всё к этому варианту) рёбра внутри блока отсортируем в порядке не убывания их веса. Блоки (множества подмножеств) упорядочим в порядке не убывания количества содержащихся в них рёбер с тем, чтобы блоки меньшей мощности обрабатывались раньше. Подробнее о формировании блоков будет сказано ниже.

Распределение рёбер по блокам позволяет сократить перебор за счёт досрочного возврата к блоку вышестоящего уровня. Для этого после обработки очередного блока необходимо получить быстрый прогноз: может ли обработка последующих блоков улучшить достигнутый на данный момент результат? Так, при поиске паросочетания *минимального* веса прогноз будет положительным, если соблюдается одно из условий:

- Обработка последующих блоков может увеличить мощность паросочетания в сравнении с текущим лучшим;
- Обработка последующих блоков может дать паросочетание той же мощности, что и текущее, но меньшей стоимости.

Разумеется, что *положительный* прогноз будет заведомо оптимистичным, то есть, не будет гарантировать улучшения результата. Зато *отрицательный* прогноз будет абсолютно надёжным и отсечёт бесперспективные ветви перебора.

Рассмотрим способ быстрого получения такого прогноза. Напомню, что мы перебираем рёбра, сгруппированные в блоки с метками вершин *A*, *B*, *C* и т.д. Без потери общности можно считать, что блоки перебираются в алфавитном порядке. В ходе перебора накапливается очередное паросочетание и множество вершин, входящих в него. Стало быть, после включения в текущее паросочетание очередного ребра, известна мощность накопленного паросочетания, его стоимость, и множество входящих в него вершин. То же самое известно о текущем оптимальном паросочетании (если оно уже получено). Пусть на момент прогноза после обработки блоков с метками *A* и *B* в накопитель включены рёбра *A-N* и *B-M*. Тогда в ходе погружения в рекурсию блоки *N* и *M* будут пропущены, а из остальных блоков, начиная с блока *C*, для прогноза можно взять по одному самому «дешёвому» ребру. Сложив их стоимость со стоимостью накопленного паросочетания, получим искомый прогноз. Остаётся сравнить его с текущим лучшим решением по критериям, указанным выше.

Итак, в данном алгоритме будут использованы следующие приёмы:

- группировка рёбер в блоки так, чтобы каждое ребро входило лишь в один блок;
- сортировка рёбер внутри блока в порядке не убывания веса;
- сортировка блоков в порядке не убывания размеров блоков;
- рекурсивный перебор рёбер в блоках;
- быстрое вычисление прогноза для отсечения заведомо бесперспективных ветвей перебора.

28.3. О формировании блоков

Блоки — подмножества рёбер — должны вмещать в себя все рёбра графа. В неориентированном графе ребро представлено двумя встречными линками одинаковой длины, но в блок будет включен один из них. В ориентированном графе две вершины тоже могут быть связаны двумя встречными дугами, в общем случае разной длины. Здесь в блок попадёт лишь одна из них: большей или меньшей стоимости — в зависимости от решаемой задачи.

Отметим, что время перебора зависит не только от общего количества рёбер, но и от распределения их по блокам. Предположим, имеется возможность создать N блоков по M рёбер в каждом. Тогда произведение NxM будет равно количеству рёбер в графе, а количество операций полного перебора рёбер составит M^N. То есть, трудоёмкость перебора имеет полиномиальную зависимость от числа рёбер в блоках M, и экспоненциальную зависимость от количества блоков N. Таким образом, предпочтительней иметь дело с малым числом крупных блоков, чем с большим количеством мелких.

Поскольку каждое ребро может быть включено лишь в один из двух возможных блоков, распределение рёбер по ним зависит от порядка обработки вершин. Рассмотрим пример на рис. 28-5.

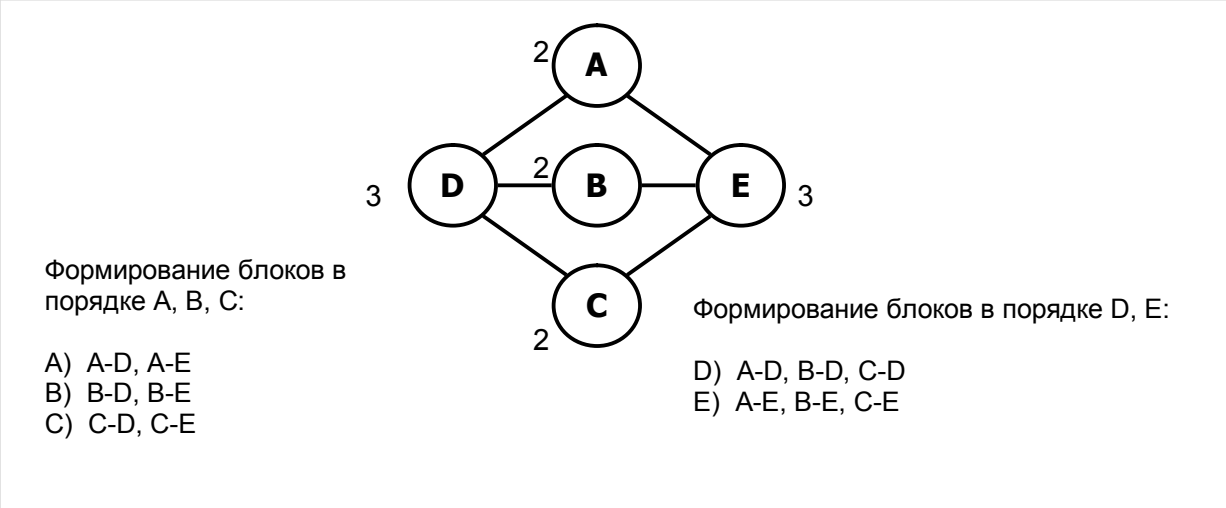


Рис. 28-5 — Состав и количество блоков зависят от порядка их формирования

Если в ходе формирования блоков обрабатывать вершины в алфавитном порядке, то будут порождены три блока по два ребра в каждом (на рисунке слева). Если же формировать блоки в порядке *D*, затем *E*, то получим два блока по три ребра в каждом (справа).

Для укрупнения блоков и уменьшения их количества поступим так. Перед формированием блоков окрасим все линки чёрным, а каждой вершине припишем число, равное количеству смежных с нею чёрных линков (исходящих и входящих). Назовём это *степенью* вершины, начальные значения степеней показаны на рис. 28-5 рядом с вершинами. Выберем одну из вершин с максимальной степенью (вершину *D*) и сформируем блок с этой меткой, включив в него все чёрные линки, инцидентные данной вершине. По мере вставки линков в блок, степени инцидентных им вершин будем уменьшать всякий раз на единицу. И тогда мощность вершины *D* снизится с трёх до нуля, а степени трёх соседних с нею вершин (*A*, *B* и *C*) — с двух до единицы. Затем выберем следующую вершину с максимальной текущей степенью (вершину *E*) и повторим с нею те же действия, в результате её степень и степени соседних вершин снизятся до нуля. В конце концов, формирование блоков остановится, когда все линки будут распределены, а степени всех вершин обнулятся.

Таким образом, выбирая всякий раз для формирования блока вершину с наибольшей текущей степенью и попутно корректируя эту степень в соседних вершинах, получим максимально крупные блоки, и минимальное их количество. Обсудив основные идеи, обратимся теперь к деталям алгоритма, и вначале опишем вспомогательные объекты: подмножество вершин (пару), и блок.

28.4. Вспомогательные объекты

Для представления пары вершин (линка) построим тип на базе класса **`TCostSet`**, который уже содержит в себе поле подмножества **`mSet`** и его стоимость **`mCost`**. Представление пары множеством позволяет при необходимости

легко видоизменить алгоритм с целью поиска в графе **M**-сочетаний. Метод сравнения обеспечивает сортировку пар внутри блока по не убыванию стоимости.

Листинг 28-1 — Объект, представляющий пару вершин (ребро, дугу)

```
type TPair = class (TCostSet)
       function Compare(arg: TItem): TCompare;  override;
       procedure Print(var aFile: TextFile); override;
     end;

function TPair.Compare(arg: TItem): TCompare;
begin
  if Self = arg then begin Result:= cmpEq; Exit; end;
  if mCost = (arg as TCostSet).mCost then begin
    // При равной цене сравнением множества
    Result:= mSet.Compare((arg as TCostSet).mSet);
  end else begin
    // Если цены не совпали, сравнениваем по цене (дешёвые - в начало)
    if mCost > (arg as TCostSet).mCost
      then Result:= cmpGreate
      else Result:= cmpLess
  end;
end;
```

Следующий объект представляет блок линков; он построен на базе множества, и содержит дополнительные поля: метку блока (вершину) и стоимость самого «дешёвого» линка в множестве.

Листинг 28-2 — Блок пар (линков)

```
type  TPairsBlock = class (TSetList)
        mLabel : TNode;     // метка блока (центральная вершина)
        mCost  : integer;   // вес наиболее лёгкой пары в блоке
        function Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: Text); override;
        destructor Destroy;  override;
      end;

destructor TPairsBlock.Destroy;
begin
  ClrAndDestroy;  // очистка с уничтожением элементов
  Inherited;
end;

// Блоки сортируются в порядке не убывания стоимости
// самой "лёгкой" пары в блоке

function TPairsBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;
  if Self = arg then Result:= cmpEq;
  if mCost > (arg as TPairsBlock).mCost then Result:= cmpGreate
end;
```

28.5. Метод поиска паросочетаний

В следующем листинге дан метод поиска паросочетаний, перебирающий подмножества. Основными его элементами являются функция построения блоков

GenBlocks, рекурсивная процедура перебора **BlockHandle**, и функция предсказания **Prediction**.

Листинг 28-3 — Метод поиска паросочетаний в произвольном графе

```
// Метод генерации паросочетаний (п/с) трёх видов:
// pMaxN -- максимальной мощности (без учёта веса рёбер)
// pMaxW -- максимального веса
// pMinW -- минимального веса
function TGraph.GenPairs(aMode: TPairs): TCostSet;

var Blocks: TSet;          // Блоки пар (рёбер)
    BestCost: integer;     // Текущая лучшая (наименьшая) сумма
    BestCnt: integer;      // Текущая лучшая (наибольшая) мощность п/с
    Pairs: TCostSet;       // Текущее паросочетание
    Nodes: TSet;           // Текущее множество присоединённых вершин
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инициализация вершин и дуг
  procedure InitNodes;
  var Node: TNode;
      LD, LR : TLink; // прямой и обратный линки
  begin
    // Предварительная очистка поля мощности
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mPower:= 0;  // здесь будут подсчитываться смежные линки
      Node:= NodeNext;
    end;
    // Перебор всех линков с целью отбора наименьших или наибольших
    // (в случае наличия встречных линков)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        // Красим вершину белым
        mColor:= CWhite;
        // Перебор прямых линков LD
        LD:= OutLinkFirst;
        while Assigned(LD) do begin
          // Находим LR -- встречный линк
          LR:= LD.mDest.GetLink(Node);
          if Assigned(LR) then begin
            // Встречный линк найден
            if (aMode=pMaxW) and (LR.mValue < LD.mValue) then begin
              // При поиске максимума берём больший
              LD.mColor:= CBlack; // будет обрабатываться
              LR.mColor:= CWhite; // НЕ будет обрабатываться
            end else begin
              // При поиске минимума берём меньший
              LR.mColor:= CBlack; // будет обрабатываться
              LD.mColor:= CWhite; // НЕ будет обрабатываться
            end;
          end else begin
            // встречный линк НЕ найден
            LD.mColor:= CBlack; // будет обрабатываться прямой линк
          end;
          // Подсчёт инцидентных линков в двух соседних вершинах
          Inc(LD.mOwner.mPower);
          Inc(LD.mDest.mPower);
          LD:= OutLinkNext;
        end;
      end;
```

```
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск вершины, способной дать блок максимального размера
  function GetBestNode: TNode;
  var Node: TNode;
      Max : integer;
  begin
    Max:= 0; Result:= nil;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do if (mColor=CWhite)     // вершина ещё не обработана
                      and (mPower > Max) // и её текущая мощность больше
      then begin
        Result:= Node;  // очередная лучшая вершина
        Max:= mPower;   // текущее число смежных чёрных линков
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование всех возможных пар вершин (рёбер)
  // и группировка их в блоки

  function GenBlocks: TSet;
    // Создание пары и вставка её в блок:
    function MakePair(aLink: TLink): TPair;
    begin
      Result:= TPair.CreateEmpty;
      with aLink do begin
        // В множество пары вставляем смежные с линком вершины:
        Result.mSet.Insert(mOwner);
        Result.mSet.Insert(mDest);
        // Уменьшаем мощности соседних вершин:
        Dec(mOwner.mPower);
        Dec(mDest.mPower);
      end;
      // Цену (вес) пары формируем в зависимости от варианта п/с
      case aMode of
        pMaxN: Result.mCost:=  1;               // наибольшее
        pMinW: Result.mCost:=  aLink.mValue;    // минимального веса
        pMaxW: Result.mCost:= -aLink.mValue;    // максимального веса ***
      end;
    end;
  var Node: TNode;          // текущая вершина
      Link: TLink;          // текущий исходящий линк
      Pair: TPair;          // текущая пара
      Min: integer;         // наименьший вес пары в блоке
      Block: TPairsBlock;   // текущий блок

  begin  { GenBlocks }
    InitNodes;            // инициализация полей mPower
    Result:= CreateSet;   // множество блоков
    // Перебор вершин:
    Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
    while Assigned(Node) do begin
      Min:= MaxInt;     // здесь формируется вес самой лёгкой пары в блоке
      Block:=nil;       // текущий блок пока не создан

      // Перебор исходящих линков:
      Link:= Node.OutLinkFirst;
```

```
      while Assigned(Link) do begin
        // Если линк не обработан
        if Link.mColor = CBlack then begin
          Link.mColor:= CWhite;  // восстанавливаем цвет
          Pair:= MakePair(Link); // создаём пару
          // Запоминаем вес легчайшей пары в блоке
          if Min > Pair.mCost then Min:= Pair.mCost;
          // Если блок ещё не создан, то создаём:
          if not Assigned(Block) then Block:= TPairsBlock.Create;
          Block.Insert(Pair);  // вставляем пару в блок
        end;
        Link:= Node.OutLinkNext;  // следующий линк
      end; // while

      // Перебор входящих линков:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        // Если линк не обработан
        if Link.mColor = CBlack then begin
          Link.mColor:= CWhite;  // восстанавливаем цвет
          Pair:= MakePair(Link); // создаём пару
          // Запоминаем вес легчайшей пары в блоке
          if Min > Pair.mCost then Min:= Pair.mCost;
          // Если блок ещё не создан, то создаём:
          if not Assigned(Block) then Block:= TPairsBlock.Create;
          Block.Insert(Pair);  // вставляем пару в блок
        end;
        Link:= Node.InLinkNext;  // следующий линк
      end; // while
      // Если блок создан, вставляем в множество результата:
      if Assigned(Block) then begin
        Block.mLabel:= Node;   // метка блока
        Block.mCost:= Min;     // вес легчайшей пары в блоке
        Result.Insert(Block);  // вставка блока
      end;
      Node.mColor:= CBlack; // отмечаем, что вершина обработана
      Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура обработки блока

  procedure BlockHandle(aBlock: integer;  // номер блока
                        aNodes: TSet;     // накопленные вершины
                        aRes: TCostSet    // накопленные пары
                        );
  var Block: TPairsBlock; // текущий блок
      Nodes: TSet;        // Накопленное множество вершин
      Res  : TCostSet;    // Накопленное множество пар (рёбер)
      Pair : TPair;       // очередная пара из блока
      // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      // Прогноз результата

      function Prediction(aBlock: TPairsBlock;  // текущий блок
                          aNodes: TSet;         // накопленные вершины
                          aRes: TCostSet        // накопленные пары
                          ): boolean;
          Cost, Count : integer;  // прогнозируемые цена и мощность п/с
      begin
      var B: TPairsBlock;         // очередной блок
        Result:= not Assigned(aRes);
        if Result then Exit;
```

```
        // Начинаем накопление с текущей стоимости и мощности
        Cost:= aRes.mCost;  Count:= 0;
        if Assigned(aRes.mSet) then Count:= aRes.mSet.GetCount;
        // Пропускаем блоки вплоть до текущего
        B:= Blocks.GetFirst as TPairsBlock;
        while B <> aBlock do B:= Blocks.GetNext as TPairsBlock;
        // Переход к следующему блоку после текущего
        B:= Blocks.GetNext as TPairsBlock;
        // Перебор оставшихся блоков
        while Assigned(B) do begin
          if not aNodes.Exist(B.mLabel) then begin
            Inc(Count);          // накопление мощности паросочетания
            Inc(Cost, B.mCost); // накопление прогнозируемой стоимости
          end;
          B:= Blocks.GetNext as TPairsBlock;
        end;
        // Формируем положительный результат,
        // если выполняется одно из условий:
        // - есть шанс увеличить мощность паросочетания, или
        // - при той же мощности уменьшить стоимость паросочетания
        Result:= (Count > BestCnt) or
                 (Count = BestCnt) and (Cost < BestCost);
      end;
  begin { BlockHandle }
    // Извлекаем текущий блок:
    Block:= Blocks.GetItem(aBlock) as TPairsBlock;
    // Если метка текущего блока НЕ содержится в накопителе вершин,
    // то перебираем пары текущего блока
    if not aNodes.Exist(Block.mLabel) then begin
      // Вершина, которою помечен блок, ещё не входит в паросочетание
      // Перебор пар текущего блока:
      Pair:= Block.GetFirst as TPair;
      while Assigned(Pair) do begin
        // Попытка прилепить очередную пару (ребро)
        // Пара присоединяется, если множество пары
        // не пересекается с текущим множ. чёрных вершин
        if  not aNodes.TestIntersect(Pair.mSet) then begin
          // Здесь пару можно присоединить
          Nodes:= aNodes.Copy as TSet;      // копия накопленных вершин
          Res:= aRes.Copy as TCostSet;      // копия накопленных пар
          // Пристраиваем к множеству пар очередную пару
          Nodes.Add(Pair.mSet);    // накапливаем множество вершин
          Res.Insert(Pair);        // накапливаем множество пар
          if aBlock < Blocks.GetCount then begin
            // Это не последний блок.
            // Если обработка последующих блоков может увеличить
            // мощность паросочетания или снизить цену,
            // то войти в следующий блок
            if Prediction(Block,Nodes,Res)
              then BlockHandle(aBlock+1,Nodes,Res);
          end else begin
            // Достигнут последний блок (aBlock = Blocks.GetCount),
            // Запоминаем результат, если выполняется одно из условий:
            // - текущий результат Result ещё не определён
            // - мощность нового п/с превышает мощность текущего
            // - мощности п/с равны, но стоимость нового меньше
            if not Assigned(Result) or
               (Res.mSet.GetCount > BestCnt) or
               (Res.mSet.GetCount = BestCnt) and (BestCost > Res.mCost)
              then begin
                BestCost:= Res.mCost;          // лучшая (меньшая) стоимость
                BestCnt:= Res.mSet.GetCount;   // текущая лучшая мощность п/с
```

```
                Result.Free;                    // удаляем прежнее п/сочетание
                Result:= Res.Copy as TCostSet;// и сохраняем новое
              end;
          end;
          // Перед выбором следующей пары в блоке освобождаем текущие:
          Res.Free;    // накопленные пары
          Nodes.Free; // накопленные вершины
        end; // if
        Pair:= Block.GetNext as TPair;  // следующая пара в блоке
      end; // while
    end; // if
    // Если ищется наибольшее паросочетание без учёта веса,
    // и оно найдено, то выход
    if (aMode = pMaxN) and (BestCnt = Self.Nodes div 2) then Exit;
    // После перебора блока входим в следующий, если он существует
    if aBlock < Blocks.GetCount then begin
      // Текущий блок не последний.
      // Если обработка последующих блоков может увеличить
      // мощность паросочетания или снизить цену, то войти в следующий блок
      if Prediction(Block,aNodes,aRes)
        then BlockHandle(aBlock+1,aNodes,aRes);
    end else begin
      // Достигнут последний блок (aBlock = Blocks.GetCount),
      // Запоминаем результат, если выполняется одно из условий:
      // - текущий результат Result ещё не определён
      // - мощность нового п/с превышает мощность текущего
      // - мощности п/с равны, но стоимость нового меньше
      if not Assigned(Result) or
         (aRes.mSet.GetCount > BestCnt) or
         (aRes.mSet.GetCount = BestCnt) and (BestCost > aRes.mCost)
        then begin
          BestCost:= aRes.mCost;          // лучшая (меньшая) стоимость
          BestCnt:= aRes.mSet.GetCount;   // текущая лучшая мощность п/с
          Result.Free;                    // удаляем прежнее паросочетание
          Result:= aRes.Copy as TCostSet;// и сохраняем новое
        end;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инверсия знаков весов рёбер паросочетания
  // (используется при поиске паросочетания максимального веса)

  procedure RestoreCost;
  var Pair : TPair;
  begin
    // Перебираем множество пар паросочетания Result.mSet
    with Result.mSet do begin
      Pair:= GetFirst as TPair;
      while Assigned(Pair) do begin
        Pair.mCost:= -Pair.mCost;
        Pair:= GetNext as TPair;
      end;
    end;
    Result.mCost:= -Result.mCost;  // итоговая стоимость всех пар п/с
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Перед уничтожением блоков удаляем из них найденные паросочетания

  procedure RemoveFromBlocks;
  var  Block: TPairsBlock;
  begin
    with Blocks do begin
```

```
      Block:= GetFirst as TPairsBlock;
      while Assigned(Block) do begin
        Block.Sub(Result.mSet);
        Block:= GetNext as TPairsBlock;
      end;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin
  Result:= nil;         // Результат -- лучшее паросочетание
  Blocks:= GenBlocks;   // Создаём блоки
  if Blocks.GetCount = 0 then begin Blocks.Free; Exit; end;

  // Подготовка переменных:
  BestCost:= MaxInt;             // текущая лучшая сумма
  BestCnt:= 0;                   // текущая лучшая (наибольшая) мощность п/с
  Nodes:= CreateSet;             // множество вершин паросочетания
  Pairs:= TCostSet.CreateEmpty;  // начальное паросочетание

  // Рекурсивная обработка блоков:
  //Start:= Now;     // Время входа в процедуру
  BlockHandle(1, Nodes, Pairs);  // формирует Result

  // Очистка памяти:
  Pairs.Free;         // Начальное паросочетание (пустое)
  Nodes.Free;         // Начальное множество присоединённых вершин
  // Удаляем из блоков множество Result.mSet
  if Assigned(Result) then  RemoveFromBlocks;
  Blocks.ClrAndDestroy;  // Удаляем сами блоки
  Blocks.Free;           // и множество блоков

  // После поиска паросочетания максимальной стоимости
  // восстанавливаем стоимости линков:
  if aMode = pMaxW then RestoreCost;
end;
```

Остановим своё внимание ещё на нескольких моментах.

Перед созданием блоков все линки красятся чёрным, а после включения линка в блок, окраска снимается, что исключает повторное использование линка при обработке другой инцидентной ему вершины.

При формировании веса пары учитывается параметр, задающий тип искомого паросочетания: для паросочетания минимального веса вес пары не меняется, для максимального веса — инвертируется знак веса. Для поиска паросочетаний без учёта веса рёбер вес ребра принимается равным единице.

Блоки перебираются рекурсивной процедурой **BlockHandle**. Здесь при поиске паросочетания без учёта веса рёбер использовано ещё одно условие досрочного выхода:

```
    // Если ищется наибольшее паросочетание без учёта веса,
    // и оно найдено, то выход
    if (aMode = pMaxN) and (BestCnt = Self.Nodes div 2) then Exit;
```

Наконец, по завершении поиска, те пары, что попали в множество **Result**, надо удалить из блоков, иначе они будут уничтожены вместе с этими блоками при очистке памяти:

```
  if Assigned(Result) then  RemoveFromBlocks;
```

28.6. Испытания

В следующем листинге дана тестирующая программа, находящая в графе паросочетания трёх видов: a) без учёта веса рёбер, b) минимального веса, и c) максимального веса.

Листинг 28-4 — Программа для поиска паросочетаний трёх видов

```
{$APPTYPE CONSOLE}
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res : TCostSet;
    Nodes: integer;

procedure Handle(arg: TPairs);
var Start: TDateTime;
    Time : integer;
begin
  Start:= Now;
  Res:= Gr.GenPairs(arg);
  Time:= MilliSecondsBetween(Start, Now);
  Res.Expo;
  Writeln('Nodes     = ', Gr.Nodes);
  Writeln('Time (ms)=', Time:6);
  Res.mSet.ClrAndDestroy;
  Res.Free;
end;
begin
      Gr:= TGraphChars.Load('Test.txt');
      Writeln('--------------- MaxN -----------------');
      Handle(pMaxN);
      Writeln('--------------- MinW -----------------');
      Handle(pMinW);
      Writeln('--------------- MaxW -----------------');
      Handle(pMaxW);
end.
```

В

табл. 28-1 даны паросочетания для следующего неориентированного графа:

```
Неориентированный граф из 15 вершин и 52 рёбер
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
15 - количество вершин
A B C D E F G H I J K L M N O
A -> B=11 I= 1 K=98 M=14
B -> A=11 E=63 L=50
C -> E=47 H=87 I=50 M=10
D -> K=30 M=28
E -> B=63 C=47 L=70 N=37
F -> H=82
G -> J=74
H -> C=87 F=82 N=57 O=88
I -> A= 1 C=50 K=21 O=18
J -> G=74 L=77 O=99
K -> A=98 D=30 I=21 M=31 O=34
L -> B=50 E=70 J=77 M=38 N=59
M -> A=14 C=10 D=28 K=31 L=38
N -> E=37 H=57 L=59
O -> H=88 I=18 J=99 K=34
```

Табл. 28-1 — Результаты поиска паросочетаний в графе из 15 вершин

Вид паросочетания	Вес паросочетания	Множество рёбер
Без учёта веса рёбер	7 (мощность п/с)	A-B = 1 C-E = 1 D-K = 1 F-H = 1 G-J = 1 I-O = 1 L-M = 1
Минимального веса	262	C-M = 10 A-B = 11 I-O = 18 D-K = 30 E-N = 37 G-J = 74 F-H = 82
Максимального веса	479	J-O = 99 A-K = 98 F-H = 82 B-E = 63 L-N = 59 C-I = 50 D-M = 28

При поиске паросочетания минимального веса сформированы 9 блоков, они показаны в табл. 28-2, а звёздочками отмечены рёбра, вошедшие в паросочетание.

Табл. 28-2 — Блоки, сформированные при поиске минимального паросочетания

№	Метка блока	Мощность блока (количество рёбер)	Вес легчайшего ребра	Подмножество рёбер
1	M	1	28	D-M = 28
2	J	1	74	G-J = 74 *
3	E	2	37	E-N = 37 * B-E = 63
4	O	2	18	I-O = 18 * J-O = 99
5	A	3	1	A-I = 1 A-B = 11 * A-M = 14
6	H	3	57	H-N = 57 F-H = 82 * H-O = 88
7	C	4	10	C-M = 10 * C-E = 47 C-I = 50 C-H = 87
8	K	5	21	I-K = 21 D-K = 30 * K-M = 31 K-O = 34 A-K = 98
9	L	5	38	L-M = 38 B-L = 50 L-N = 59 E-L = 70 J-L = 77

Перебор — занятие весьма трудоёмкое, и потому важно нащупать границы применимости этого алгоритма. Следующая программа определяет временные характеристики на графах трёх типов: a) полных, b) двудольных (все вершины долей соединены), и c) случайных с плотностью связей 20%.

Листинг 28-5 — Программа для исследования временных характеристик

```
{$APPTYPE CONSOLE}
uses
  SysUtils,  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
var
  Gr : TGraph;
  Res : TCostSet;    // результат (паросочетание)
  Start: TDateTime; // для засечки времени
  Time : integer;
  Nodes : integer;  // количество вершин
```

```
  Retry : integer;  // количество повторений
  i : integer;
  Sum1, Sum2 : integer;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вывод результатов

procedure ResultExpo(const aFile: String);
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('----------------------------------------------------');
  Writeln('Nodes=', Nodes:4, ' Retry=', Retry:4);
  Writeln('Max= ', Sum1/Retry:7:1,  ' ms');
  Writeln('Opt= ', Sum2/2/Retry:7:1,' ms');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin
  repeat
    Write('Nodes= '); Readln(Nodes);
    if Abs(Nodes)<3 then Break;
    Write('Retry= '); Readln(Retry);
    if Retry=0 then Break;
    Sum1:= 0; Sum2:= 0;
    for i:= 1 to Retry do begin
      // Полный граф:
      // Gr:= TGraphChars.GenFull((Nodes>0), 0, 99, Abs(Nodes));
      // Двудольный насыщенный граф:
      // Gr:= THugeGraph.GenDicoty(99, Nodes);
      // Неориентированный граф 20% плотности:
      Gr:= TGraphChars.GenRandom(false, 0, 99, Nodes, -20{%});
      Write('.');
      // Без учёта веса дуг:
      Start:= Now;
      Res:= Gr.GenPairs(pMaxN);
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum1, Time);
      Res.Free;
      // Минимального веса
      Start:= Now;
      Res:= Gr.GenPairs(pMinW);
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum2, Time);
      Res.Free;
      // Максимального веса
      Start:= Now;
      Res:= Gr.GenPairs(pMaxW);
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum2, Time);
      Res.Free;
      Gr.Free;
    end;
    Writeln(#7);   ResultExpo('');
    ResultExpo('Out_17d.txt');
  until false;
end.
```

Отметим, что время поиска оптимального паросочетания берётся как среднее от времени поиска минимального и максимального паросочетаний. В следующих ниже таблицах показаны результаты испытаний на 50 случайных графах разной размерности.

Табл. 28-3 — Среднее время поиска паросочетаний в полных графах

Количество вершин	Среднее время поиска, ms	
	Без учёта веса вершин	Оптимальное паросочетание
9	0,6	9,4
10	0,6	11,6
11	0,6	145,0
12	0,6	156,0
13	0,6	2641,0
14	0,6	2703,0

Табл. 28-4 — Среднее время поиска оптимальных паросочетаний в случайных графах с плотностью связей 20%

Количество вершин	Среднее время поиска, ms	
	В орграфе	В неориентированном графе
20	18,5	61,8
21	41,6	135,5
22	61,4	294,0
23	175,3	632,0
24	324,8	1117,0
25	771,2	3157,0

Результаты для разреженных графов показаны на следующем рисунке.

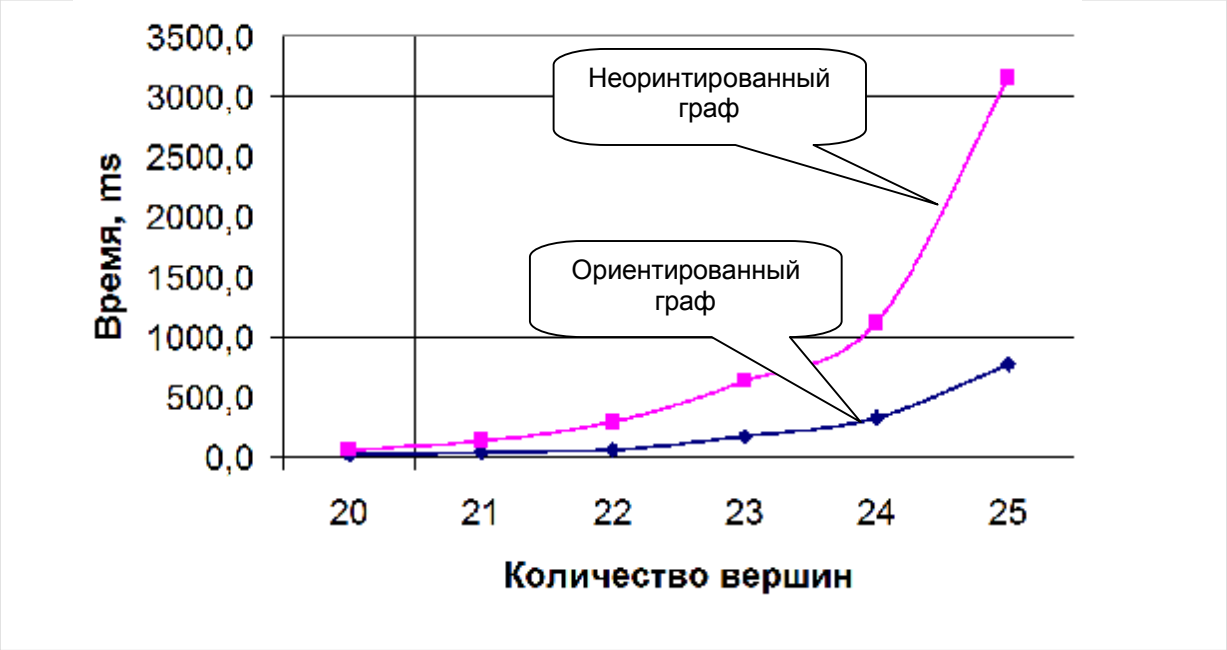


Рис. 28-6 — Среднее время поиска оптимальных паросочетаний в разреженных графах (плотность связей 20%)

Отсюда можно полагать, что граница применимости алгоритма будет тем шире, чем сильнее разрежен граф. В некоторых случаях расширить применимость перебора можно, пожертвовав точностью решения. Поскольку пары перебираются в порядке не убывания стоимости, алгоритму присуща некая похвальная жадность: даже первые промежуточные решения могут быть весьма близки к идеальному.

28.7. Итоги

- Формирование пар из равноправных однородных объектов соответствует поиску паросочетаний в произвольном графе.
- При формировании паросочетаний максимальной мощности могут преследоваться следующие цели: получить *любое* паросочетание; получить паросочетание *наименьшей* или *наибольшей* стоимости (совершенные). Все три типа задач можно свести к поиску паросочетания минимальной стоимости (веса).
- Решение задачи поиска паросочетаний (и **M**-сочетаний) можно свести к подбору непересекающихся подмножеств.
- Для снижения трудоёмкости перебора подмножеств используются следующие приёмы: формирование минимального количества блоков пар, сортировка подмножеств и блоков, перебор блоков с досрочным возвратом по результатам предсказания.

28.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 405
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 178
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 29
Паросочетание. Алгоритм Эдмондса

В предыдущей главе был реализован экспоненциально сложный алгоритм поиска совершенных паросочетаний, основанный на разбиении множества. Теперь рассмотрим гораздо более быстрый алгоритм Эдмондса, применяющий метод *чередующихся* (*альтернирующих*) цепей.

29.1. Чередующиеся, аугментальные и венгерские цепи

Рассмотрим поиск в связном графе паросочетания максимальной мощности без учёта веса рёбер. Вершины, соединённые в пару, называют *покрытыми* перосочетанием, на рисунках они будут окрашены чёрным цветом. Остальные *непокрытые* вершины будут окрашены белым. Изначально окрасим белым все вершины графа. Далее возьмём любую вершину, исследуем смежные ей, и образуем пару с первой из обнаруженных белых вершин, после чего окрасим обе вершины и смежное им ребро в чёрный цвет (парное чёрное ребро выделяется жирной линией). Затем возьмём следующую белую вершину и поступим так же. В какой-то момент может оказаться, что у очередной белой вершины не найдётся белых соседей, как это показано на Рис. 29-1 b.

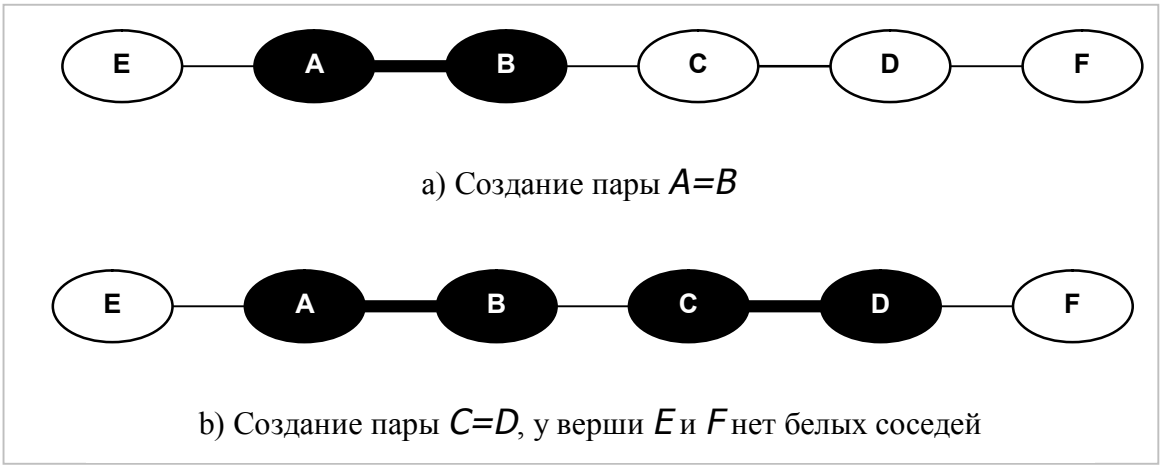


Рис. 29-1 — Неполное паросочетание

Хотя очевидно, что в данном случае может быть образовано паросочетание из трёх рёбер, как показано на Рис. 29-2.

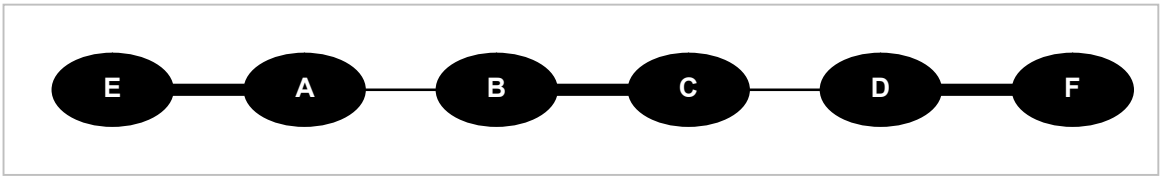


Рис. 29-2 — Полное паросочетание

Условимся далее парные рёбра обозначать символом «=», а непарные — символом «-». В данном примере полное паросочетание может быть получено инвертированием цепи *E-A=B-C=D-F* между крайними белыми вершинами *E* и *F*.

После такой инверсии парные жирные рёбра станут тонкими, а непарные тонкие — жирными. Теперь в паросочетании станет на одно ребро больше. Цепь, состоящую из нечётного количества чередующихся рёбер с белыми вершинами на концах, называют *увеличивающей* или *аугментальной*.

На Рис. 29-3 показан случай, когда чередующаяся цепь между белой вершиной *E* и чёрной вершиной *D* имеет чётное количество рёбер. Инверсия такой цепи не даст приращения паросочетания, такую цепь называют *венгерской*.

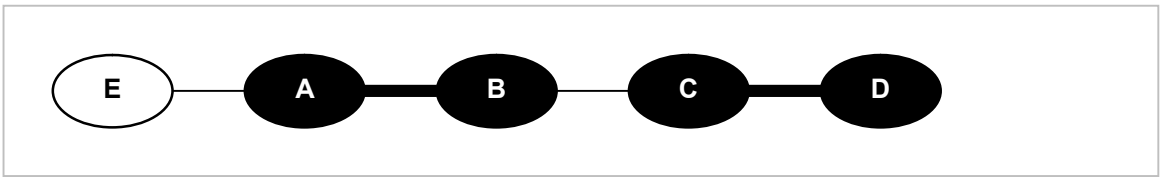


Рис. 29-3 — Венгерская цепь

Забегая вперёд, отметим, что при поиске паросочетаний с учётом веса рёбер иногда инвертируют и венгерские цепи, как это показано на Рис. 29-4. Такую венгерскую цепь назовём *слабой увеличивающей* цепью.

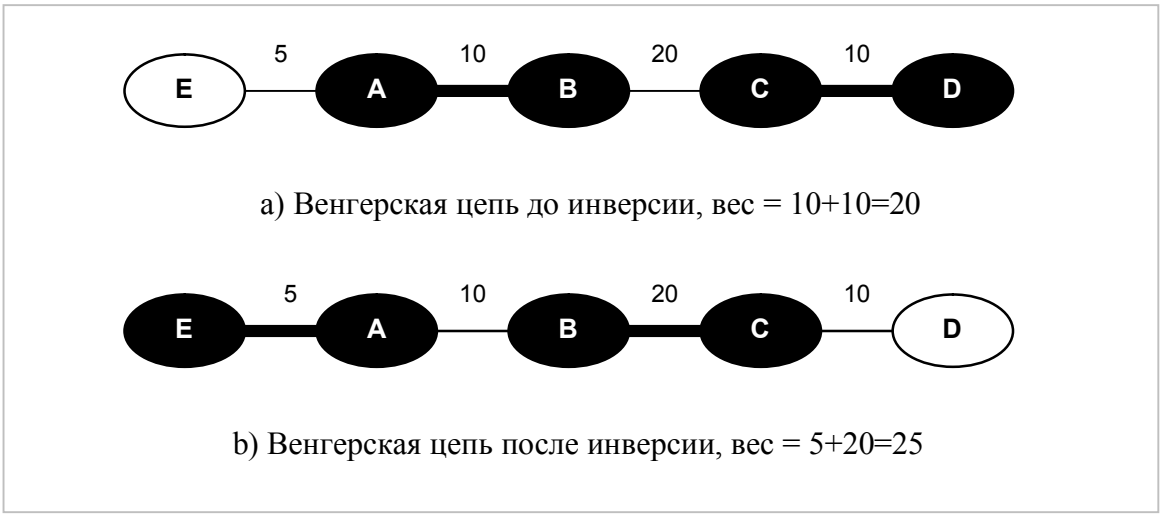


Рис. 29-4 — Инвертирование слабой увеличивающей (венгерской) цепи

29.2. Построение аугментальных цепей

Из сказанного выше следует, что для подбора пары к очередной непокрытой белой вершине необходимо построить соответствующую увеличивающую цепь (сильную либо слабую). Здесь вполне годится «взращивание» дерева с корнем в этой белой вершине обходом графа в ширину, подобно тому, как это делается при поиске кратчайших путей. С той лишь разницей, что для чередования парных и непарных рёбер, в момент присоединения к дереву одной из вершин, в очередь на расширение дерева ставится не эта, а парная ей вершина. Первую из названных покрытых вершин назовём *внутренней*, а вторую — *внешней*. Внутренняя вершина пары находится на одно ребро ближе к корню, чем внешняя.

Обратите внимание: статус *внутренняя* или *внешняя* относится только к позиции вершины по отношению к корню данного дерева. При построении дерева из другого корня статус вершины в паре может измениться.

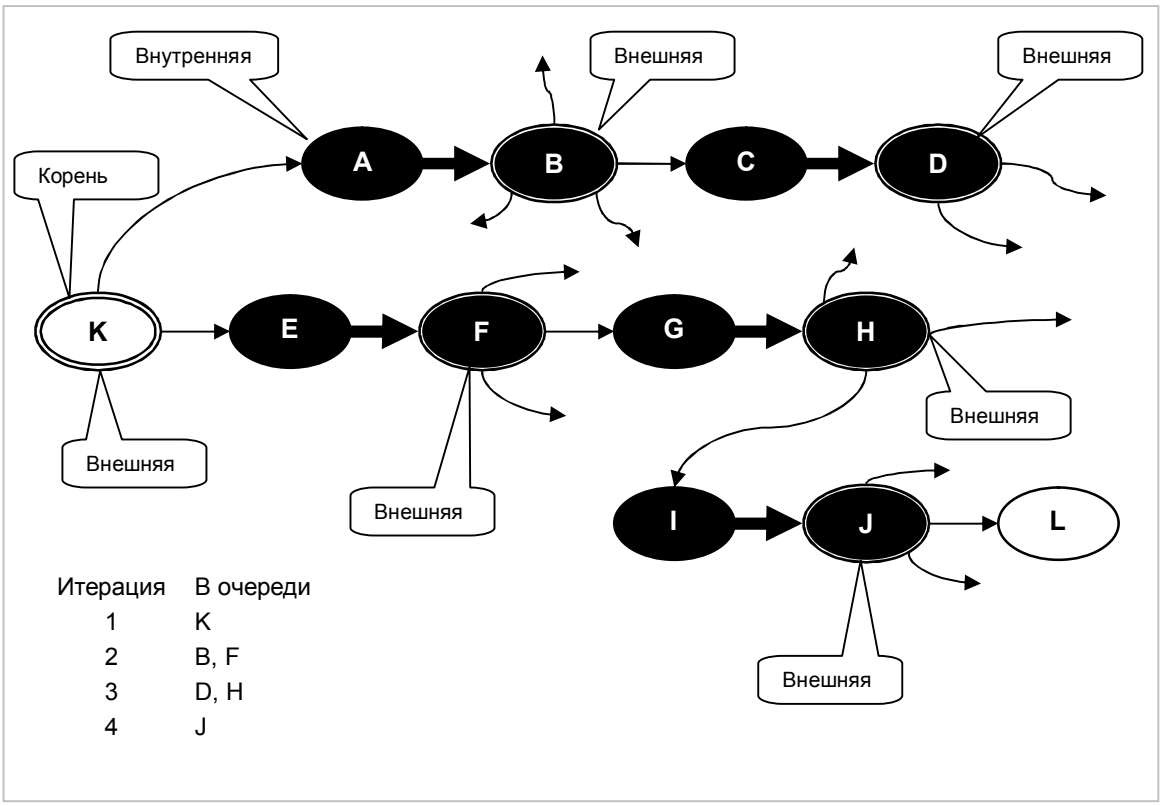


Рис. 29-5 — Построение дерева с корнем в вершине K. Внешние вершины ограничены двойной линией

На Рис. 29-5 показано построение дерева с корнем в вершине *K*. Корень является внешней вершиной, и потому на первой итерации сразу попадает в очередь расширения. Из него доступны покрытые вершины *A* и *E*, однако эти вершины внутренние по отношению к корню, и потому в очередь на расширение ставятся только парные им внешние вершины *B* и *F*. Расширение продолжается вплоть до обнаружения белой вершины *L*. Отсюда по обратным ссылкам можно вернуться к корню дерева, попутно инвертируя парные и непарные рёбра, тем самым добавляя к текущему паросочетанию ещё одну пару.

В общем случае построение дерева с корнем в некоторой вершине может завершиться:

a) Обнаружением сильной увеличивающей цепи (аугментальной).
b) Обнаружением слабой увеличивающей цепи (венгерской).
c) Безуспешно после исчерпания очереди на расширение дерева.

В первых двух случаях инверсией цепи в направлении корня можно увеличить мощность паросочетания на единицу или увеличить его стоимость. В третьем случае — после безуспешного расширения — корневая вершина остаётся непокрытой, белой.

29.3. Проблема нечётных циклов

Представленный выше поиск увеличивающих цепей корректен в отсутствие в графе циклов нечётной длины. При наличии таких циклов возникает неопределённость, связанная с порядком обработки смежных вершин. Обратимся к Рис. 29-6, где имеется цикл нечётной длины *E-A=B*. Если в ходе расширения дерева будет выбрана сначала внутренняя вершина *A*, то, в конечном счете, поиск «упрётся» в чёрную вершину *D*. Искомая аугментальная цепь *E-B=A-F* будет найдена, если построение дерева пойдёт в направлении внутренней вершины *B*.

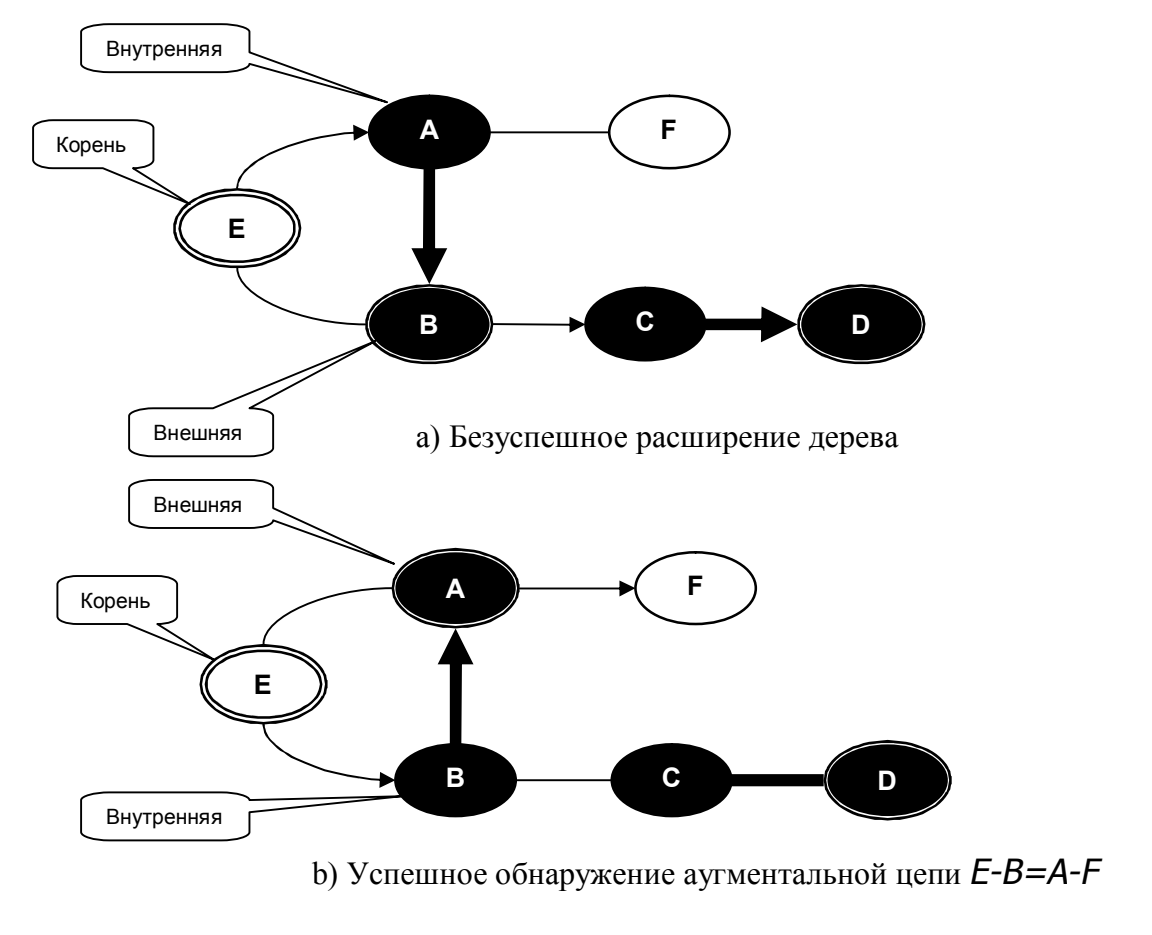
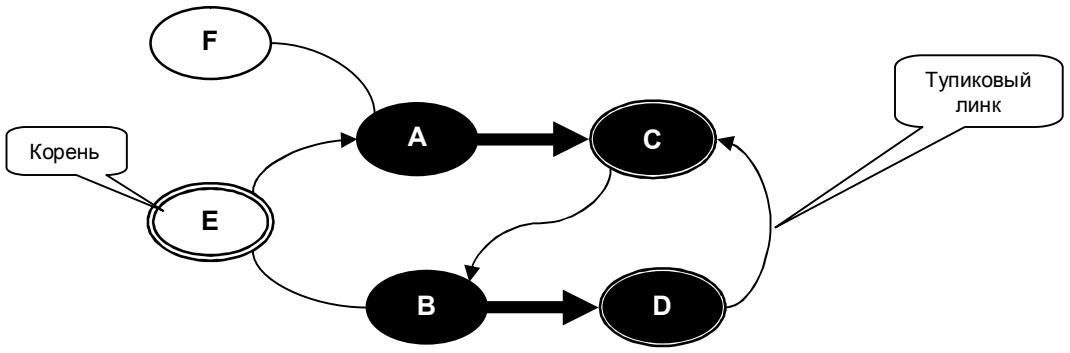
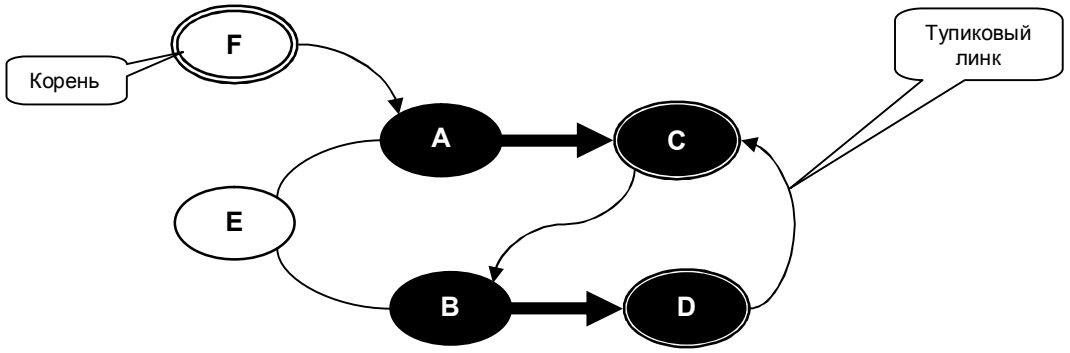


Рис. 29-6 — Варианты продвижения по циклу нечётной длины

Порядок обработки вершин в графе фиксирован неким соглашением, например, по алфавиту, и потому угадать подходящую очередную вершину в каждом таком случае невозможно. Вот ещё пример, где перебор вершин в алфавитном порядке заводит поиск в тупик (Рис. 29-7). Здесь при выборе в качестве корня любой из двух вершин расширение дерева прекратится с образованием венгерских цепей. Успешный поиск цепи был бы возможен, если бы расширение дерева пошло в направлении *E-B=D-C=A-F*. Однако такой порядок противоречил бы принятому соглашению обхода по алфавиту.



a) Безуспешное расширение дерева из корня *E*



b) Безуспешное расширение дерева из корня *F*

Рис. 29-7 — Тупиковые расширения дерева в графе с циклом нечётной длины

29.4. Цветки

Джек Эдмондс (Jack Edmonds) для решения этой проблемы предложил заменять циклы нечётной длины специальными вершинами — *цветками*. Рассмотрим Рис. 29-8, где изображён граф с циклом нечётной длины. В соответствии с принятым порядком перебора здесь доступны к просмотру только рёбра, исходящие из внешних вершин (обозначены сплошными линиями). Хотя к решению ведёт просмотр рёбер, исходящих из внутренней вершины *A*. Если свернуть все пять вершин нечётного цикла в одну, и рассматривать далее рёбра, исходящие из *всех* вершин цветка, то нужное решение находится.

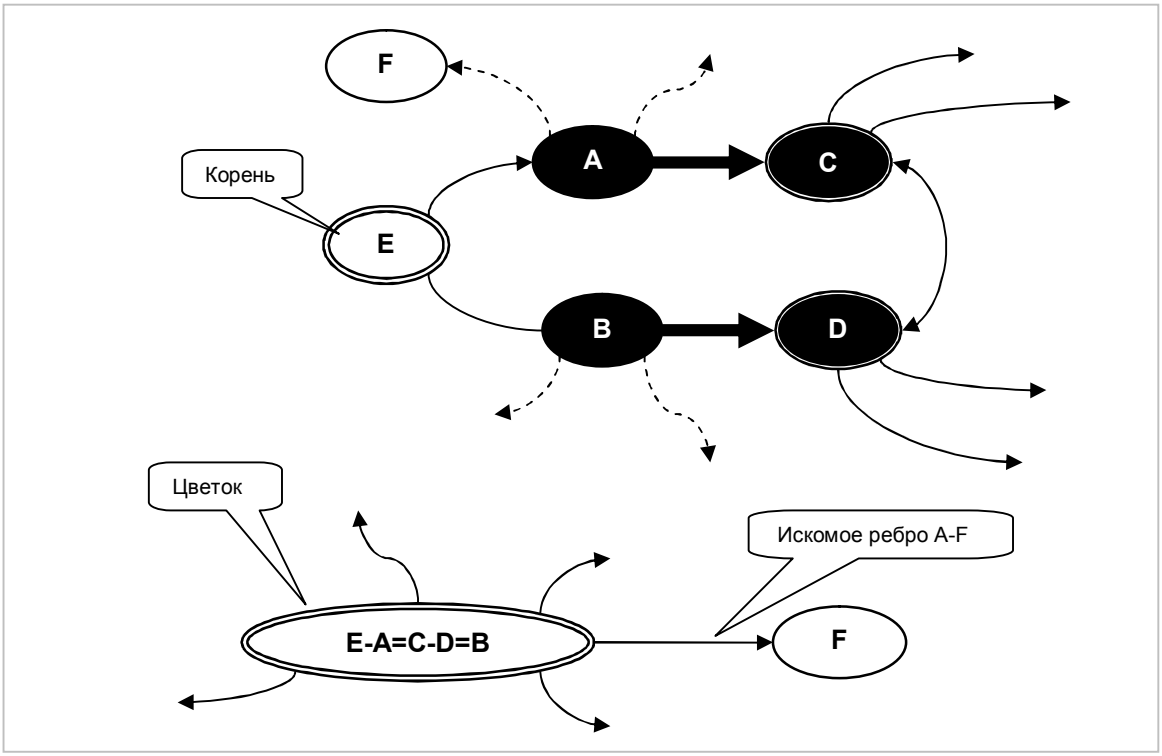


Рис. 29-8 — Граф с нечётным циклом и соответствующий цветок

Прежде, чем продемонстрировать применении цветков на одном из «проблемных» графов, обсудим структуру цветка и технику его использования, для чего обратимся к Рис. 29-9.

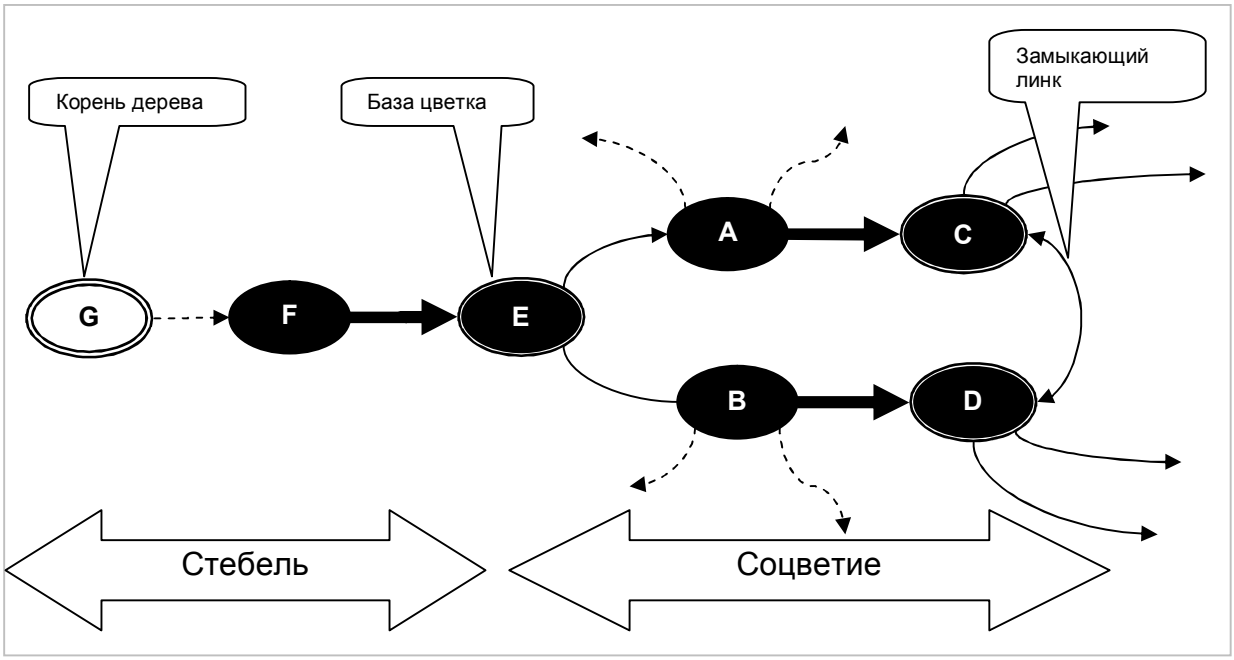


Рис. 29-9 — Структура цветка: стебель, соцветие и база

Здесь вершины, составляющие цикл нечётной длины *E-A=C-D=B*, образуют *соцветие*. Чётное количество вершин соцветия образует пары, и только одна вершина, называемая *базой* цветка, либо имеет пару за пределом соцветия, либо вовсе не имеет пары, то есть, является непокрытой, белой (см. Рис. 29-8).

Как обнаруживаются нечётные циклы и создаются цветки? Пусть в момент исследования внешней вершины дерева обнаружится линк на другую внешнюю вершину того же дерева. В нашем случае это может быть либо вершина *C*, либо вершинная *D*. Такой линк назовём *замыкающим*. Из этого следует, что обнаружен нечётный цикл, и расширение дерева следует продолжить из всех вершин этого цикла. То есть, нарушив правило чередования рёбер, поставить в очередь расширения наряду с внешними, ещё и внутренние вершины *A* и *B*. Для обхода правила чередования рёбер можно было бы преобразовать граф, стянув нечётный цикл в одну единственную вершину и связав эту вершину необходимыми рёбрами с остальным графом, — дело весьма затратное, особенно с учётом обратного преобразования в последующем. На практике быстрее и проще снабдить вершины нечётного цикла атрибутами, определяющими их принадлежность к некоторому цветку. Рассмотрим процесс создания такого цветка.

Обнаружив замыкающий линк, возьмём одну из смежных ему вершин, в данном случае *C* или *D*, и по цепочке обратных связей пройдём к корню дерева, помечая каким-либо образом вершины вдоль этой цепи. Затем по цепочке обратных связей пройдём из другой вершины замыкающего линка в поисках первой из помеченных вершин, — она и будет *базой* цветка, в данном примере это вершина *E*. Итак, база цветка — один из его атрибутов — найдена.

Далее надо организовать циклические связи в цветке с тем, чтобы иметь возможность обойти составляющие его вершины, как по часовой стрелке, так и против. Разумеется, что эти направления условны. Для организации циклических ссылок вновь проходим дважды от концов замыкающего линка к базе цветка.

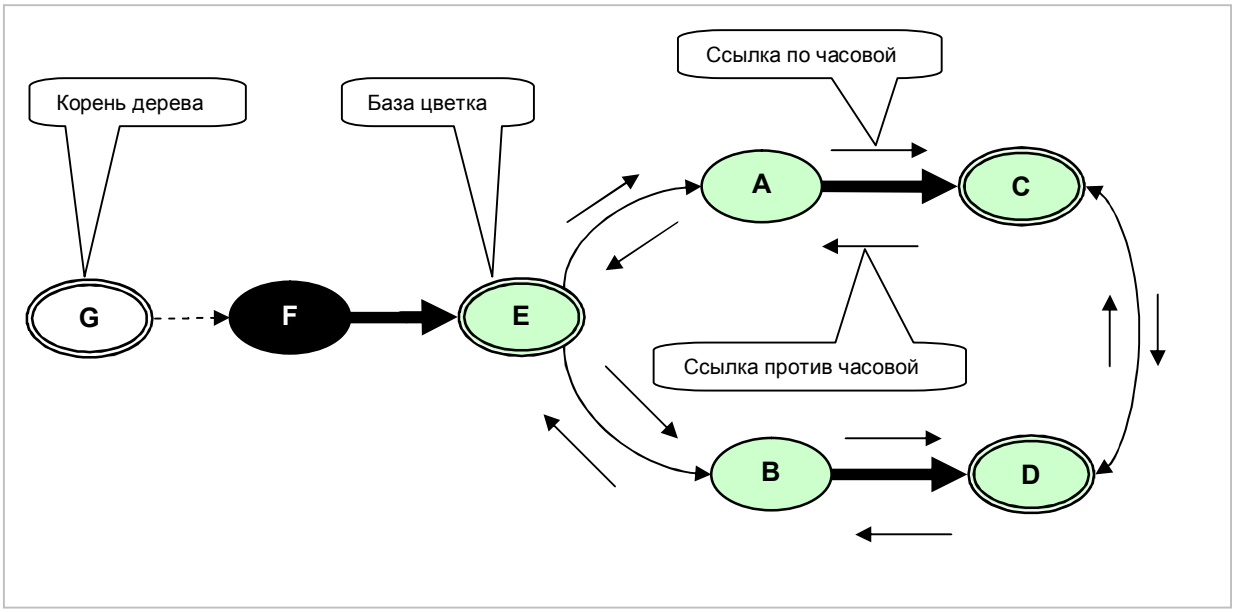


Рис. 29-10 — База цветка и циклические ссылки

В заключение все вершины цветка необходимо снабдить некой меткой принадлежности к созданному цветку. Такой меткой будет ссылка на особый объект **`TBlossom`** — потомок объекта «вершина» **`TNode`**. Этот объект, наряду с полями и методами вершины, будет содержать ряд дополнительных полей и

методов, например поле, содержащее базу цветка. Цветок хранится вне графа и не меняет его структуру. Создав объект-цветок, снабдим ссылками на него все вершины нечётного цикла (вершины соцветия). Для вершин, не состоящих в цветке, ссылка на объект-цветок изначально содержит саму это вершину. После создания цветка в очередь расширения дерева ставятся все его внутренние вершины.

Теперь рассмотрим применение цветков на примере ранее приведенного «проблемного» графа, представленного на Рис. 29-7. При расширении дерева с корнем в вершине *E*, линк из внешней вершины *D* приводит в другую внешнюю вершину того же дерева — вершину *C*. Это говорит о наличии цикла нечётной длины, и потому нечётный цикл *C-B=D* сжимаем в цветок *001*. Далее для отличия цветков от вершин графа будем нумеровать цветки числами. Базой цветка *001* является вершина *C*. После сжатия цикла, ставится в очередь вершина *B*, изменившая свой статус с внутренней на внешнюю, поскольку цветок *001* в целом также становится внешней вершиной (Рис. 29-11).

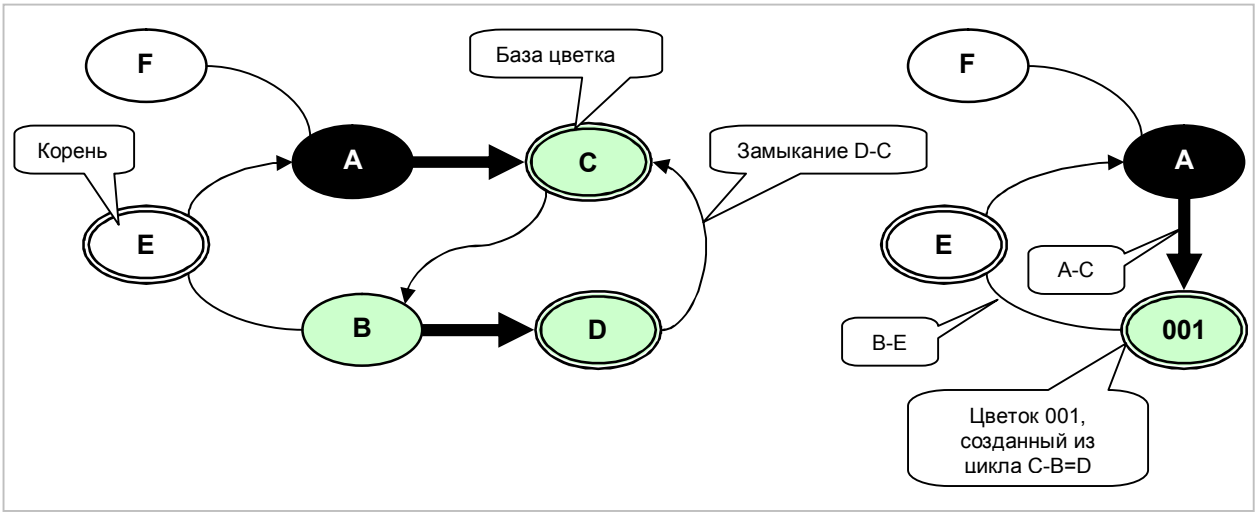


Рис. 29-11 — Нечётный цикл C-B=D (слева) и эквивалентный ему граф с цветком (справа)

На следующей стадии расширения дерева линк из внешней вершины *B* приводит к внешней вершине *E* того же дерева (и по совместительству корню). Тем самым обнаруживается нечётный цикл *E-A=001*, который необходимо сжать в цветок *002* с базой в вершине *E*. Бывшая ранее внутренней вершина *A* обретает статус внешней (как и весь цветок) и ставится в очередь расширения дерева. В конце концов, линк *A-F* приводит к непокрытой вершине *F*, замыкающей аугментальную цепь. После инверсии этой цепи (Рис. 29-12) образуется новое паросочетание *F=002*.

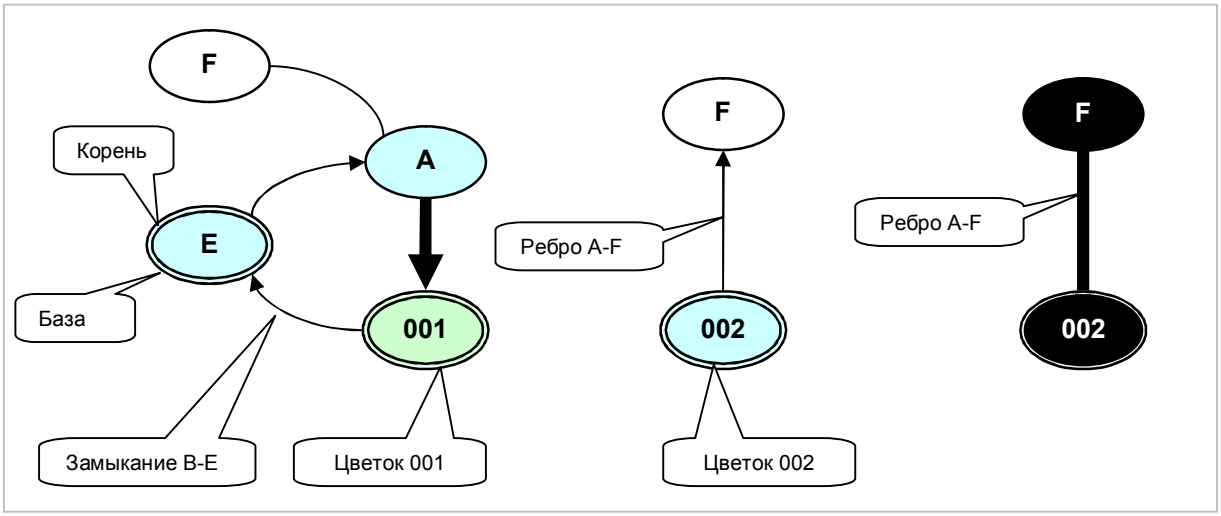


Рис. 29-12 — Нечётный цикл E-A=001 (слева), эквивалентный ему граф с цветком (в центре) и образованная в итоге пара (справа)

После обработки всех вершин графа и создания максимально возможного количества пар следует завершающая стадия алгоритма — роспуск ранее созданных цветков. Читатель, вероятно, заметил, что некоторые вершины внутри цветка оказались состоящими как бы в двух парах сразу, т.е. имеют парой и вершину внутри цветка, и вершину вне оного, как это показано на Рис. 29-13 для вершины *A*.

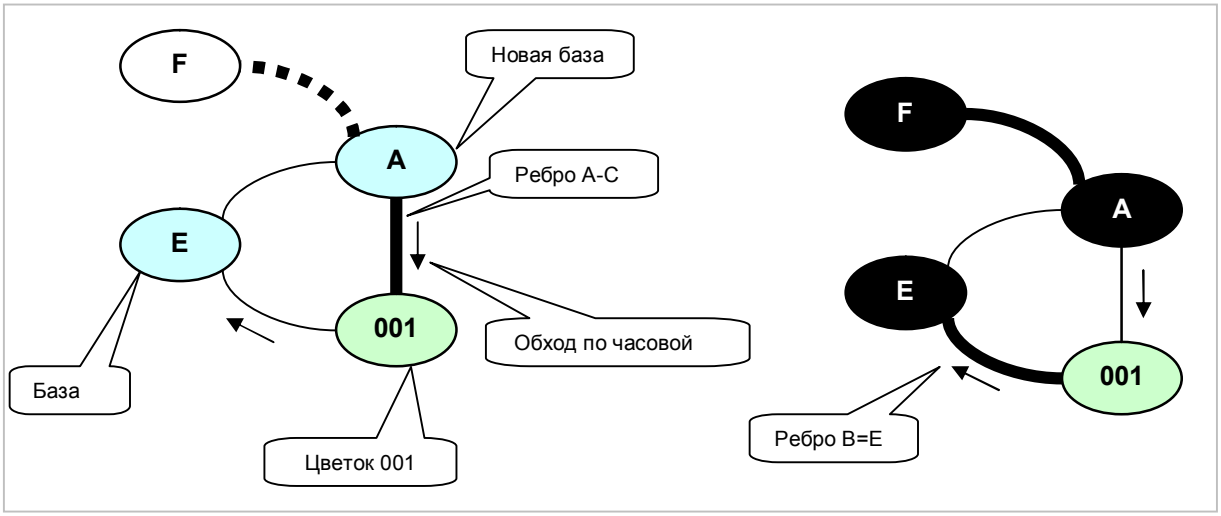


Рис. 29-13 — Роспуск цветка *002* (слева) и результат (справа)

Очевидно, что в «правильном» цветке такая «странная» вершина должна стать его *новой* базой. Так и назовём её — *новая* база цветка (в отличие от просто базы или *первичной* базы). Для приведения цветка в порядок, необходимо «отцепить» новую базу цветка *A* от её внутренней пары *001* и соединить с вершиной парой вне цветка — вершиной *F*. Для этого пройдём по цепи от новой базы к первичной базе, попутно инвертируя парные и непарные рёбра. Этот обход выполняется либо по часовой, либо против часовой стрелке, — направление задаёт внутренняя пара. Затем соединим в пару новую базу цветка *A* с внешней вершиной *F*. В итоге получаем промежуточный результат, показанный на Рис. 29-13 справа.

Далее следует роспуск цветка *001* (Рис. 29-14). Здесь проблемной оказывается вершина *B*, её следует сделать новой базой и выполнить действия, описанные в предыдущем абзаце.

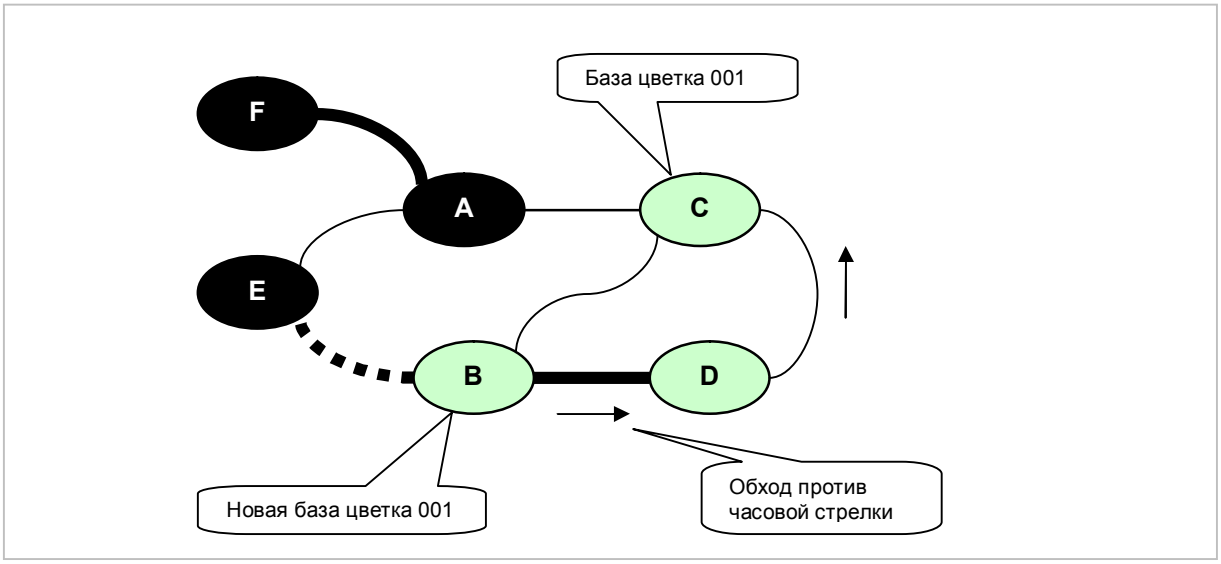


Рис. 29-14 — Роспуск цветка *001*, обход против часовой стрелки

В конце концов, после роспуска всех цветков в порядке, обратном их созданию, получим искомое паросочетание максимальной мощности (Рис. 29-15).

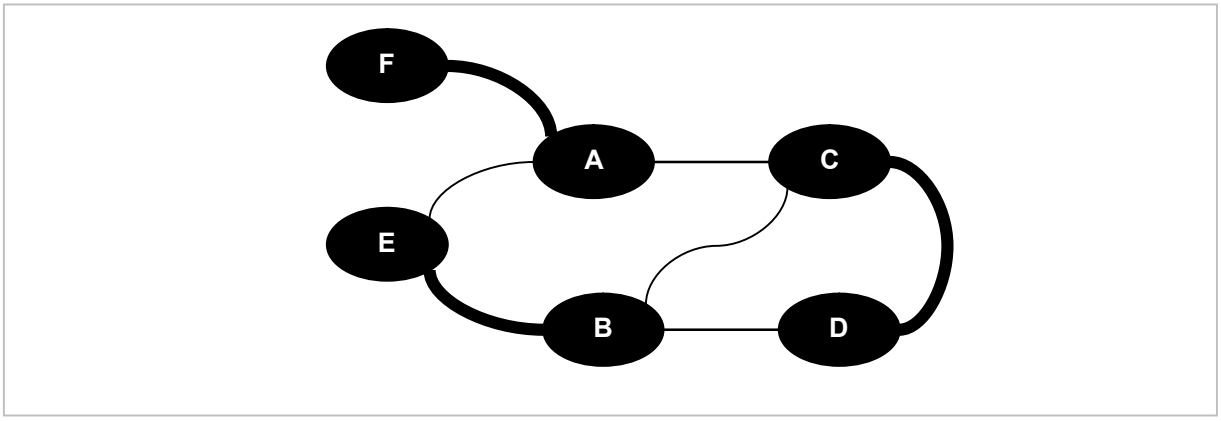


Рис. 29-15 — Итог роспуска всех цветков

Подведём промежуточные итоги:

- Для расширения текущего паросочетания необходимо найти в графе чередующуюся цепь (аугментальную), ограниченную непокрытыми вершинами, и содержащую нечётное количество рёбер (по меньшей мере, одно ребро).
- Инверсия рёбер вдоль такой аугментальной цепи увеличивает мощность текущего паросочетания на единицу.
- Для поиска аугментальных цепей строится дерево с корнем в непокрытой вершине.

- Наличие в графе циклов нечётной длины влечёт неопределённость результата поиска, поскольку результат будет зависеть от выбранного порядка обследования вершин.
- Устранение неопределённости достигается сворачиванием нечётных циклов в псевдо-вершины, именуемые цветками.
- Раскрытие цветков выполняется в порядке, обратном их созданию. В ходе раскрытия цветка инвертируется та или иная внутренняя цепочка рёбер, соединяющая первичную базу цветка с новой базой. При этом выполняется обход рёбер по часовой, либо против часовой стрелки.

29.5. Взвешенные графы

Итак, решена задача безотносительно весов рёбер. При таком подходе, в зависимости от порядка обработки вершин, можно получить несколько разных паросочетаний одинаковой мощности. В графе с взвешенными рёбрами эти паросочетания будут иметь разный вес. По меньшей мере, одно из них будет обладать максимальным весом, и одно — минимальным. Такие экстремальные паросочетания называют *оптимальными*, соответственно максимального, либо минимального веса. Не исключено, что оптимальное паросочетание окажется единственным, то есть и максимального, и минимального веса одновременно.

Далее рассмотрим изменения в алгоритме, необходимые для поиска *оптимального* паросочетания *максимального* веса. Поиск *оптимального* паросочетания *минимального* веса сводится к тому же путём предварительной инверсии знака веса рёбер.

Основная идея состоит в следующем. Назначим всем вершинам графа некоторый произвольный вес с тем условием, чтобы сумма весов смежных вершин любого ребра была не меньше веса этого ребра, то есть:

$$Vx + Vy \geq Lxy$$

где **Vx** и **Vy** — веса вершин *X* и *Y*, **Lxy** — вес соединяющего их ребра *X-Y*.

Далее будем считать ребро L доступным для построения дерева поиска, если сумма весов смежных ему вершин в точности совпадает с весом ребра:

$$Vx + Vy = Lxy$$

Такое доступное ребро назовём *уравновешенным*, поскольку здесь напрашивается аналогия с гиревыми весами, где на одну чашу положено ребро, а на другую — пара гирь (вершин). Без потери общности и для наглядности предположим, что всем вершинам первоначально назначены достаточно большие веса так, что все рёбра оказываются неуравновешенными, т.е. для всех рёбер выполняется неравенство:

$$Vx + Vy > Lxy$$

Иначе говоря, для алгоритма поиска цепей все рёбра графа станут недоступными, поскольку они не уравновешены, и граф, условно говоря, окажется совокупностью изолированных вершин. Далее положим, что, манипулируя весами вершин и *уменьшая* их общий вес, удалось уравновесить некоторую часть рёбер и найти нужное паросочетание максимального веса. Очевидно, что сумма весов составляющих его вершин окажется равной сумме весов рёбер этого паросочетания (иначе они не попали бы в аугментальную цепь). Очевидно, также, что в ходе решения для достижения равновесия нужных рёбер необходимо будет уменьшать начальные веса вершин в минимальной степени. Ведь чем *меньшими* будут эти уменьшения весов, тем *большим* окажется оставшийся совокупный вес вершин паросочетания.

Рассмотрим следующий пример (Рис. 29-16). Здесь всем вершинам заданы заведомо большие веса, так, что все рёбра, смежные корню, не уравновешены и показаны пунктиром. Добиться равновесия хотя бы одного из рёбер возможно уменьшением веса корневой вершины на величину Δ, которая должна быть по возможности минимальной. Значение Δ для любой пары вершин *X* и *Y* вычисляется из условия равновесия:

$$\Delta = Vx + Vy - Lxy$$

Среди всех таких вершин наименьшее значение Δ=1 даёт ребро *A-C*, и после уменьшения веса вершины *A* ребро *A-C* уравновешивается, присоединяется к дереву поиска, и в итоге пополняет паросочетание. Таким образом, присоединение к текущему дереву смежных рёбер происходит не в порядке нумерации вершин, а в порядке не убывания весов смежных им рёбер.

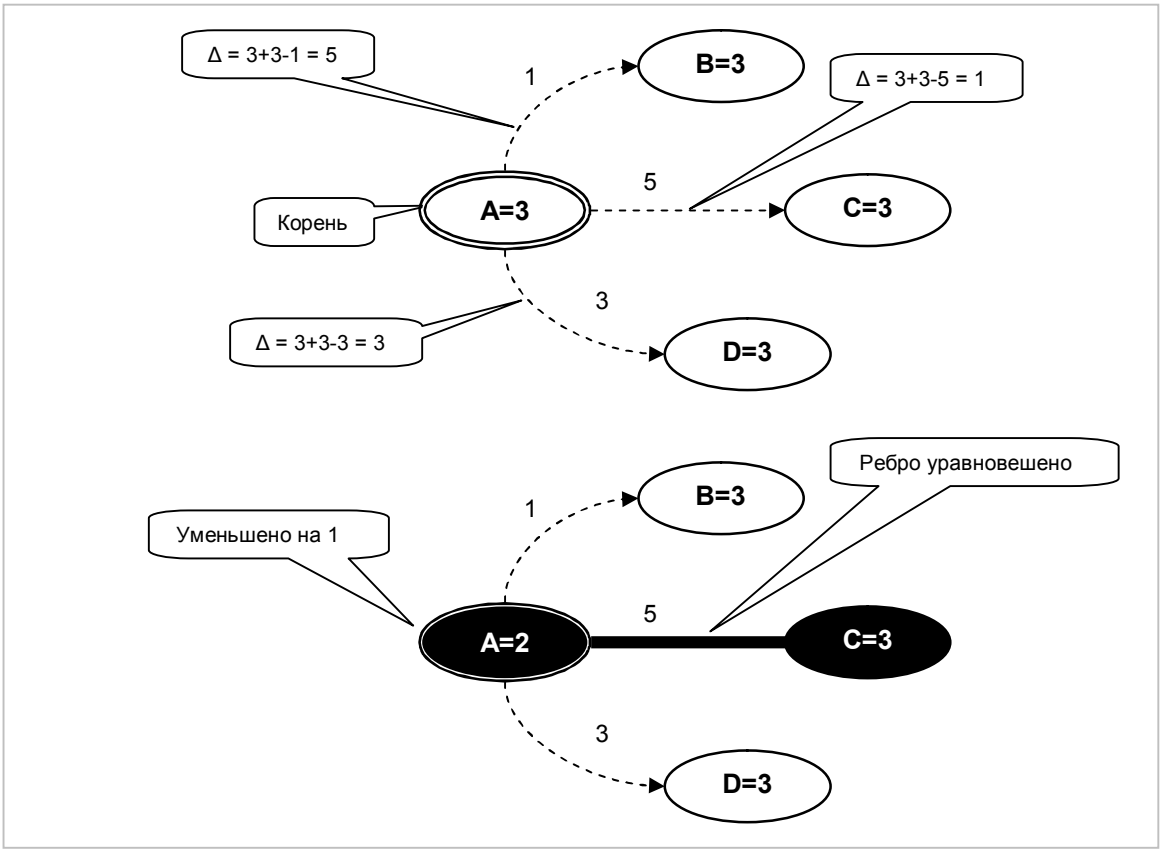


Рис. 29-16 — Выбор смежного ребра максимального веса

Рассмотрим пересчёт весов вершин в более длинных цепях (Рис. 29-17). Здесь, как и в предыдущем примере, для присоединения к дереву ребра *A-C* необходимо уменьшить вес внешней вершины *A*. Однако после этого нарушится равновесие парного ребра *E=A*, и потому на ту же величину необходимо увеличить вес вершины *E*, нарушив при этом равновесие ребра *F-E*. В итоге для сохранения равновесия всех рёбер цепи вплоть до корневой вершины необходимо *уменьшить* на величину Δ веса всех *внешних* вершин, и *увеличить* в той же степени веса *внутренних* вершин. А если вспомнить, что из корня дерева могут исходить несколько ветвей, то указанные операции следует выполнить со всеми вершинами данного дерева. Таким образом, после присоединения нового ребра суммарный вес всего текущего дерева уменьшится на Δ.

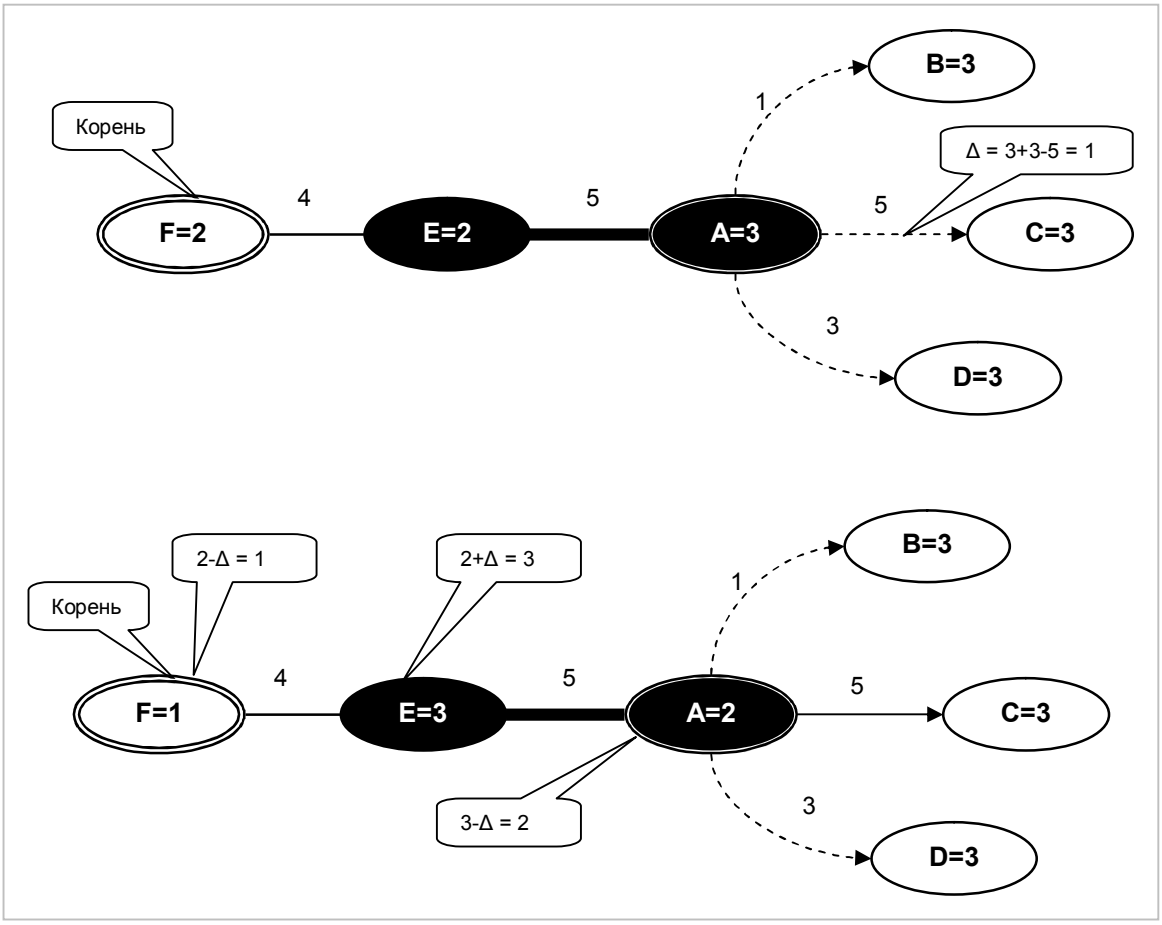


Рис. 29-17 — Пересчёт весов в цепи F-E=A-C

29.6. Случай нечётного цикла (цветка)

Рассмотрим корректировку весовых коэффициентов в случае нечётного цикла, свёрнутого в цветок. Пусть необходимо расширить дерево ребром *D-F*, как показано на Рис. 29-18. На этот момент все рёбра дерева сбалансированы, и теперь следует поменять их веса на величину Δ=1. В соответствии с нашим правилом веса внешних вершин уменьшим, а внутренних увеличим. Поскольку все вершины цветка *001* являются внешними, равновесие составляющих цветок рёбер окажется нарушенным. Но смежные цветку рёбра сохранят равновесие, а ведь именно эти рёбра важны для расширения дерева. Потому нарушением равновесия рёбер внутри цветка можно пренебречь, сохранив, однако, степень этого нарушения (компенсацию) внутри этого объекта, для чего использовать поле, хранящее вес цветка.

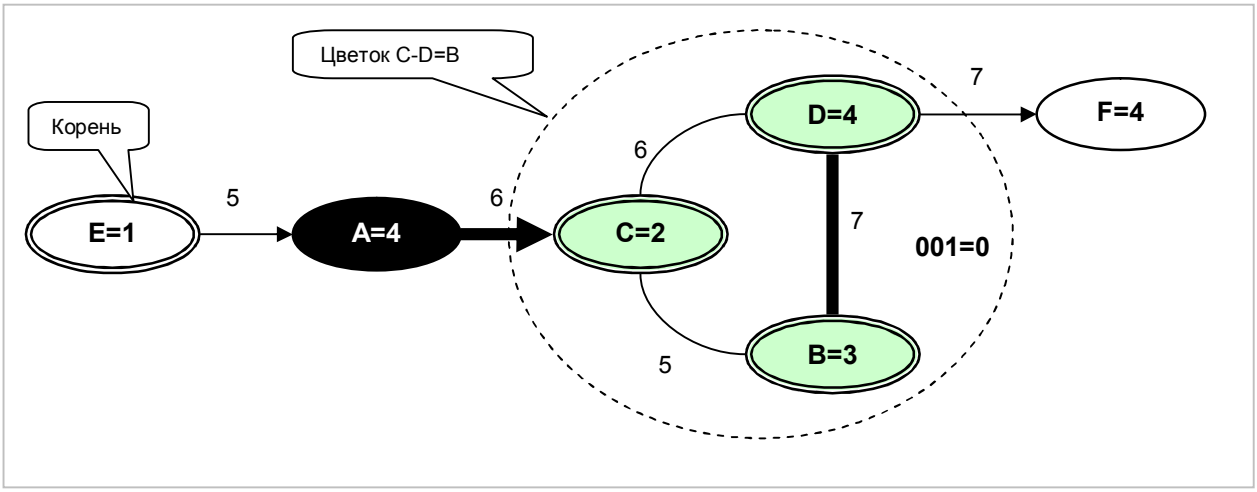


Рис. 29-18 — Весовые коэффициенты до корректировки

На Рис. 29-19 весовые коэффициенты уже скорректированы, при этом вес цветка *001* увеличен на 2•Δ — это компенсация потерянного равновесия пары внешних вершин внутри цветка.

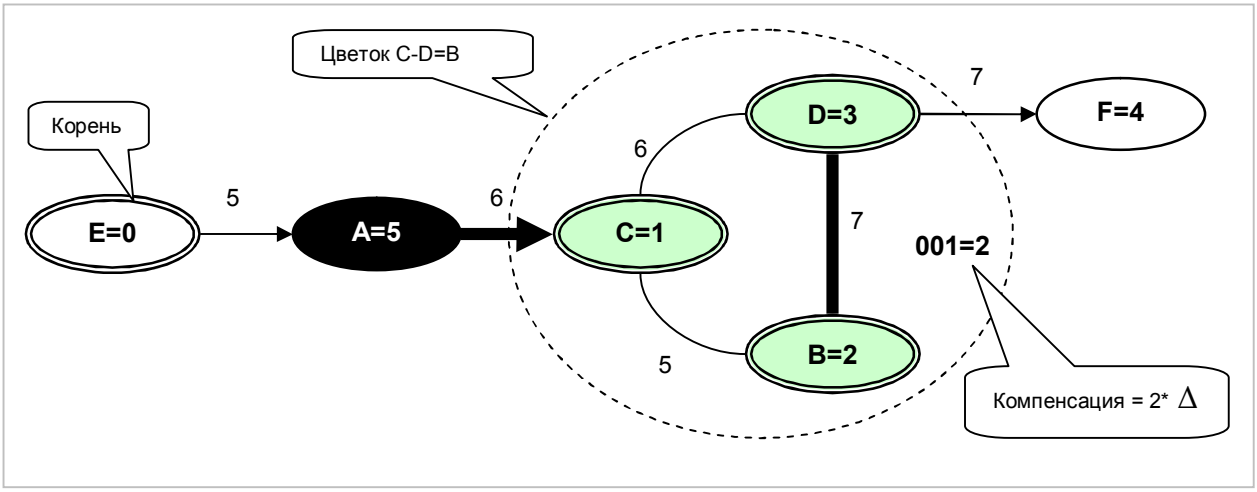


Рис. 29-19 — Весовые коэффициенты после корректировки, Δ=1

При роспуске цветка половина этой компенсации (Δ) будет добавлена к весу каждой вершины, за исключением новой базы цветка. Тем самым будет восстановлено равновесие всех парных вершин раскрытого цветка.

При создании очередного цветка сам цветок и включённые в него вершины получают статус внешних. Далее по ходу решения задачи цветок и входящие в него вершины в другой цепи могут оказаться *внутренними*. Тогда при корректировке весов к вершинам внутреннего цветка прибавляется значение Δ, а из веса цветка вычитается компенсация 2•Δ. Алгоритм построен так, что вес цветка (компенсация) всегда остаётся неотрицательной. Если после уменьшения веса цветка компенсация достигает нуля, то значит, составляющие цветок рёбра вновь сбалансированы. Такой цветок можно и *нужно* распустить, после чего некоторые его вершины вновь станут *внешними*. Если после этого перезапустить постройку дерева из корня, то оно примет иную конфигурацию. Часть вершин и рёбер бывшего цветка

окажется вне этого нового дерева, и впоследствии эти вершины могут вновь присоединиться к дереву, но уже в иной его конфигурации.

Рассмотрим следующий пример (Рис. 29-20). В начальном состоянии всем вершинам назначены веса так, что все рёбра, за исключением *A-B* оказываются неуравновешенными, закрытыми (изображены пунктиром).

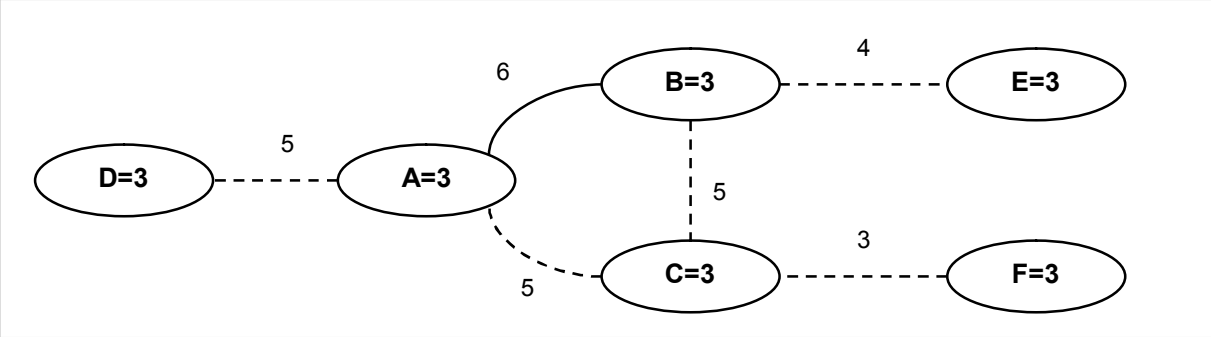


Рис. 29-20 — Начальное состояние вершинных чисел и рёбер

При построении дерева с корнем в *A* создаётся пара *A=B*, поскольку данное ребро уже сбалансировано (Рис. 29-21).

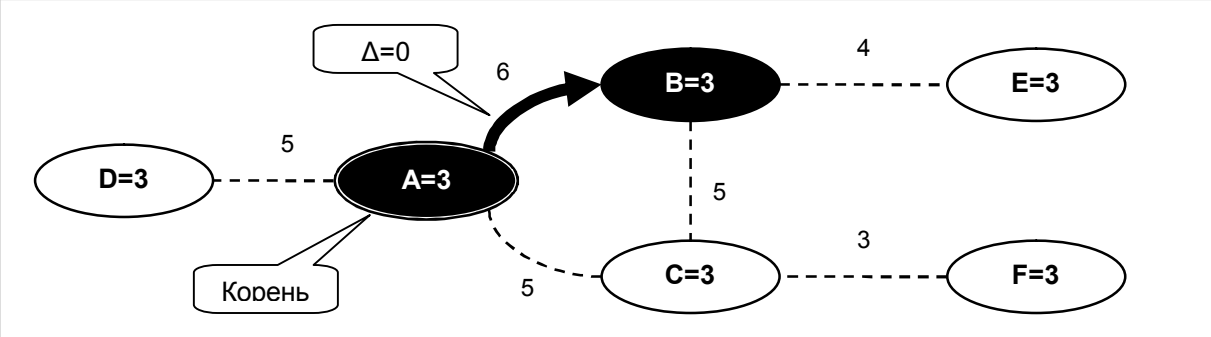


Рис. 29-21 — Образование первой пары A=B

Далее, при построении дерева с корнем в *C* минимальное значение Δ=1 дают рёбра *C-A* и *C-B*, а потому после корректировки веса вершины *C* открываются упомянутые рёбра с образованием нечётного цикла *C-A=B*. Этот цикл сжимается в цветок *001*, и далее дерево расширяется из этого внешнего цветка (Рис. 29-22).

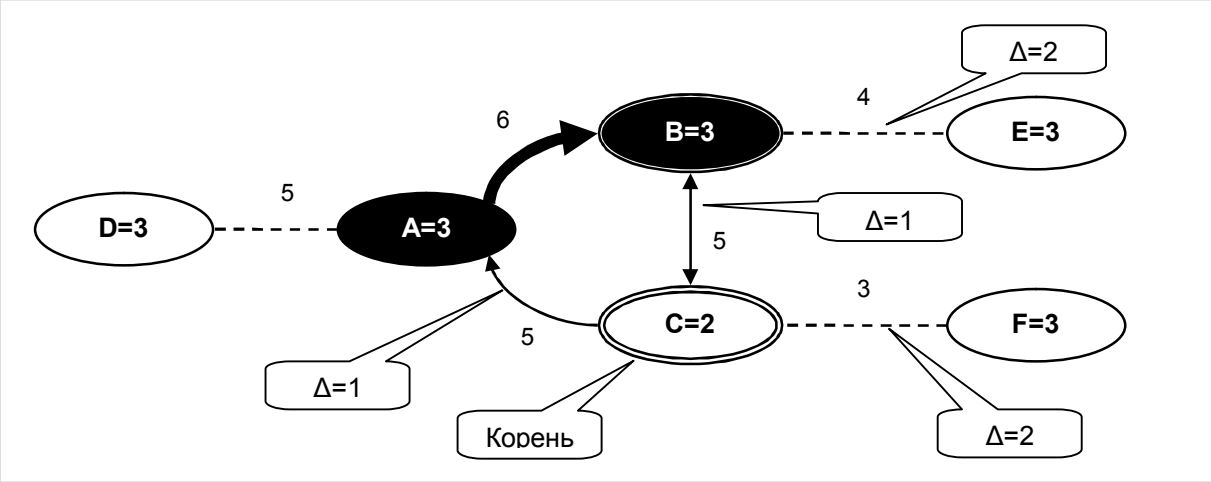


Рис. 29-22 — Построение дерева с корнем в C, образование цветка C-A=B

При обследовании вершин цветка выясняется, что наименьшее значение Δ=1 даёт ребро *A-D* (наиболее «тяжёлое» среди смежных цветку). Потому веса вершин цветка (все они внешние) уменьшаются на 1, а вес самого цветка (компенсатор) увеличивается на 2. После этого к дереву добавляется ребро *A-D*., что показано на Рис. 29-23.

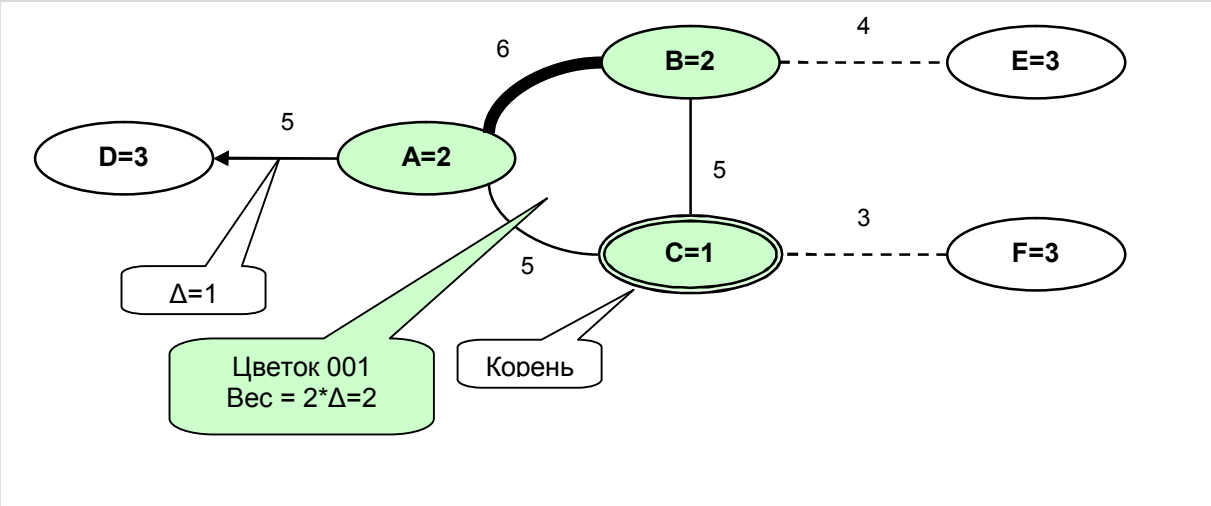


Рис. 29-23 — Корректировка весов и добавление ребра A-D

Поскольку вершина *D* не покрыта, расширение дерева прекращается (найдена аугментальная цепь). После реверса цепи создаётся пара *001=D* (Рис. 29-24).

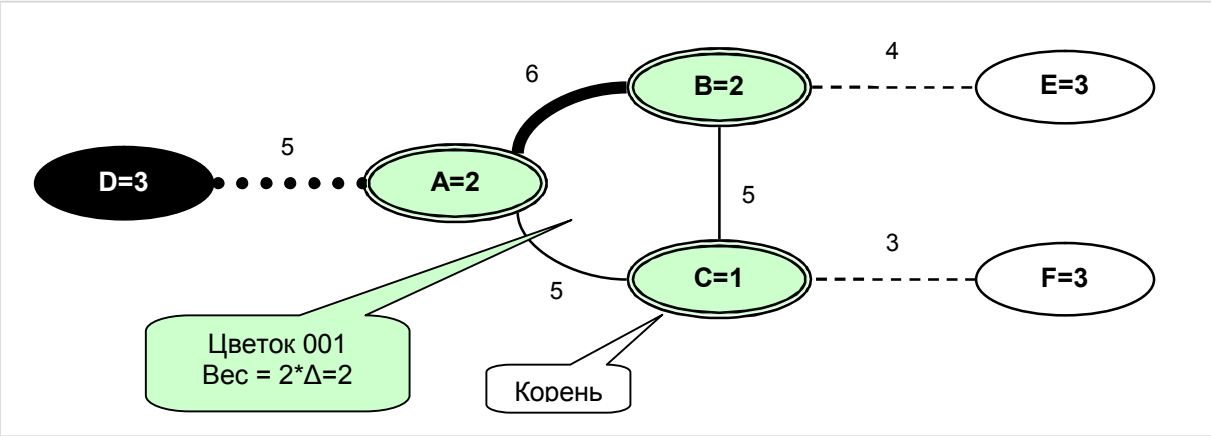


Рис. 29-24 — Создание пары 001=D

Следующее дерево строится из вершины *E* (Рис. 29-25). После необходимой корректировки веса этой вершины, открывается ребро *E-B*, ведущее в цветок *001* и далее в парную ему вершину *D*. С этого момента дальнейшее расширение дерева путём открытия новых рёбер становится невозможным, поскольку такое ребро (*C-F*) исходит только из цветка *001*, который теперь является внутренней вершиной по отношению к корню *E*, а внутренние вершины не могут расширять дерево.

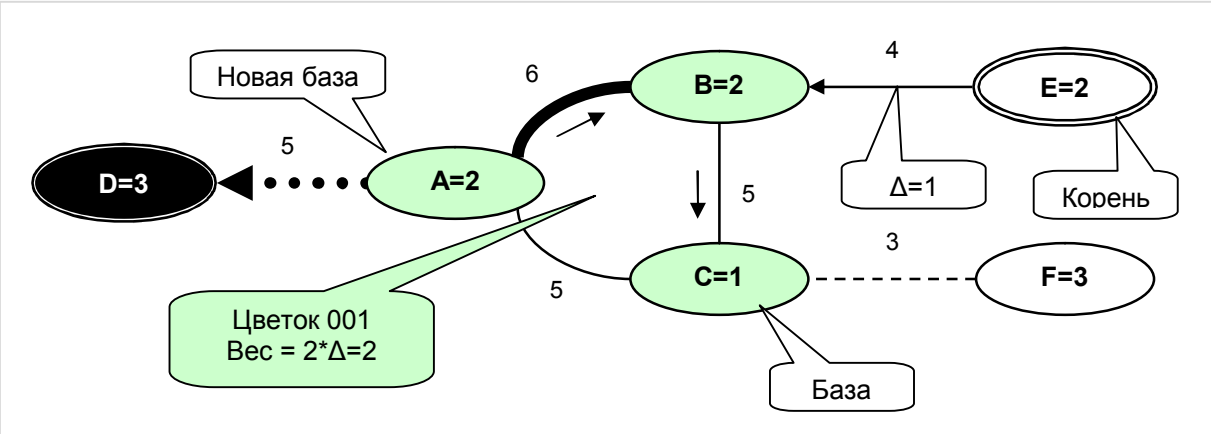


Рис. 29-25 — Построение дерева с корнем в E

Единственная возможность дальнейшего расширения состоит в том, чтобы скорректировать веса всех вершин текущего дерева на величину компенсации, хранящейся в цветке (т.е. на половину веса цветка). Таким образом, веса всех внешних вершин дерева (*E,D*) уменьшаются на 1, а внутренних (*A,B,C*) — увеличиваются на 1. Вес внутреннего цветка *001* уменьшается на 2, и становится равным нулю.

Поскольку вес цветка упал до нуля, его роспуск не повлияет на баланс ни внутренних, ни внешних ему рёбер *A-D*, *B-E*, поэтому такой роспуск вполне допустим. В ходе роспуска двигаемся по цепи от новой базы к первичной базе в направлении «жирного» ребра, попутно инвертируя парные и непарные рёбра (двигаемся в направлении стрелок). Результат роспуска показан на Рис. 29-26.

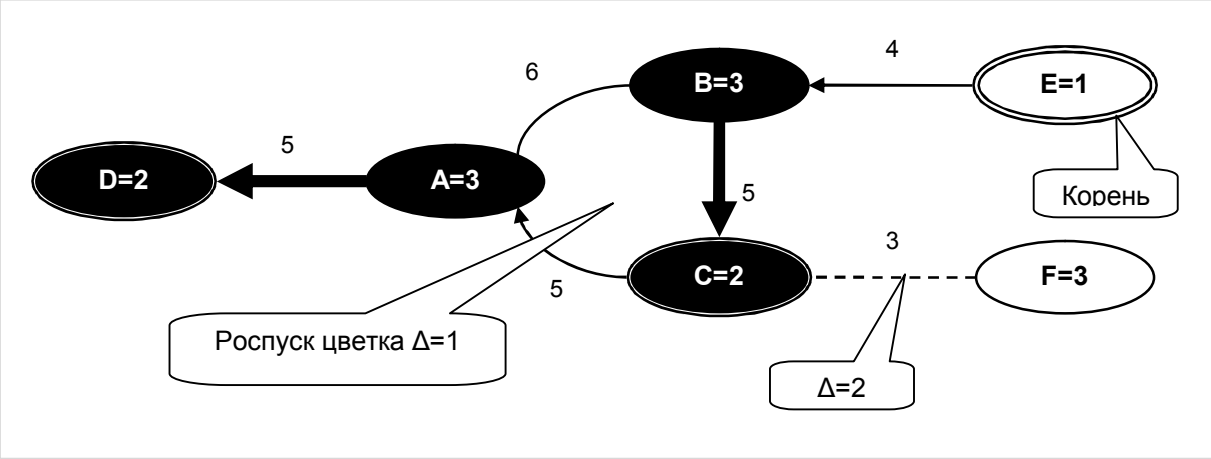


Рис. 29-26 — Перестройка дерева после роспуска цветка 001 (C-A=B)

После роспуска цветка дерево с корнем в *E* строим заново. Обратите внимание на то, что бывшая ранее *внутренней* (в цветке) вершина *C*, покинув цветок, стала *внешней*. Стало быть, ребро *C-F* тоже стало кандидатом на расширение дерева. После необходимой корректировки весов на величину Δ=2 находится аугментальная цепь *E-B=C-F* (Рис. 29-27).

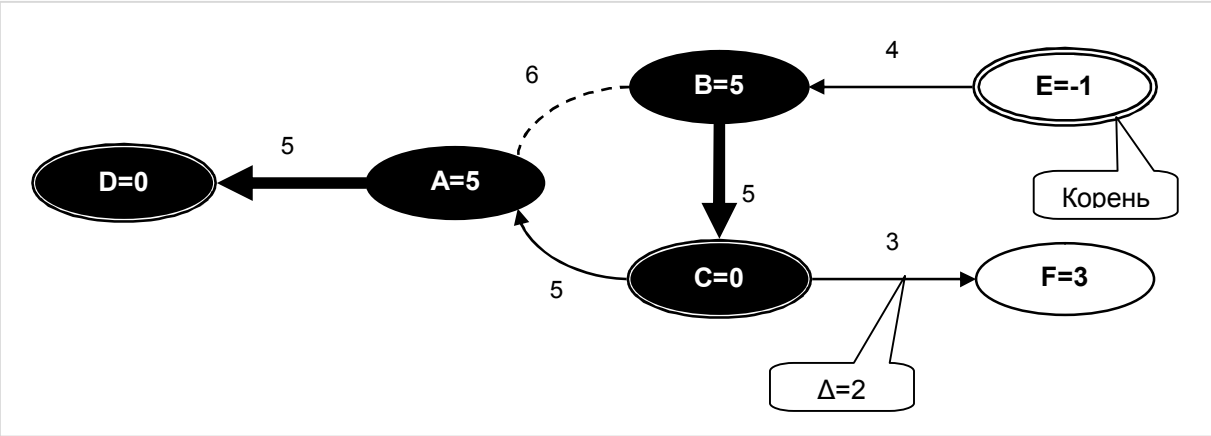


Рис. 29-27 — Корректировка весов и добавление ребра C-F

После реверса аугментальной цепи *E-B=C-F* получаем конечный результат, показанный на Рис. 29-28. Обратите внимание на то, что сумма весов всех покрытых (парных) вершин совпадает с суммой весов парных рёбер.

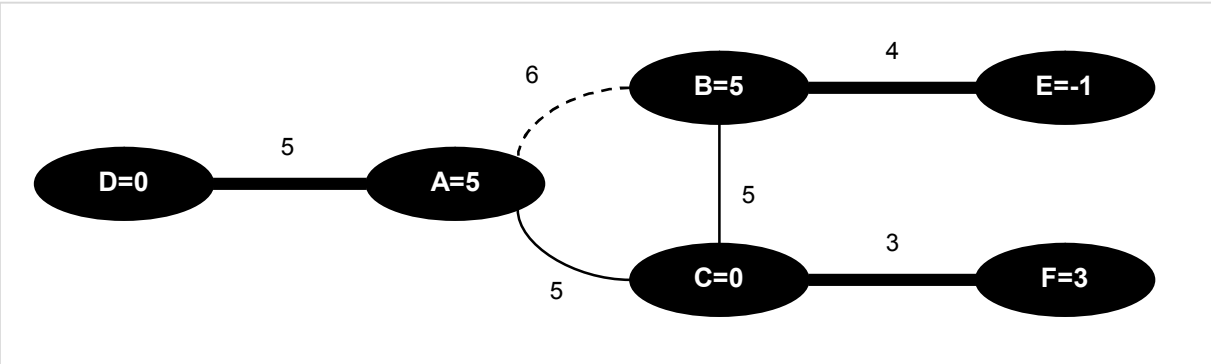


Рис. 29-28 — Конечный результат

Подведём промежуточные итоги.

- Поиск аугментальных цепей в графе со взвешенными рёбрами (и построение соответствующего дерева) выполняется только на подмножестве уравновешенных (доступных) рёбер. Ребро считается уравновешенным (доступным), если сумма весов смежных ему вершин в точности равна весу ребра.
- В начале работы алгоритма всем вершинам назначаются достаточно большие, но произвольные веса с тем условием, чтобы все рёбра или часть из них оказались неуравновешенными. Иными словами, сумма весов смежных ребру вершин должна быть не меньше веса ребра.
- По мере работы алгоритма веса вершин пересчитываются так, чтобы общий их вес всякий раз снижался в минимальной степени на некую величину Δ. В конечном итоге это приводит к тому, что все парные рёбра становятся уравновешенными, а их общий вес оказывается максимально возможным.
- На каждом шаге расширения дерева минимальное значение Δ выбирается из условия того, чтобы пересчёт весов вершин повлёк:

  a) либо присоединение к дереву новой вершины и ребра;
  b) либо присоединение к дереву ребра, замыкающего нечётный цикл;
  c) либо образование внутреннего цветка с нулевым весом.

- Для условия (a) необходимо исследовать все рёбра, ведущие из внешних вершин дерева в вершины, не принадлежащие дереву. В качестве Δa берётся минимальный дисбаланс исследуемых ребер.
- Для условия (b) исследуются рёбра, ведущие из внешней вершины в другую внешнюю вершину данного дерева. В качестве Δb берётся половина минимального дисбаланса исследуемых ребер.
- Для условия (c) исследуются веса внутренних цветков данного дерева. В качестве Δc берётся половина минимального веса внутреннего цветка (Δc может быть нулевым).
- В итоге из трёх упомянутых значений Δ выбирается минимальное, после чего:

  a) веса всех внешних вершин дерева уменьшаются на величину Δ;
  b) веса всех внешних цветков дерева увеличиваются на величину 2•Δ;
  c) веса всех внутренних вершин дерева увеличиваются на Δ;
  d) веса всех внутренних цветков дерева уменьшаются на 2•Δ;
  e) внутренние цветки, вес которых стал нулевым, распускаются; при наличии таковых дерево строится заново (от корня).

- После такой перестройки дерева суммарный вес всех его вершин уменьшается на величину Δ, но равновесие всех рёбер дерева сохраняется.

29.7. Слабая увеличивающая цепь

Описанные выше операции ведут, так или иначе, к последовательному расширению дерева поиска вплоть до одного из двух событий: a) до обнаружения непокрытой вершины, b) до получения «тупикового» венгерского дерева. При этом на каждом промежуточном шаге расширения образуются исходящие из корня венгерские цепи, ограниченные внешними вершинами. Среди таких цепей могут оказаться упомянутые ранее *слабые увеличивающие цепи* (см. Рис. 29-4). Инверсия такой цепи не расширяет паросочетания, но увеличивает его вес.

Вполне естественно искать такие цепи по ходу расширения дерева, стараясь избегать при этом существенных накладных расходов. В данном алгоритме это достигается подсчётом так называемых *профитов* внешних вершин, среди которых отличают *линейный* профит, и профит внутри цветка (*цветочный*).

Линейный профит представляет собой сумму весов рёбер, лежащих в цепи между корнем и данной вершиной, причём вес непарных рёбер берётся со знаком плюс, а парных — со знаком минус. Такой подсчёт выполняется в ходе расширения дерева при добавлении к нему очередной пары вершин (внутренней и внешней). Если профит какой-либо *внешней* вершины оказывается больше нуля, значит, эта вершина замыкает слабую увеличивающую цепь. И тогда она помечается как непокрытая (белая) и возвращается в главную очередь обработки, а оставшаяся часть цепи инвертируется.

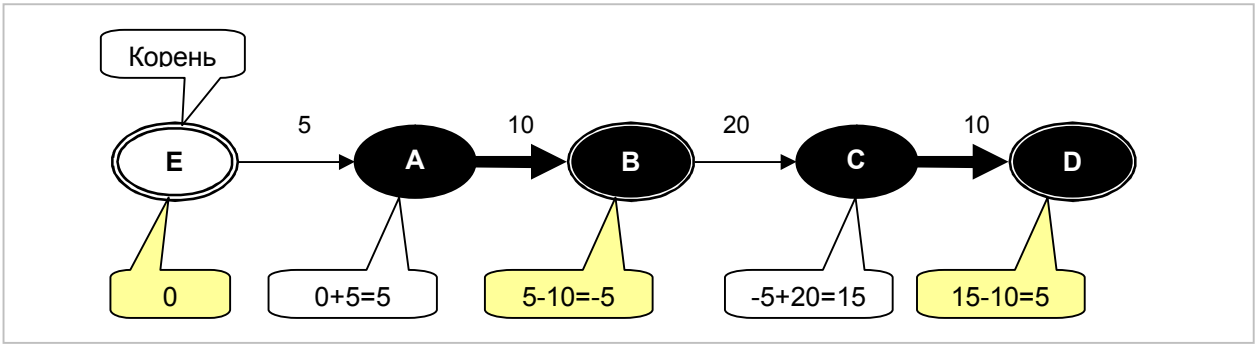


Рис. 29-29 — Подсчёт линейного профита

Подсчёт линейного профита в ходе расширения дерева показан на Рис. 29-29. Здесь профит внешней вершины *D* оказывается больше нуля, — эта вершина замыкает слабую увеличивающую цепь.

Труднее подсчитать линейный профит при наличии в цепи цветков. Здесь необходимо учесть внутренний путь, по которому пойдёт инверсия внутри цветка (как если бы цветок распускался нужным образом из новой базы цветка).

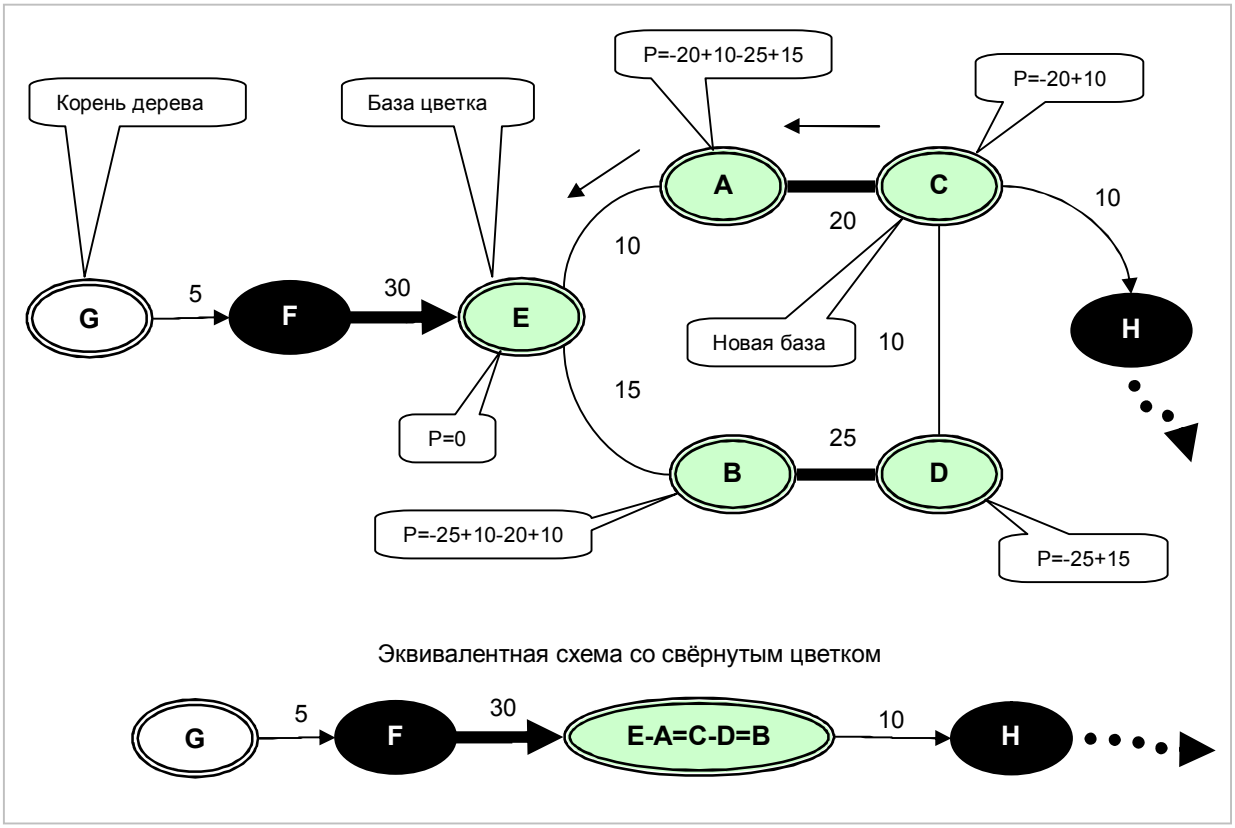


Рис. 29-30 — Вычисление профитов внутри цветка, случай для вершины *C*

Рассмотрим Рис. 29-30, где в одной из ветвей дерева оказался цветок *E-A=C-D=B*. В нижней части рисунка показан тот же фрагмент цепи со свёрнутым цветком. Если бы на месте цветка была простая вершина, то линейный профит для вершины *H* составил бы +5-30+10 = -15. Однако с учётом рёбер внутри цветка к этому результату следует добавить профит внутренней цепи +10-20. В итоге профит для вершины *H* составит -25.

На Рис. 29-30 проставлены «цветочные» профиты (обозначены буквой *P*) для всех вершин цветка. Нетрудно заметить, что по смыслу они представляют собой изменение общего веса рёбер цветка при перемещении новой базы в данную вершину. В этом примере в случае инверсии цепи новая база должна перейти из вершины *E* в вершину *C*. Для первичной базы профит равен нулю.

В цветках со сложной многоуровневой структурой, когда одни цветки вложены в другие, весьма непросто проследить внутреннюю цепь при перемещении новой базы из одной вершины в другую. К тому же это требует и существенных накладных расходов. Проще при создании цветка заранее посчитать внутренние «цветочные» профиты вершин по отношению к первичной базе. И тогда при переходе новой базы из вершины *X* в вершину *Y*, внутреннее изменение профита можно вычислить как разность P(*Y*)-P(*X*), где P(*N*) — профит вершины *N* относительно первичной базы.

Подведём краткий итог.

- В ходе расширения дерева поиска необходимо отслеживать в числе прочего появление слабых увеличивающих цепей. Такая цепь обнаруживает себя тем, что замыкающая её внешняя вершина приобретает не нулевой положительный линейный профит.
- Линейный профит подсчитывается суммированием веса парных и непарных рёбер цепи на пути от вершины к корню, причём вес парных рёбер берётся со знаком минус.
- Когда в указанной цепи встречается цветок, необходимо вдобавок учесть внутреннюю цепь: путь, проходимый в цветке — цветочный профит.
- Цветочный профит соответствует «выгоде» при перемещении новой базы из одной вершины в другую; новая база изменяется ввиду инверсии смежных цветку рёбер.
- Для упрощения подсчёта цветочных профитов они вычисляются заранее в процедуре создания цветка.

Осветив основные идеи рассматриваемого алгоритма, перейдём к его программной реализации.

29.8. Программная структура вершин и цветков

Прежде всего, рассмотрим структуру цветков, создаваемых из нечётных циклов графа. С одной стороны, такие цветки заменяют собою группу вершин, а с другой стороны должны рассматриваться как цельная вершина. Потому во избежание путаницы сразу условимся о терминах. *Простой вершиной* будем называть вершину графа в обычном понимании. *Цветком* или *псевдо-вершиной* назовём то, что создано из группы вершин. Термин *обобщённая вершина* или просто *вершина* (без уточнения, если это понятно из контекста) будет относиться к обоим этим объектам: и к простым вершинам, и к цветкам.

Ввиду того, что цветок должен обладать многими свойствами простой вершины, вполне естественно породить его от объекта-предка **TNode**.

```
TNodeInt = class (TNode) // Вершина, визуально помечаемая числами
. . .
end;


TBlossom = class(TNodeInt) // Объект-цветок (псевдо-вершина)
 . . .
end;
```

Цветки, в отличие от простых вершин, будем помечать не символами, а числами. Отличать их в программе позволяет конструкция вида:

```
if Node is TBlossom then begin
    // Обращаемся к полям и методам цветка и простой вершины
end else begin
    // Обращаемся к полям и методам только простой вершины
end;
```

Некоторые процедуры и функции рассматриваемого алгоритма будут реализованы как методы классов **TNode** и **TBlossom**. Рассмотрим их здесь.

В объекте **TNode** алгоритмом будут задействованы следующие поля и методы, детали использования которых будут пояснены на примерах.

```
  TNode = class (TItem)    // Вершина
      mRoot    : TItem;    // Корень дерева
      mDist    : integer;  // Линейный профит от корня дерева
      mPred    : TNode;    // Предшествующая обобщённая вершина в цепи
      mLink    : TLink;    // Линк на предшествующую простую вершину в цепи
      mBlossom: TNode;     // Ссылка на цветок (TBlossom)
      mPair    : TLink;    // Линк для связи с парной вершиной или цветком
      mRight   : TNode;    // Следующая вершина при обходе по часовой
      mLeft    : TNode;    // Следующая вершина при обходе против часовой
      mRLnk    : TLink;    // Линк при обходе цветка по часовой
      mOut     : boolean;  // Статус в альтернирующей цепи:
                           // false -- внутренняя, true -- внешняя
      mOwnDest : boolean;  // Простая парная вершина в цветке:
                           // false -- источник, true -- приёмник линка
      mProfit : integer;   // Профит (выгода) внутри цветка
                           // от перемещении базы в данную вершину
      function GetBlossom: TNode; // Возвращает крайний охватывающий цветок
      function GetPair: TNode;    // Возвращает парную простую вершину
      . . .
  end;
```

Поля и методы объекта «цветок» представлены ниже.

```
TBlossom = class(TNodeInt) // Объект-цветок (псевдо-вершина)
    mBase: TNode;      // Первичная база цветка
    mNewBase: TNode;   // Новая (текущая) база цветка
    mNext: TNode;      // Парная вершина текущей базы
                       // указывает направление роспуска цветка
               // Проверяет наличие простой вершины в данном цветке
    function IsPresent(aNode: TNode): boolean;
               // Устанавливает новую базу цветка
               // и возвращает вершину mNext
    function SetNewBase(aNewBase: TNode): TNode;
               // Возвращает корневую первичную базу цветка
    function  GetBase: TNode;
               // Возвращает корневую новую базу цветка
    function  GetNewBase: TNode;
               // Функция возвращает выгоду внутреннего пути
               // между простыми вершинами aIn и aOut
    function GetProfit(aIn, aOut : TNode): integer;
    function GetName: string; override;
    procedure Print(var aFile: TextFile); override;
end;
```

Рассмотрим применение полей и методов на графе, показанном на Рис. 29-31. Здесь перед построением дерева из корня *P* уже образовано несколько пар, однако

не создано ни одного цветка. Поля **mBlossom** всех вершин указывают на сами эти вершины. Далее по мере расширения дерева будет создано несколько вложенных цветков.

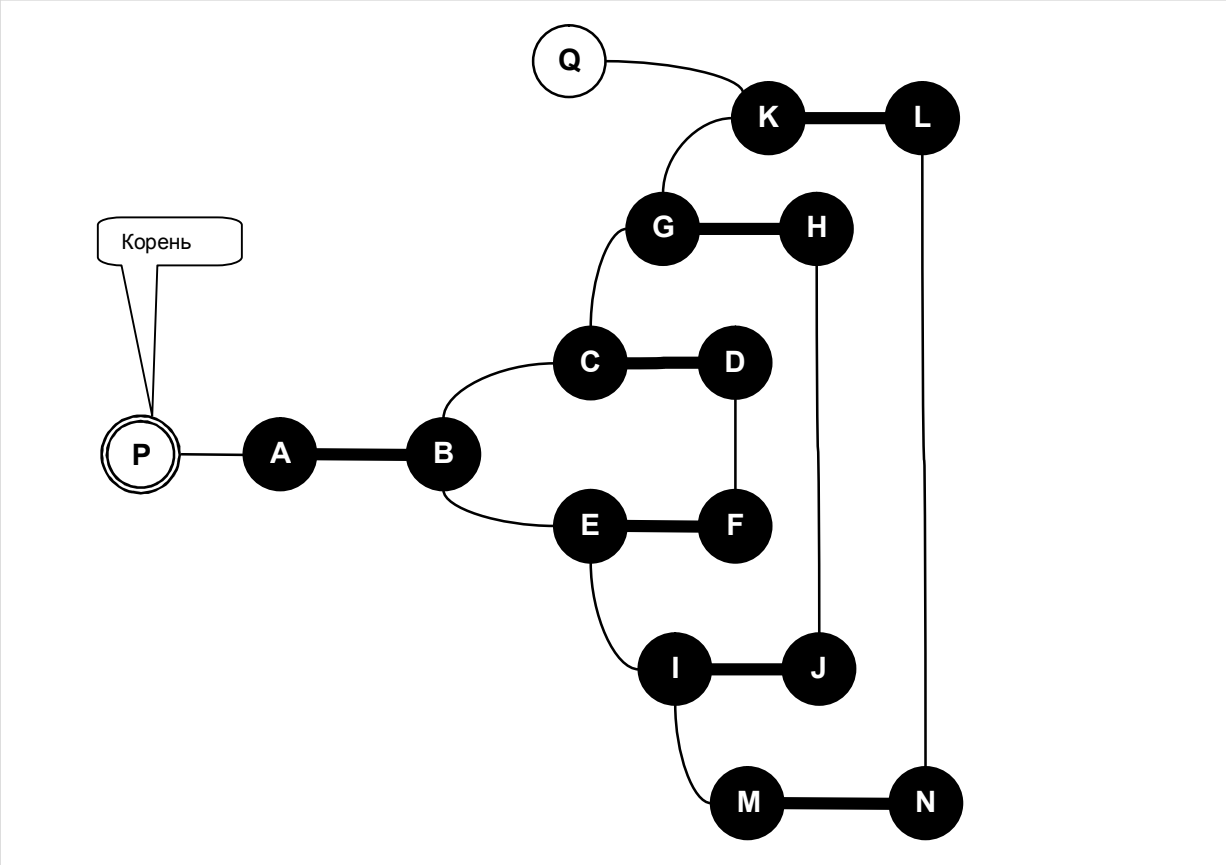
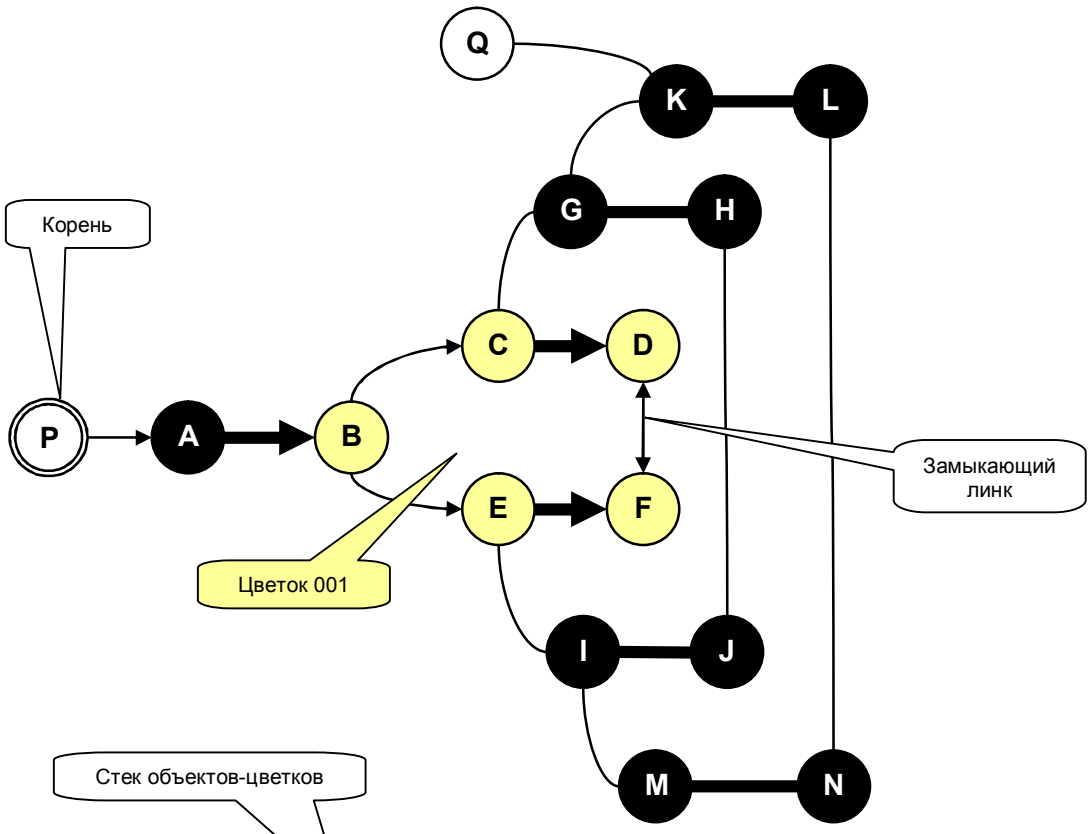


Рис. 29-31 — Состояние графа на момент постройки дерева из корня P

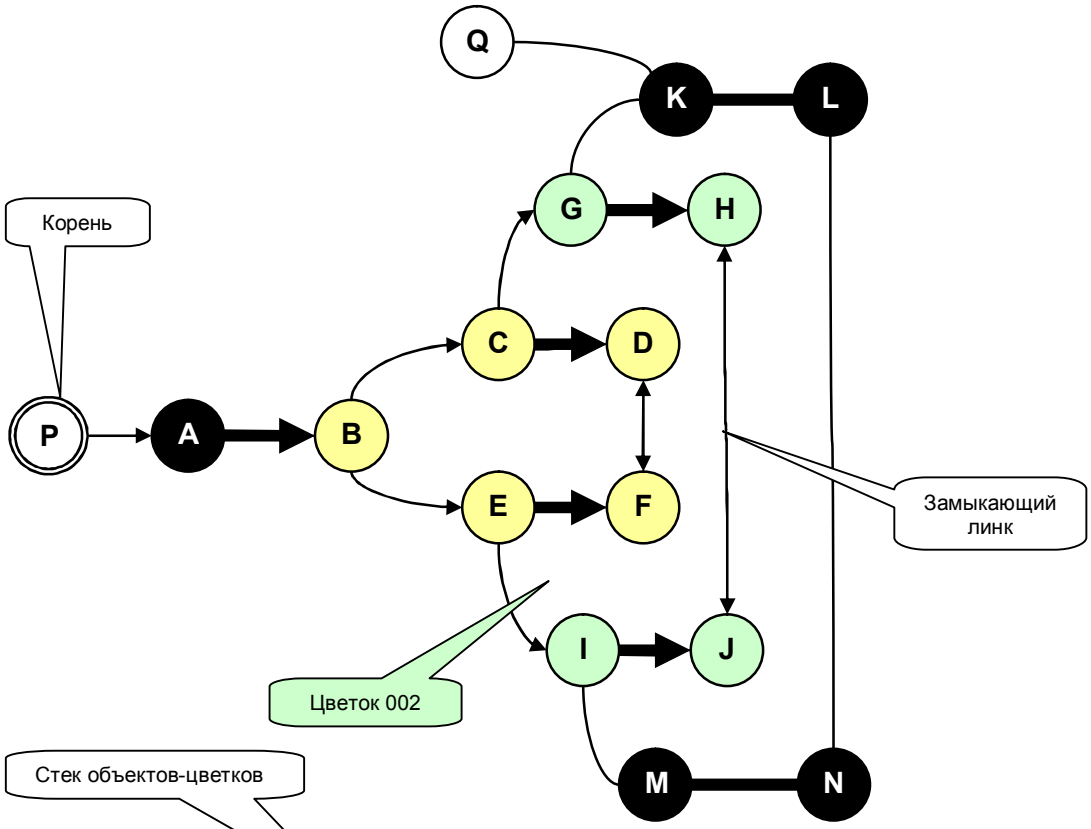
В ходе расширения дерева обнаруживается линк *D-F*, замыкающий нечётный цикл, и в результате создаётся цветок *001* (Рис. 29-32). Поля **mBlossom** всех входящих в него вершин ссылаются теперь на этот цветок. Сам по себе объект-цветок хранится отдельно от графа в специальном стеке. Базой цветка является вершина *B*, а парой — вершина *A*.



Цветок	База	Новая база	Ссылки из вершин	Пара
001	B	B	B. C, D, E, F	A

Рис. 29-32 — Образование цветка 001

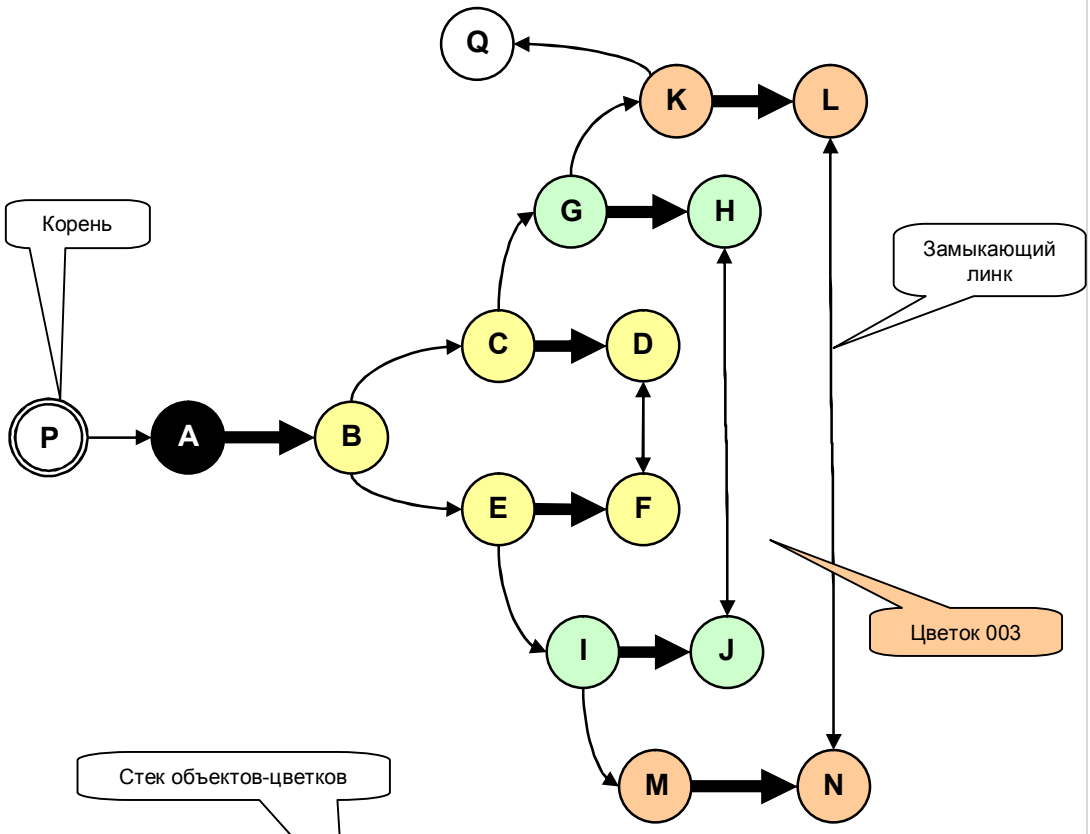
Далее по мере расширения дерева (Рис. 29-33) обнаруживается замыкающий линк *H-J* и создаётся цветок *002*. Базой цветка является цветок *001*, а парой по-прежнему вершина *A*. Поле **mBlossom** цветка *001* теперь указывает на цветок *002* (то же и для прочих простых вершин цветка *002*).



Цветок	База	Новая база	Ссылки из вершин	Пара
001	B	B	B. C, D, E, F	A
002	001	001	001, G, H, I, J	A

Рис. 29-33 — Образование цветка 002

И, наконец (см. Рис. 29-34), после обнаружения замыкающего линка *L-N* создаётся цветок *003*, базой которого становится цветок *002*. Эквивалентный граф показан на Рис. 29-35, здесь обнаруживается аугментальная цепь, результат инверсии которой показан на том же рисунке.



Цветок	База	Новая база	Ссылки из вершин	Пара
001	B	B	B. C, D, E, F	A
002	001	001	001, G, H, I, J	A
003	002	002	002, K, L, M, N	A

Рис. 29-34 — Образование цветка 003

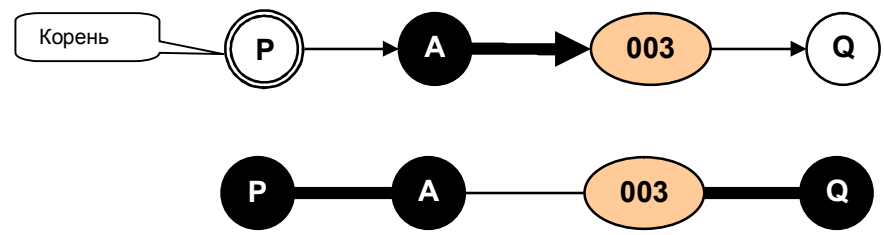
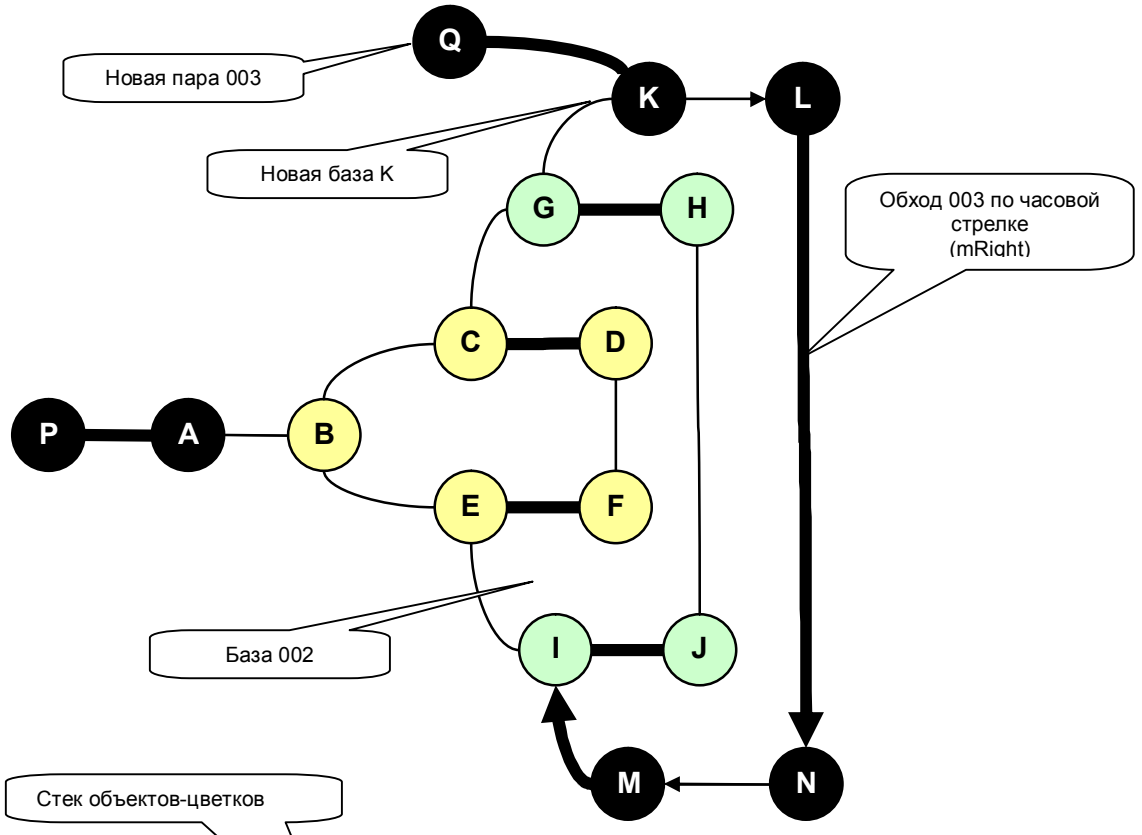


Рис. 29-35 — Эквивалентная схема до и после инверсии аугментальной цепи

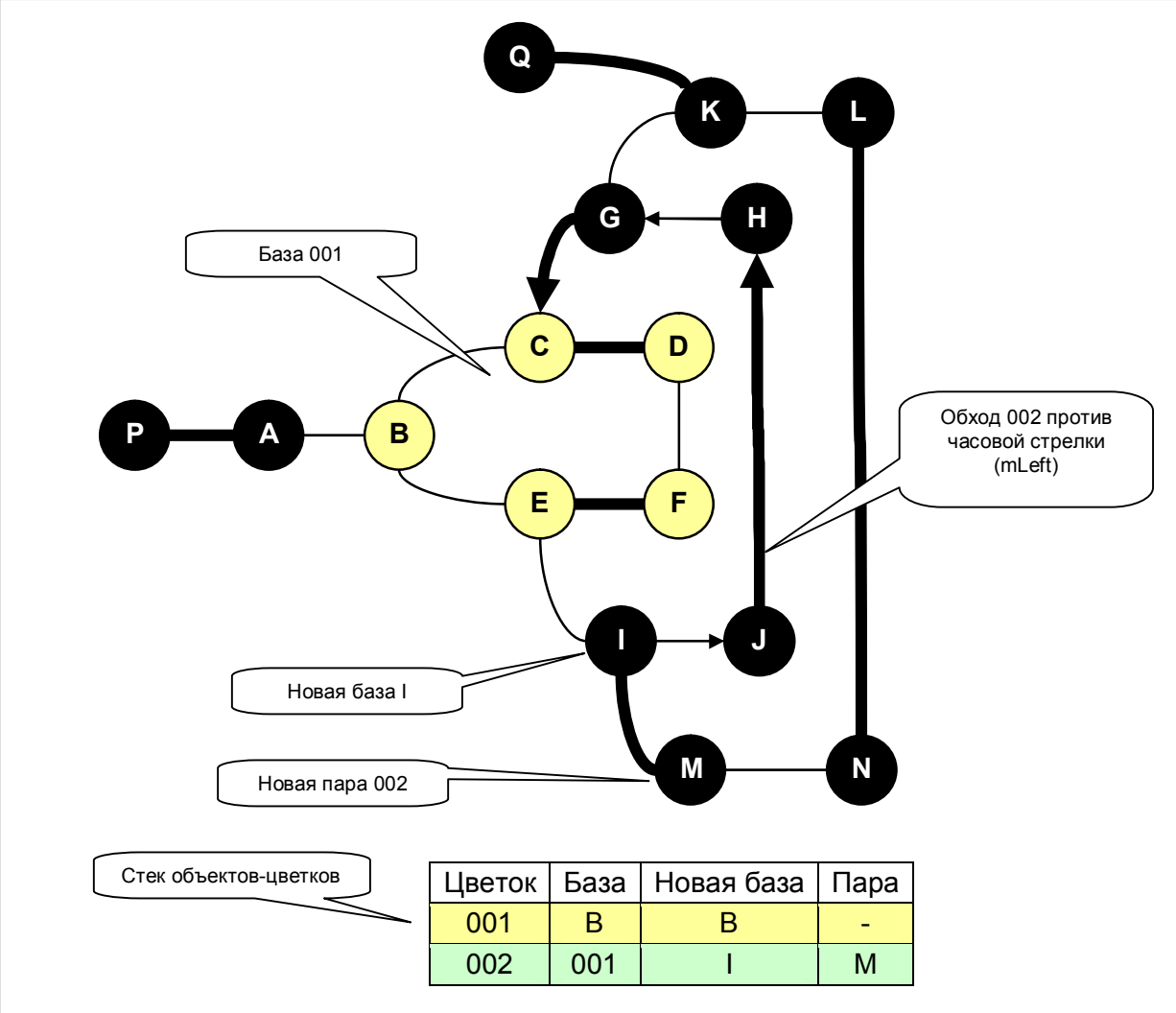
После обработки всех вершин графа наступает этап роспуска цветков, выполняемый в порядке, обратном их созданию. После того, как парой цветка *003* стала вершина *Q*, смежная ей вершине цветка *K* становится новой базой этого цветка. При его роспуске необходимо пройти путь от новой базы *K* к первичной базе *002* в направлении «жирного» парного ребра *K=L*. В данном случае это выполняется по указателям **mRight**. На Рис. 29-36 показано направление обхода и результат инверсии рёбер в цветке. После роспуска цветка поля **mBlossom** всех входивших в него вершин вновь указывают на сами эти вершины, — тем самым разрывается их связь с цветком. Затем цветок удаляется из стека и уничтожается.



Цветок	База	Новая база	Пара
001	B	B	-
002	001	001	-
003	002	K	Q

Рис. 29-36 — Роспуск цветка 003

Далее наступает очередь цветка *002* (Рис. 29-37). Его новой парой становится вершина *M*, а новой базой — вершина *I*. Обход выполняется в направлении пары *I=J* против часовой стрелки по указателям **mLeft**. На рисунке показан результат инверсии.



Цветок	База	Новая база	Пара
001	B	B	-
002	001	I	M

Рис. 29-37 — Роспуск цветка 002

Последним распускается самый первый из созданных — цветок *001*, результат роспуска показан на Рис. 29-38.

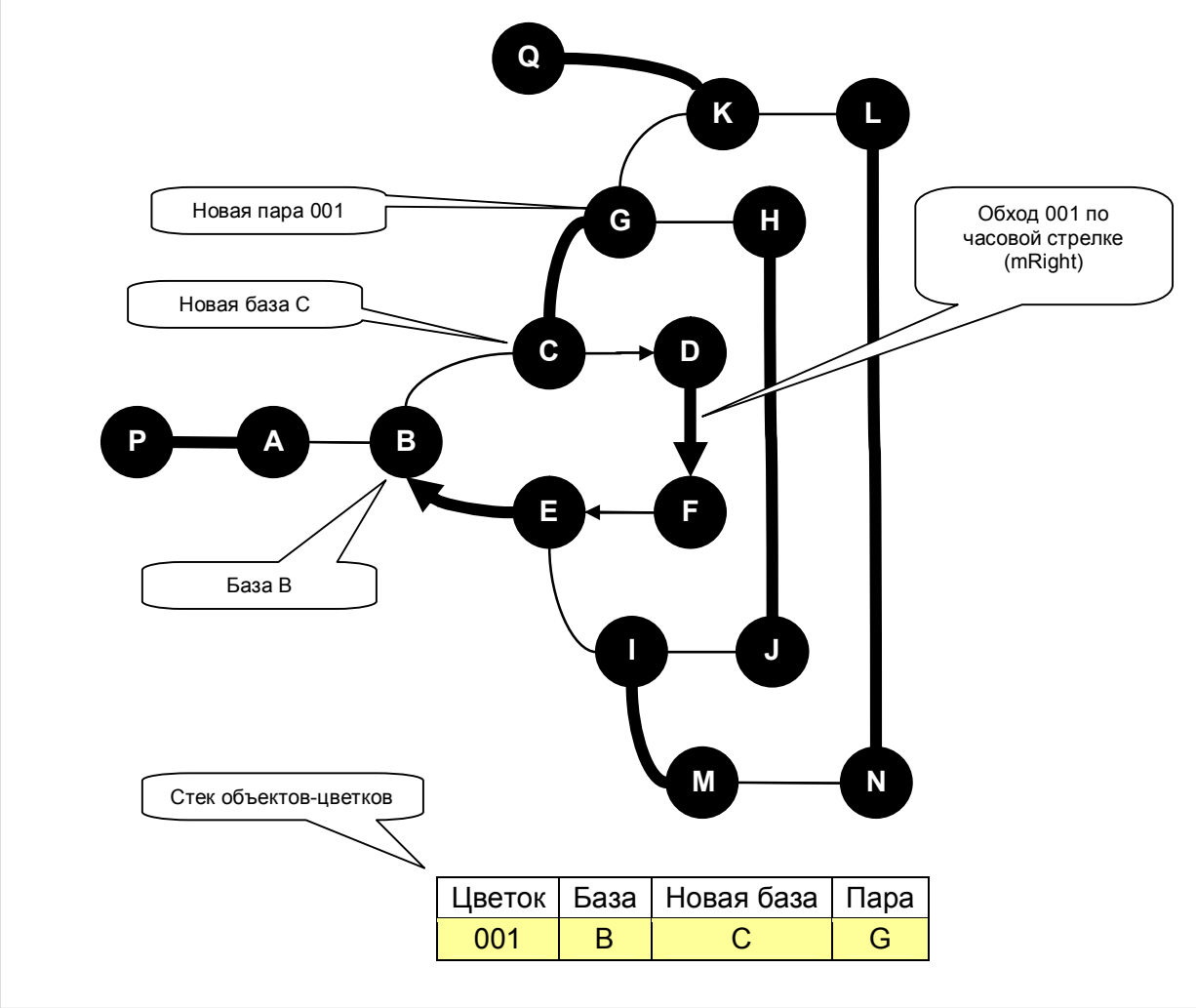


Рис. 29-38 — Роспуск цветка 001

Подытожим информацию о структуре цветков.

- Объек-цветок типа **`mBlossom`** унаследован от объекта-вершины графа **`TNode`**. Созданные объекты этого типа хранятся в отдельном стеке, они не связаны напрямую какими либо рёбрами с графом.
- Взаимные связи между обычными вершинами графа и цветками осуществляются посредством ссылок. В цветке такими ссылками являются указатели на первичную и новую базу цветка (**`mBase`**, **`mNewBase`**). При создании цветка оба они ссылаются на одну и ту же вершину. После изменения пары цветка, указатель новой базы перестраивается на соответствующую вершину внутри цветка.
- Связь вершин с цветком осуществляется через указатели **`mBlossom`**. Если вершина не включена в цветок, этот указатель ссылается на саму эту вершину. При обходе вершин в ходе роспуска цветка используются указатели **`mRight`** и **`mLeft`**.

- То, что упомянутые выше поля присутствуют как в обычных вершинах, так и в цветках, позволяет создавать иерархические структуры любой сложности. Иными словами, цветки могут входить в качестве обобщённых вершин в другие, охватывающие их цветки, в том числе, в качестве базы.
- Роспуск цветков выполняется обычно в порядке, обратном их созданию. Исключение — досрочный роспуск внутренних цветков, описанный ранее. В любом случае охватывающий цветок всегда распускается ранее охватываемых.

29.9. Методы вершин и цветков

Теперь рассмотрим несколько простых методов для вершин и цветков, существенно упрощающих обращение с этими объектами в контексте рассматриваемой задачи.

Поле **mBlossom**, как было сказано выше, ссылается на цветок, к которому принадлежит данная обобщённая вершина *непосредственно*. Следующий ниже рекурсивный метод находит *крайний* охватывающий цветок для этой вершины.

```
function TNode.GetBlossom: TNode;
begin
  if mBlossom <> Self
    then Result:= mBlossom.GetBlossom
    else Result:= Self;
end;
```

Парная связь двух обобщённых вершин осуществляется посредством полей **mPair: TLink** и **mOwnDest: boolean**. Как видно по Рис. 29-39, поля **mPair** обеих парных вершин содержит один и тот же линк, однако в полях **mOwnDest** установлены противоположные значения.

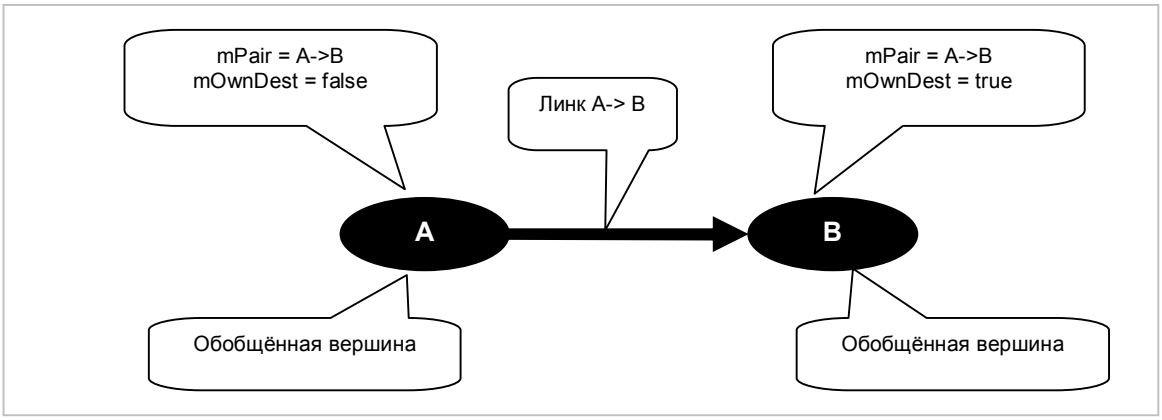


Рис. 29-39 — Связь двух обобщённых парных вершин

Когда возникает необходимость определить *простую* парную вершину, которая, возможно, лежит на некоторой глубине внутри цветка, вызывается следующий метод.

```
// Получение ссылки на простую парную вершину, если она существует.
// Используются поля:
// mPair    : TLink;      -- Парный линк
// mOwnDest  : boolean; -- Положение данной вершины в паре:
//                       false -- внутр, true -- внешняя

function TNode.GetPair: TNode;
begin
  Result:= nil;
  if not Assigned(mPair) then Exit;
  if mOwnDest
    then Result:= mPair.mOwner  // источник линка
    else Result:= mPair.mDest   // приёмник линка
end;
```

Следующая функция выясняет, существует ли некоторая вершина в составе данного цветка на любом уровне вложения. Здесь поиск происходит изнутри цветка наружу.

```
// Проверка наличия вершины aNode в данном цветке

function TBlossom.IsPresent(aNode: TNode): boolean;
var N: TNode;
begin
  N:= aNode;
  repeat
    N:= N.mBlossom;       // Непосредственный цветок
    Result:= N = Self;    // true - вершина в этом цветке
    if Result then Break;// Выход, если aNode существует в данном цветке
  until N = N.mBlossom;  // Выход, если достигнут крайний охватывающий цветок
end;
```

Представленные ниже две рекурсивные функции возвращают простые вершины, которые являются корневыми базами данного цветка: соответственно первичной и новой базы.

```
// Возвращает корневую первичную базу цветка
// Корневая база является простой вершиной

function  TBlossom.GetBase: TNode;
begin
  // Двигаемся рекурсивно от наружного цветка внутрь
  if mBase is TBlossom
    then Result:= (mBase as TBlossom).GetBase
    else Result:= mBase;  // здесь достигли простой вершины
end;

// Возвращает корневую новую базу цветка
// Корневая база является простой вершиной

function  TBlossom.GetNewBase: TNode;
begin
  // Двигаемся рекурсивно от наружного цветка внутрь
  if mNewBase is TBlossom
    then Result:= (mNewBase as TBlossom).GetNewBase
    else Result:= mNewBase;  // здесь достигли простой вершины
end;
```

Наряду с методами вершин и цветков рассмотрим ещё две простые вспомогательные функции.

В ходе расширения дерева поиска те или иные обобщённые вершины включаются в данное текущее дерево пометкой поля **mRoot**. При этом нет необходимости помечать все входящие в некоторый цветок вершины, достаточно пометить сам цветок. Тогда корневую вершину дерева можно получить функцией:

```
// Функция возвращает корень текущего дерева,
// к которому принадлежит крайний цветок вершины aNode

function  GetRoot(aNode: TNode): TNode;
begin
  Result:= aNode.GetBlossom.mRoot as TNode;
end;
```

Когда вершина или цветок присоединяется к дереву поиска, они помечаются либо как внутренние, либо как внешние по отношению к корню. Здесь также достаточно помечать только объект-цветок, поскольку статус всех входящих в него вершин приобретает это же значение, и возвращается следующей функцией:

```
// Функция возвращает статус крайнего охватывающего вершину цветка
// (true -- внешний, false -- внутренний)

function  GetOut(aNode: TNode): boolean;
begin
  Result:= aNode.GetBlossom.mOut;
end;
```

Далее следует обзор метода поиска паросочетания алгоритмом Эдмонса. Полный текст данного метода представлен в приложении G.

29.10. Метод TGraph.GenPairs_Edmonds(aMode: TPairs): TCostSet

Метод поиска паросочетания **TGraph.GenPairs_Edmonds** принимает один параметр и возвращает множество пар. Параметр **aMode** может принимать три значения:

- **pMaxN** — ищется паросочетание максимальной мощности без учёта веса рёбер;
- **pMaxW** — ищется оптимальное паросочетание максимального веса;
- **pMinW** — ищется оптимальное паросочетание минимального веса.

В теле метода инициализируются вспомогательные объекты:

- **gMainQue : TBuffer** — Главная очередь вершин; вначале сюда помещаются все вершины графа. Затем, по мере их обработки, при обнаружении слабой увеличивающей цепи некоторые вершины

возвращаются в начало этой очереди. Процедура поиска завершается по исчерпании очереди.

- **gQue : TBuffer** — Очередь вершин для расширения текущего дерева; используется для обхода в ширину.
- **gAppendix: TSet** — Множество добавляемых в очередь **gQue** вершин. Всякий раз, когда после пересчёта вершинных чисел к дереву присоединяются очередные рёбра, соответствующие им внешние вершины дерева сначала попадают в это множество, а затем в очередь расширения **gQue**.
- **gBlossomStack : TBuffer** — Стек для хранения цветков.
- **gBlosNum: integer** — Уникальный идентификатор цветка, служит для сквозной нумерации цветков.

Далее следует обработка главной очереди вершин. Если очередная вершина не имеет пары, вызывается процедура её обработки **Handle**. По завершении обработки всех вершин вызывается процедура роспуска оставшихся нераскрытых цветков, после чего формируется результат — множество пар, а затем удаляются все вспомогательные объекты.

procedure Handle(aNode: TNode)

Обработка очередной непокрытой вершины начинается с попытки обнаружить для неё альтернирующую цепь функцией **Alter**. Если такая цепь обнаружена, вызывается процедура её реверса **Reverse**. Если обнаружена слабая увеличивающая цепь, то крайняя её вершина отрывается от своей пары и возвращается в начало главной очереди **gMainQue**, а остаток цепи инвертируется. При ином исходе (построено венгерское дерево) никаких действий не выполняется.

function Alter(aRoot: TNode): TNode

Корневая вершина ставится в очередь расширения дерева **gQue**. Затем для всех содержащихся в **gQue** вершин вызывается функция **NodeHandle**, которая пытается расширить дерево на уже уравновешенных (доступных) рёбрах и найти либо альтернирующую, либо слабую увеличивающую цепь. При успехе функция **Alter** возвращает крайнюю вершину этой цепи (непокрытую — для альтернирующей цепи, покрытую — для слабой увеличивающей цепи).

Если упомянутого результата не достигнуто, вызывается функция **CalcForExpand**, которая пытается определить минимальный вычет **DeltaMin** (Δ) для пересчёта вершинных чисел. Если подходящее значение **DeltaMin** существует, вызывается функция **ChangeValues**, которая пересчитывает вершинные числа дерева и распускает, если возможно, некоторые внутренние цветки. Если произошёл роспуск хотя бы одного такого цветка, то корневая вершина **aRoot** возвращается в начало главной очереди вершин **gMainQue**, а функция **Alter** возвращает **nil**. Таким образом, дерево из этого корня строится заново. Если цветки не распускались, то сформированное в **CalcForExpand**

множество вершин **gAppendix** добавляется в очередь расширения дерева **gQue**, и обработка этой очереди продолжается.

Если функция **CalcForExpand** не нашла возможности пересчёта вершинных чисел или роспуска цветков, то функция **Alter** возвращает **nil**.

```
function NodeHandle(aNode: TNode): TNode;
```

Вызывается из функции **Alter**, и обрабатывает очередную внешнюю вершину дерева **aNode**. Если данная вершина замыкает слабую увеличивающую цепь, то **NodeHandle** возвращает эту вершину. Если **aNode** смежна непокрытой «белой» вершине, то **NodeHandle** возвращает эту непокрытую вершину. Иначе ставит в очередь расширения дерева **gQue** покрытые внешние вершины, которые доступны через смежные **aNode** внутренние вершины, и возвращает **nil**. Учитываются только уравновешенные рёбра.

Если в ходе расширения дерева обнаруживается линк, соединяющий две внешние вершины текущего дерева (замыкающий линк), то из **NodeHandle** вызывается функция создания цветка **BlossomMake**. Внутренние вершины цветка обретают статус внешних, и ставятся в очередь расширения дерева. Сам цветок помещается в стек цветков.

```
function BlossomMake(aLink: TLink): TBlossom
```

Вызывается из **NodeHandle**, принимает линк, замыкающий нечётный цикл, и создаёт объект-цветок. Вначале, путём движения по обратным ссылкам, находится база цветка. Далее в обобщённых вершинах проставляются ссылки для обхода цветка вправо и влево (по часовой, и против часовой стрелки). Затем внутренние вершины этого нечётного цикла ставятся в очередь расширения дерева, и все вершины цикла метятся данным цветком (поле **mBlossom**). В завершение для всех обобщённых вершин вычисляются профиты относительно базы цветка.

```
function CalcForExpand(var aSuccess: boolean): integer;
```

Вызывается из функции **Alter** после исчерпания очереди расширения текущего дерева. Вычисляет минимальный вычет **DeltaMin** (Δ) для пересчёта вершинных чисел. Возвращает признак **aSuccess=true**, если такой пересчёт может привести к расширению дерева. Формирует, наряду с этим, множество внешних вершин текущего дерева **gAppendix**, которые необходимо вернуть в очередь расширения дерева **gQue**. Минимально возможное значение результата **DeltaMin** (Δ) равно нулю.

Поначалу в стеке цветков исследуются внутренние цветки, и среди них выбирается минимальное значение **mValue**. Результат **mValue/2** берётся в качестве начального значения **Result**. Если оно равно нулю, функция завершает работу.

Затем исследуются неуравновешенные рёбра всех внешних вершин текущего дерева. Если ребро **L** вершины **Node** соединяет эту вершину с вершиной, не принадлежащей дереву, то для него значение Δ вычисляется по формуле:

```
d:= L.mOwner.mValue + L.mDest.mValue – L.mValue
```

Если же смежная вершина является внешней вершиной того же дерева, то значение Δ берётся вдвое меньше:

```
d:= (L.mOwner.mValue + L.mDest.mValue – L.mValue)/2
```

В первом случае корректировка вершинных чисел на величину **d** может привести к добавлению к дереву новых вершин, во втором — к образованию новых цветков. Так, или иначе, значение **d** сравнивается с текущим результатом **Result**. Если **d < Result**, то значение **Result** обновляется, множество добавляемых в очередь вершин **gAppendix** очищается, и к нему добавляется вершина **Node**. Если же **d=Result**, то вершина **Node** просто добавляется к множеству **gAppendix**.

По окончании обработки всех вершин может случиться так, что значение **Result** окажется дробным. В таком случае, во избежание операций с дробными числами, результат умножается на 2, и вызывается процедура **Mul_2**, которая масштабирует все рёбра графа, вершинные числа и профиты умножением на 2.

Может случиться и так, что к текущему дереву уже невозможно добавить ни вершин, ни рёбер, и тогда функция возвращает **aSuccess=false**. Принявшая такой результат функция **Alter** завершается с результатом **nil**.

```
function ChangeValues(aDelta: integer): boolean
```

Эта функция пересчитывает вершинные числа текущего дерева, и вызывается из функции **Alter** в том случае, если функция **CalcForExpand** вернула **aSuccess=true**. После пересчёта вершинных чисел, проверяются также все внутренние цветки текущего дерева и распускаются те из них, у которых оказалось **mValue=0**. Если таким образом будет распущен хотя бы один цветок, функция **ChangeValues** возвращает **true**. Получив такой результат, функция **Alter** возвращает корневую вершину в начало главной очереди и завершает свою работу, дерево из этого корня будет построено заново. При ином результате вершины из множества **gAppendix** переносятся в очередь расширения дерева **gQue**, и процесс расширения дерева с поиском альтернирующей цепи продолжается.

```
procedure RemoveBlossoms

procedure BlossExpand(aBloss: TBlossom)
```

После обработки в теле метода всех вершин главной очереди **gMainQue** выполняется роспуск оставшихся нераспущенными цветков. Процедуры роспуска принимают очередной извлечённый из стека цветок, распускают его и уничтожают.

Если распускаемый цветок соединён в пару, то к моменту роспуска его новая база **mNewBase** уже определена. Если она не совпадает с первичной базой **mBase**, то выполняется обход вершин цветка, начиная с **mNewBase** в направлении **mBase** либо по часовой, либо против часовой стрелки с инверсией этой цепи.

Если распускаемый цветок не соединён в пару, то новой базой **mNewBase** назначается вершина цветка с минимальным вершинным числом (на любом уровне вложения). Установка новой базы выполняется рекурсивно изнутри вовне. Затем вершины цветка обходятся, либо по часовой, либо против часовой стрелки с инверсией цепи так, как указано выше.

function PairsMake: TCostSet

Данная функция вызывается из тела метода, и формирует результирующее множество пар, исходя из ранее проставленных пометок полей **mPair**.

29.11. Тестирующие программы

Ниже рассмотрены тестовые программы для испытания разработанного здесь метода **GenPairs_Edmonds**. В первую очередь следует проверить его корректность на большом количестве случайных графов. Воспользуемся разработанным ранее методом **GenPairs_Dissect**. В приведенной ниже программе **Compare** пользователь указывает количество вершин графа, плотность связей (плотность определяет количество рёбер), и количество генерируемых графов. Программа повторяется вплоть до ввода пользователем некорректных данных.

При вводе количества вершин и плотности графов следует учесть, что положительное значение количества вершин **Nodes** влечёт генерацию графа, содержащего случайное количество вершин в диапазоне от **Nodes/2** до **Nodes**. Отрицательное значение даёт генерацию графа с **Nodes** вершин точно.

То же касается плотности **Density**. Максимальное значение плотности составляет 100% (или -100%), что соответствует количеству рёбер, равному:

Nodes•(Nodes-1)/2

Испытания, проведенные на десятках тысяч случайных графов разной размерности (до 25 вершин) и плотности (до 100%) не выявили различий между результатами, возвращаемыми функциями **GenPairs_Dissect** и **GenPairs_Edmonds**, что даёт основание предполагать корректность последней.

```
program Compare;
{
  Проверка корректности генерации паросочетаний алгоритмом Эдмондса
  TGraph.GenPairs_Edmonds
  Сравнением его с алгоритмом генерации паросочетаний через разбиения
  TGraph.GenPairs_Dissect
}
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Assembly in '..\..\Common\Assembly.pas',
  Dissect in '..\..\Common\Dissect.pas',
  Graph in '..\..\Common\Graph.pas',
  GrChars in '..\..\Common\GrChars.pas',
  Items in '..\..\Common\Items.pas',
  Root in '..\..\Common\Root.pas',
  SetList in '..\..\Common\SetList.pas',
  SetUtils in '..\..\Common\SetUtils.pas';

var Nodes: integer;      // Количество вершин графа
    Density: integer;    // Плотность связей, %
    Retry: integer;      // Количество графов (повторений)
    i: integer;          // Счётчик цикла

    Graph: TGraph;       // Проверяемый граф
    Pairs: TCostSet;     // Множество пар
    Error: boolean;      // Признак ошибки

    Cost_Dissect: integer;  // Стоимость паросочетания через разбиения
    Cost_Edmonds: integer;  // Стоимость паросочетания Эдмондсом
    Mode: TPairs; {integer;}// Вариант паросочетания

begin
  try
    repeat
      Writeln;
      // Задаём количество вершин графа (+, -)
      Write('Nodes      (>4) = '); Readln(Nodes);
      if Abs(Nodes)<5 then Break;
      // Задаём плотность связей, %  (+, -)
      Write('Density,% (>0) = '); Readln(Density);
      if Abs(Density)<1 then Break;
      // Задаём количество повторений
      Write('Retry      (>0) = '); Readln(Retry);
      if Retry<=0 then Break;
      // Цикл по количеству повторений
      for i := 1 to Retry do begin
        Write('.');
        // Генерируем случайный граф
        Graph:= TGraphChars.GenRandom(
                false,  // неориентированный граф
                1,      // вершины нагружены
                99,     // рёбра нагружены, предельный вес = 99
                Nodes,  // количество вершин, не менее
                Density // плотность связей, не менее
                );
        // - - - - - - - - - - - - - - - - - - - - -
        // Сравниваем в варианте максимальной мощности
        Mode:= pMaxN;
        Pairs:= Graph.GenPairs_Edmonds(Mode);
        Cost_Edmonds:= Pairs.mCost;
```

```
        Pairs.Free;
        Pairs:= Graph.GenPairs_Dissect(Mode);
        Cost_Dissect:= Pairs.mCost;
        Pairs.Free;
        Error:= Cost_Edmonds <> Cost_Dissect;
        if Error then Break;
        // - - - - - - - - - - - - - - - - - - - -
        // Сравниваем в варианте максимального веса
        Mode:= pMaxW;
        Pairs:= Graph.GenPairs_Edmonds(Mode);
        Cost_Edmonds:= Pairs.mCost;
        Pairs.Free;
        Pairs:= Graph.GenPairs_Dissect(Mode);
        Cost_Dissect:= Pairs.mCost;
        Pairs.Free;
        Error:= Cost_Edmonds <> Cost_Dissect;
        if Error then Break;
        // - - - - - - - - - - - - - - - - - - - -
        // Сравниваем в варианте минимального веса
        Mode:= pMinW;
        Pairs:= Graph.GenPairs_Edmonds(Mode);
        Cost_Edmonds:= Pairs.mCost;
        Pairs.Free;
        Pairs:= Graph.GenPairs_Dissect(Mode);
        Cost_Dissect:= Pairs.mCost;
        Pairs.Free;
        Error:= Cost_Edmonds <> Cost_Dissect;
        if Error then Break;
        // Удаление графа
        Graph.Free;
      end;
    until Error;
    if Error then begin
      Graph.Save('Graph.txt');
      Write('### Error! Press Enter...');
      Readln;
    end else begin
      Writeln('OK!');
    end;
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
end.
```

Следующая ниже программа **Compare_Time** сравнивает скорости работы двух методов на большом количестве случайных графов. Параметры графов указываются так, как сказано выше. Ввиду высокого быстродействия метода Эдмондса, для отсечки достаточно большого для измерения временного промежутка, метод вызывается многократно (**CPack= 1000**).

```
program Compare_Time;
{
  Сравнение временных характеристик двух методов поиска паросочетаний
  - алгоритмом Эдмондса TGraph.GenPairs_Edmonds
  - алгоритмом поиска через разбиения TGraph.GenPairs_Dissect
}
{$APPTYPE CONSOLE}

uses
  SysUtils, DateUtils, Math,
  Assembly in '..\..\Common\Assembly.pas',
  Dissect in '..\..\Common\Dissect.pas',
  Graph in '..\..\Common\Graph.pas',
  GrChars in '..\..\Common\GrChars.pas',
  Items in '..\..\Common\Items.pas',
  Root in '..\..\Common\Root.pas',
  SetList in '..\..\Common\SetList.pas',
  SetUtils in '..\..\Common\SetUtils.pas';


const CMode = pMaxW;  // Ищем паросочетание максимального веса
      CPack= 1000;    // Пачка повторений для GenPairs_Edmonds

var
  Nodes: integer;     // Количество вершин графа
  Density: integer;   // Плотность связей, %
  Retry: integer;     // Количество графов (повторений)
  Time_Dissect: Extended; // Общее время для GenPairs_Dissect
  Time_Edmons : Extended; // Общее время для GenPairs_Edmonds
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вывод результатов в файл (на консоль)

procedure ResultExpo(const aFile: String);
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln;
  Writeln('----------------------------------------------------');
  Writeln('Nodes=', Nodes:4, ' Retry=', Retry:4, ' Dens=', Density,' %');
  Writeln('Dissect= ', Time_Dissect/Retry:7:1,' ms');
  Writeln('Edmons = ', Time_Edmons/Retry:7:3,' ms');
  Writeln('Dissect/Edmons= ', Time_Dissect/Time_Edmons:7:0,
          '  Lg =',Log10(Time_Dissect/Time_Edmons):5:2);
  Writeln('----------------------------------------------------');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var
  Graph: TGraph;      // Проверяемый граф
  Pairs: TCostSet;    // Множество пар
  Start: TDateTime;   // Засечка времени
  Time : integer;     // Время исполнения
  k, i: integer;      // счётчики

begin
  try
  repeat
```

```
    // Задаём количество вершин графа (+, -)
    Write('Nodes      (>4) = '); Readln(Nodes);
    if Abs(Nodes)<5 then Break;
    // Задаём плотность связей, %
    Write('Density,% (>0) = '); Readln(Density);
    if Abs(Density)<1 then Break;
    // Задаём количество испытаний
    Write('Retry      (>0) = '); Readln(Retry);
    if Retry<=0 then Break;

    Time_Dissect:= 0; Time_Edmons:= 0;
    for i:= 1 to Retry do begin
      Write('.');
      // Генерируем случайный граф
      Graph:= TGraphChars.GenRandom(
              false,  // неориентированный граф
              1,      // вершины нагружены
              99,     // рёбра нагружены, предельный вес = 99
              Nodes,  // количество вершин, не менее
              Density // плотность связей, % не менее
              );
      // Тестирование метода GenPairs_Edmonds
      Start:= Now; // Засекаем начало работы
      // Выполняем пачку вызовов
      for k:=1 to CPack do begin
        Pairs:= Graph.GenPairs_Edmonds(CMode);
        Pairs.Free; // Удаление паросочетания
      end;
      // Время исполнения пачки
      Time:= MilliSecondsBetween(Start, Now);
      // Общее время для GenPairs_Edmonds
      Time_Edmons:= Time_Edmons + Time/CPack;

      // Тестирование метода GenPairs_Dissect
      Start:= Now; // Засекаем начало работы
      Pairs:= Graph.GenPairs_Dissect(CMode);
      Pairs.Free;  // Удаление паросочетания
      // Время исполнения
      Time:= MilliSecondsBetween(Start, Now);
      // Общее время для GenPairs_Dissect
      Time_Dissect:= Time_Dissect + Time;

      Graph.Free;  // Удаление графа
    end;  // for  i:= 1 to Retry
    ResultExpo('');
    ResultExpo('Compare_Time.txt');
  until false;
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
end.
```

Ввиду низкого быстродействия экспоненциально сложного метода **GenPairs_Dissect**, представленная выше программа даёт возможность сравнения времён лишь для небольших графов. Следующая ниже программа **Edmonds_Time** оценивает скорость работы метода Эдмондса на гораздо более крупных и насыщенных графах.

```
program Edmonds_Time;
{
  Определение временных характеристик метода поиска паросочетаний
  алгоритмом Эдмондса TGraph.GenPairs_Edmonds
}

{$APPTYPE CONSOLE}

uses
  SysUtils, DateUtils, Math,
  Assembly in '..\..\Common\Assembly.pas',
  Dissect in '..\..\Common\Dissect.pas',
  Graph in '..\..\Common\Graph.pas',
  GrChars in '..\..\Common\GrChars.pas',
  Items in '..\..\Common\Items.pas',
  Root in '..\..\Common\Root.pas',
  SetList in '..\..\Common\SetList.pas',
  SetUtils in '..\..\Common\SetUtils.pas';

const CMode = pMaxW;  // Ищем паросочетание максимального веса

var
  Nodes: integer;      // Количество вершин графа
  Density: integer;    // Плотность связей, %
  Retry: integer;      // Количество графов (повторений)
  Time_Edmons : Extended; // Общее время для GenPairs_Edmonds
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вывод результатов в файл (на консоль)

procedure ResultExpo(const aFile: String);
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln;
  Writeln('----------------------------------------------------');
  Writeln('Nodes=', Nodes:4, ' Retry=', Retry:4, ' Dens=', Density,' %');
  Writeln('Edmons = ', Time_Edmons/Retry:7:3,' ms');
  Writeln('----------------------------------------------------');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var
  Graph: TGraph;       // Проверяемый граф
  Pairs: TCostSet;     // Множество пар
  Start: TDateTime;    // Засечка времени
  Time : integer;      // Время исполнения
  Pack : integer;      // Пачка повторений для GenPairs_Edmonds
  k, i: integer;       // счётчики

begin
  try
  repeat
    // Задаём количество вершин графа (+, -)
    Write('Nodes      (>5) = '); Readln(Nodes);
    if Abs(Nodes)<5 then Break;
    // Задаём плотность связей, %
    Write('Density,% (>0) = '); Readln(Density);
```

```
    if Abs(Density)<1 then Break;
    // Задаём количество испытаний
    Write('Retry      (>0) = '); Readln(Retry);
    if Retry<=0 then Break;

    // Вычисляем размер пачки для повторных вызовов
    Pack:= 1+(100000 div abs(Nodes)) div abs(Density);

    Time_Edmons:= 0;
    for i:= 1 to Retry do begin
      Write('.');
      // Генерируем случайный граф
      Graph:= TGraphChars.GenRandom(
              false,  // неориентированный граф
              1,      // вершины нагружены
              99,     // рёбра нагружены, предельный вес = 99
              Nodes,  // количество вершин, не менее
              Density // плотность связей, % не менее
              );
      // Тестирование метода GenPairs_Edmonds
      Start:= Now; // Засекаем начало работы
      // Выполняем пачку вызовов
      for k:=1 to Pack do begin
        Pairs:= Graph.GenPairs_Edmonds(CMode);
        Pairs.Free; // Удаление паросочетания
      end;
      // Время исполнения пачки
      Time:= MilliSecondsBetween(Start, Now);
      // Общее время для GenPairs_Edmonds
      Time_Edmons:= Time_Edmons + Time/Pack;

      Graph.Free;  // Удаление графа
    end;  // for  i:= 1 to Retry
    ResultExpo('');
    ResultExpo('Edmonds_Time.txt');
  until false;
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
end.
```

29.12. Результаты испытаний

Сравним результаты работы двух алгоритмов представленными выше программами (Табл. 29-1). Здесь представлено время обработки графов с разным количеством вершин и плотностью связей 20%. Для каждой размерности было сгенерировано по 50 случайных графов. Вычисления выполнялись на процессоре AMD Ryzen 5 2600 Six-Core Processor 3.40 GHz. Поскольку физическое время вычислений зависит от конкретной модели процессора, метод разбиений и метод Эдмондса сравнивались по соотношению **D/E**, оно дано в этой же таблице, в том числе в логарифмическом виде.

Табл. 29-1 — Сравнение быстродействия двух методов

Количество вершин	Время исполнения, ms		Соотношение времён D/E	Lg(D/E)
	Методом разбиений D	Алгоритмом Эдмондса E		
25	34	0,077	448	2,65
27	66	0,089	740	2,87
29	160	0,112	1431	3,16
31	505	0,154	3279	3,52
33	1 322	0,189	6998	3,84
35	5 637	0,275	20528	4,31
37	16 294	0,346	47104	4,67

По данным из этой таблицы построены графики, представленные на Рис. 29-40 и Рис. 29-41.Уже на графах небольшой размерности метод Эдмондса работает на несколько порядков быстрее, и это преимущество растёт экспоненциально с ростом размерности графа.

Рис. 29-40 — Соотношение времён в зависимости от количества вершин

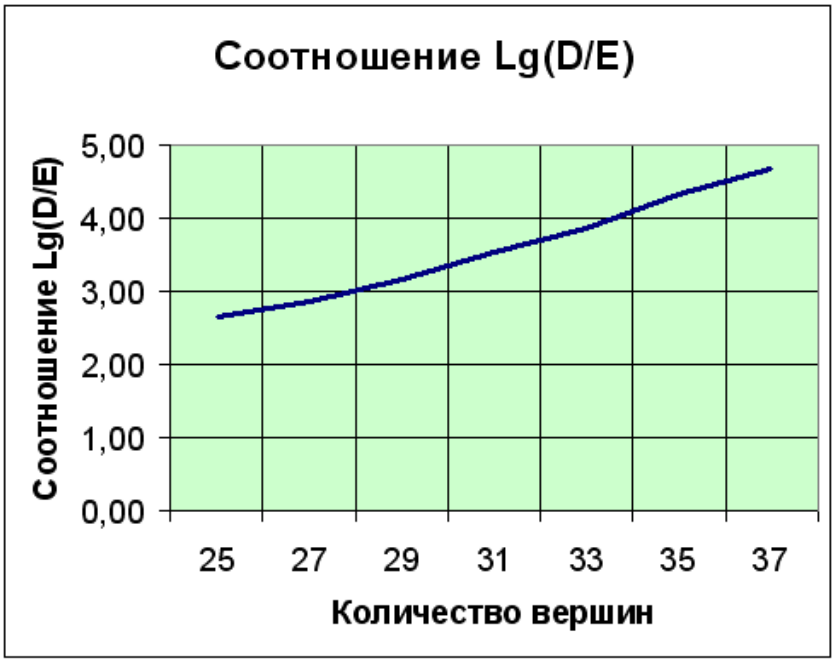


Рис. 29-41 — Соотношение времён D/E в логарифмическом масштабе

Следующая таблица содержит время вычислений методом Эдмондса на графах большой размерности, не доступной методу разбиений. Для каждой размерности было сгенерировано по 50 случайных графов.

Табл. 29-2 — Время вычислений методом Эдмондса, ms

Количество вершин	Плотность связей				
	10%	20%	30%	40%	50%
20	0,043	0,094	0,128	0,125	0,171
30	0,125	0,297	0,368	0,452	0,417
40	0,382	0,663	0,833	0,926	1,045
50	0,840	1,250	1,624	1,921	1,942
60	1,578	2,192	2,287	2,600	3,016
70	2,466	3,647	3,940	3,981	4,334
80	3,940	4,865	5,645	5,487	6,503
90	5,662	7,315	7,993	6,972	9,261
100	8,244	9,556	9,827	11,976	10,880

Продожение таблицы, ms

Количество вершин	Плотность связей				
	60%	70%	80%	90%	100%
20	0,177	0,176	0,213	0,197	0,214
30	0,529	0,567	0,600	0,607	0,640
40	0,941	1,208	1,240	1,220	1,210
50	1,993	1,951	2,141	2,138	2,180
60	3,590	3,520	3,255	3,751	3,851
70	4,851	5,512	5,003	5,382	6,303
80	6,695	7,542	7,580	7,984	7,957
90	10,595	10,172	10,448	10,364	11,980
100	12,595	13,560	13,470	13,924	13,655

По этим данным построены графики, показанные на Рис. 29-42 и Рис. 29-43. Первый из них демонстрирует зависимость времени вычислений от размерности графов при разных плотностях связей. Есть основания полагать, что эта зависимость является полиномом со степенью между 2 и 3: для разреженных графов она ближе к квадратичной, а для насыщенных — к кубической.

На втором графике показана зависимость времени вычислений от плотности связей при разных размерностях. Поначалу эта зависимость близка к линейной вплоть до 50%, но по мере насыщения графа рёбрами сходит на нет.

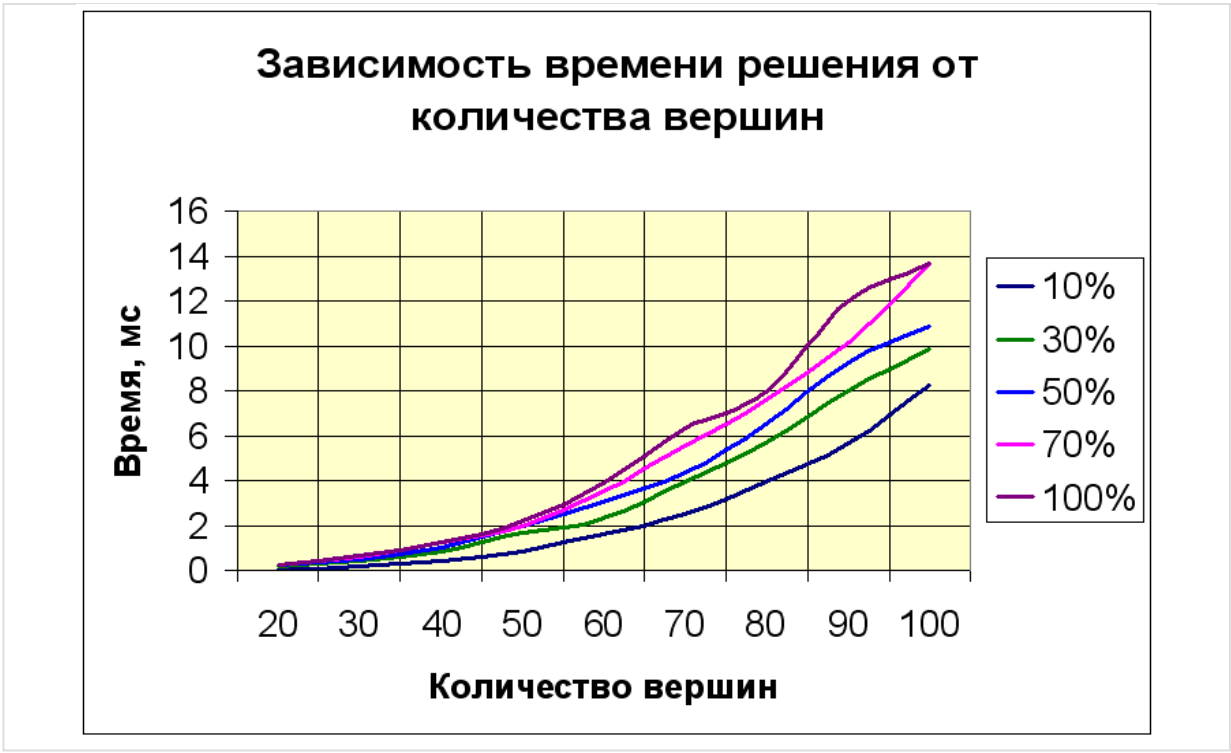


Рис. 29-42 — Зависимость времени решения методом Эдмондса от количества вершин при разных плотностях связей (полином)

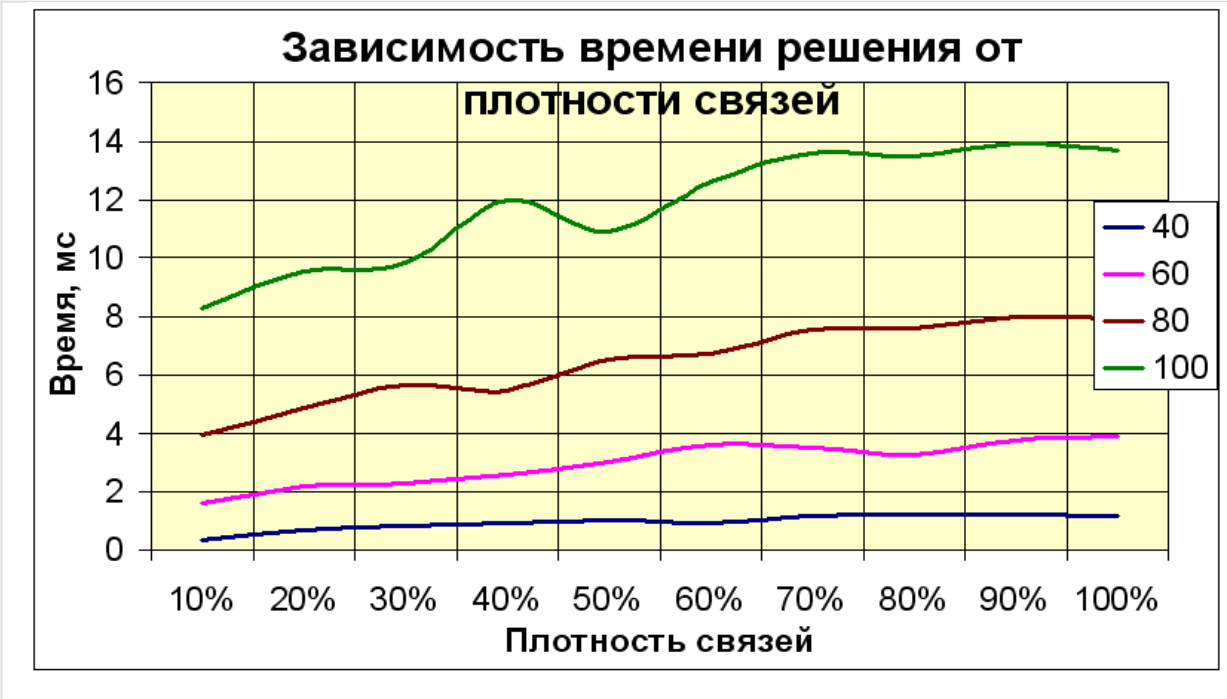


Рис. 29-43 — Зависимость времени решения методом Эдмондса от плотности связей при разном количестве вершин

29.13. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 405
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 178
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 30
Задача почтальона на неориентированном графе

Что связывает математика Леонарда Эйлера с бедным китайским почтальоном? Вы не поверите — мосты города Кёнигсберга. Гуляя по ним, Эйлер задался вопросом: можно ли пройти по всем мостам города и вернуться в исходную точку, пройдя мосты ровно по одному разу? В переводе на язык графов это звучит так: можно ли, начав движение из любой вершины графа пройти по всем его рёбрам ровно по одному разу и вернуться в исходную вершину? Эйлер показал, что для этого все вершины графа должны иметь *чётную* степень. Такой граф называется *чётным*. Действительно, если сжигать пройденные мосты, то после каждого посещения промежуточной вершины она будет лишаться двух своих рёбер: одного при заходе в вершину, другого — после выхода из неё. Если же у промежуточной вершины (с нечётной степенью) останется не пройденным одно ребро, то, зайдя по нему в эту вершину, покинуть её и вернуться в исходную вершину не получится.

А что китайский почтальон? Он разносит письма по всем улицам города и возвращается в исходную точку, пытаясь при этом покрыть минимальное расстояние. Если бы в графе, отражающем структуру его города, все вершины имели бы чётные степени, то почтальон прошёлся бы по каждой улице ровно по одному разу. Однако если часть вершин будет нечётной, то некоторые улицы придётся пройти более одного раза, и тогда встаёт задача минимизации этого излишнего пути.

Рассмотрим решение задачи Эйлера — найдём цикл, в котором каждое ребро встречается единожды. Затем покажем, как сводится к циклу Эйлера задача почтальона.

30.1. Формирование цикла Эйлера

Будем оперировать со *связным* неориентированным графом, все вершины которого имеют *чётные* степени (рис. 30-1), то есть с *чётным* графом.

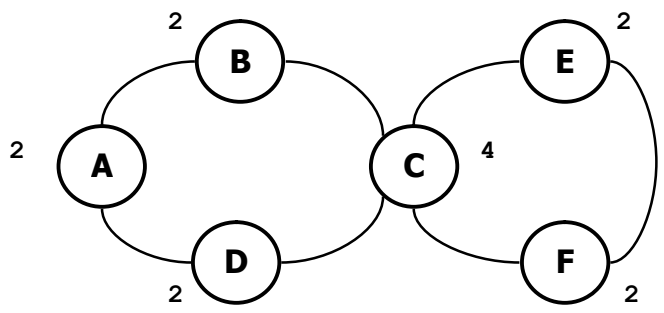


Рис. 30-1 — Связный неориентированный чётный граф

Поскольку результат не зависит от стартовой вершины, начнём постройку цикла с вершины *A*. Припишем каждой вершине число, равное степени этой вершины (степень — это количеству инцидентных вершине рёбер). Здесь у всех вершин, кроме вершины *C*, степени равны двум. При обходе рёбер будем уменьшать это число на единицу при каждом входе в вершину и при каждом выходе из неё (дважды). Обходим граф в глубину, двигаясь только по ещё не пройденным рёбрам вплоть до исходной вершины.

Из вершины *A* попадём в *B*, а затем неминуемо в вершину *C*. Здесь имеем альтернативу в виде вершин *D* или *E*. Если двинуться сначала в *E*, то затем неминуемо попадём в *F* и снова в *C*. Здесь альтернативы нет: в этой точке все «мосты», кроме *C-D*, «сожжены», и потому далее через вершину *D* возвращаемся в исходную вершину *A*. Посещены ли все «мосты»? Пройдя вдоль цикла, убеждаемся, что все вершинные числа обнулились, стало быть, задача решена, и в результате получен цикл:

```
A-> B-> C-> E-> F-> C-> D-> A
```

Если же из вершины *C* направиться сначала в вершину *D*, то достигнем исходной вершины *A* по короткому циклу:

```
A-> B-> C-> D-> A
```

Степени вершин, состоящих в этом цикле, уменьшились на два, и обнулились у всех вершин, кроме вершины *C*, где степень уменьшилась с 4 до 2. Обнаружив это, вновь построим цикл, начиная с вершины *C* и двигаясь по ещё целым «мостам». Этот второй цикл будет вставлен в разрез первого между вершинами **C->D**. В итоге получим результат, совпадающий с первым:

```
A-> B-> C->                D-> A    - 1-й цикл

           E-> F-> C->              - 2-й цикл

A-> B-> C-> E-> F-> C-> D-> A    - результат
```

Таким образом, *эйлеров цикл* строится последовательным прохождением циклов, стартующих из вершин с ненулевыми степенями. Эти частичные циклы вставляются в результирующий цикл, который первоначально пуст. Отметим, что каждая вершина посещается **P/2** раз, где **P** — степень вершины.

30.2. Приведение задачи почтальона к задаче Эйлера

Итак, построить Эйлеров цикл в графе с *чётными* степенями вершин не составляет труда. Однако китайский почтальон, напомню, курсирует по графу с вершинами *нечётных* степеней, и потому неминуемо проходит некоторые улицы повторно. Будем искать эти улицы, а также одолеваемое расстояние из условия его минимизации. Путь к решению лежит через модификацию графа с целью

превращения его в *чётный* граф. С этой целью рассмотрим пару простых закономерностей.

Первая закономерность состоит в том, что *количество нечётных вершин в графе чётно*. Возьмём чётный граф, все вершины которого чётны. Если добавить в граф ребро, оно сделает нечётными сразу две вершины: эту и смежную. Если в полученном нечётном графе добавить ребро к нечётной вершине, оно сделает её чётной, а смежную ей — либо чётной (если она нечётная), либо нечётной (если она чётная). Потому количество нечётных вершин либо уменьшится на два, либо останется прежним. Если в том же нечётном графе добавить ребро к чётной вершине, оно сделает её нечётной, а смежную ей — либо чётной (если она нечётная), либо нечётной (если она чётная). Здесь количество нечётных вершин либо увеличится на два, либо не изменится. Как бы то ни было, количество нечётных вершин в графе остаётся чётным.

Пусть при обходе улиц почтальон посетил все вершины графа некоторое число раз, уменьшая каждый раз степени вершин на два. В конце концов, степени нечётных вершин уменьшатся до единицы. Выйдя из некоторой нечётной вершины, и «сжёгши» последний инцидентный ей «мост», будет естественным последовать в другую нечётную вершину, поскольку количество *нечётных* вершин в графе *чётно*. Так, в конце концов, почтальон посетит все нечётные вершины (и чётные, разумеется, тоже). Стало быть, для сведения задачи почтальона к задаче Эйлера, достаточно связать некоторые пары нечётных вершин дополнительными рёбрами, превратив их в чётные. Вопрос лишь в том, как внести при этом минимальное приращение к длине пути.

Вспомним об алгоритме Флойда и Данцига, находящем кратчайшие пути между всеми вершинами графа, он обсуждался в главе 15. Пусть мы нашли все нечётные вершины, а также расстояния между ними. Отсюда можно построить *полный* вспомогательный граф, где все нечётные вершины соединяются рёбрами, длина которых равна длине кратчайшего пути между вершинами. Паросочетание *минимального* веса в этом графе даст именно те пары нечётных вершин, которые надо соединить дополнительными рёбрами (о *паросочетаниях* сказано в главе 29). Так найдём минимальную длину пути, преодолеваемого почтальоном, а заменив дополнительные рёбра эквивалентными цепочками исходных рёбер, сформируем и сам цикл.

Но есть проблема: в простых графах кратные рёбра не допустимы, то есть, нельзя соединять пару вершин более чем одним ребром, а это необходимо. Осилим затруднение так: назначим каждому ребру исходного графа число, назовём его *степенью* ребра, или счётчиком *кратности*. Изначально всем рёбрам назначим единичную степень. Найдя оптимальные пути между нечётными вершинами, нарастим степени рёбер вдоль этих путей, тем самым некоторые рёбра удвоятся, утроятся, и т.д. При последующем прохождении циклов каждое посещение ребра будет уменьшать его степень на единицу вплоть до нуля, после чего ребро станет недоступно.

К слову сказать, после такого увеличения степеней рёбер, степени всех вершин графа станут *чётными*, если подсчитывать их с учётом степеней рёбер. Отсюда появляется возможность построения Эйлерова цикла.

Рассмотрим изложенную последовательность действий на примере графа, текстовое представление которого дано ниже:

```
Граф с четырьмя вершинами нечётной степени
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A B C D E F G H
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

Граф показан на рис. 30-2, он имеет 4 нечётные вершины, выделенные серым (степени вершин указаны вместе с их обозначением). Здесь нетрудно определить кратчайшие пути между нечётными вершинами (табл. 30-1).

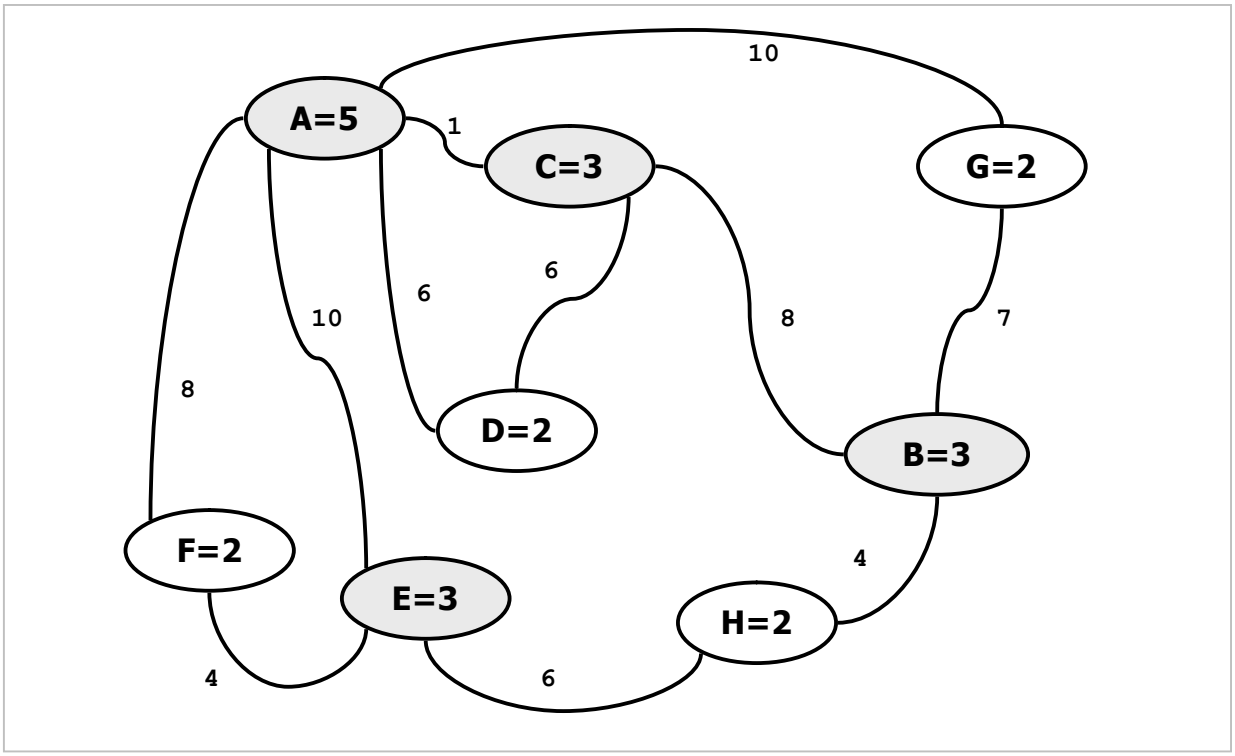


Рис. 30-2 — Взвешенный граф с четырьмя нечётными вершинами

Табл. 30-1 — Кратчайшие пути и расстояния между нечётными вершинами

Нечетные вершины	A	B	C	E
A	-	A-C-B = 9	A-C = 1	A-E = 10
B	A-C-B = 9	-	B-C = 8	B-H-E = 10
C	A-C = 1	B-C = 8	-	C-A-E = 11
E	A-E = 10	B-H-E = 10	C-A-E = 11	-

По этим данным строится вспомогательный полный граф, в нём находится паросочетание минимального веса (рис. 30-3).

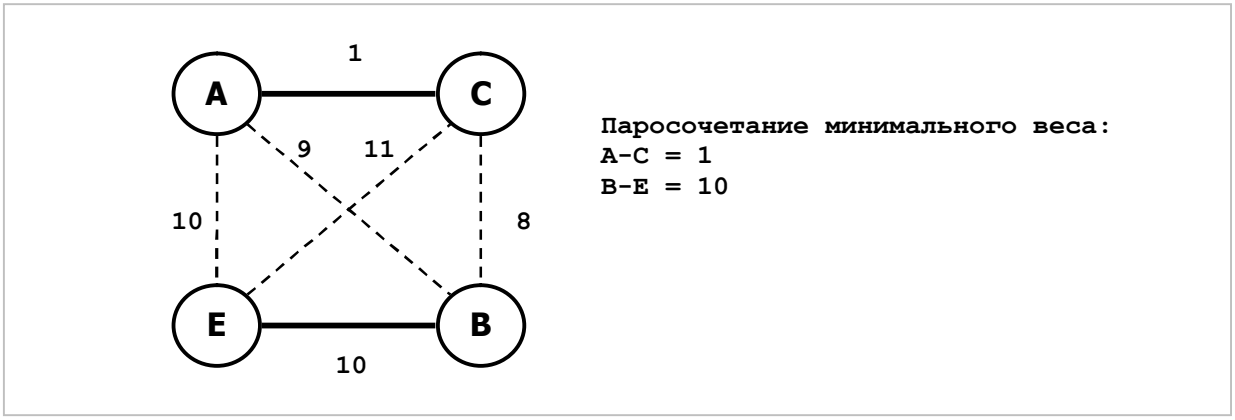


Рис. 30-3 — Вспомогательный полный граф
и паросочетание минимального веса

Здесь необходимо увеличить степени рёбер, лежащих на кратчайших путях между вершинами *A-C* и *B-E*, результат показан на рис. 30-4, где увеличены степени рёбер *A-C*, *B-H* и *H-E*. Теперь степени всех вершин стали чётными, и граф готов для поиска Эйлерова цикла, но уже с учётом степеней рёбер.

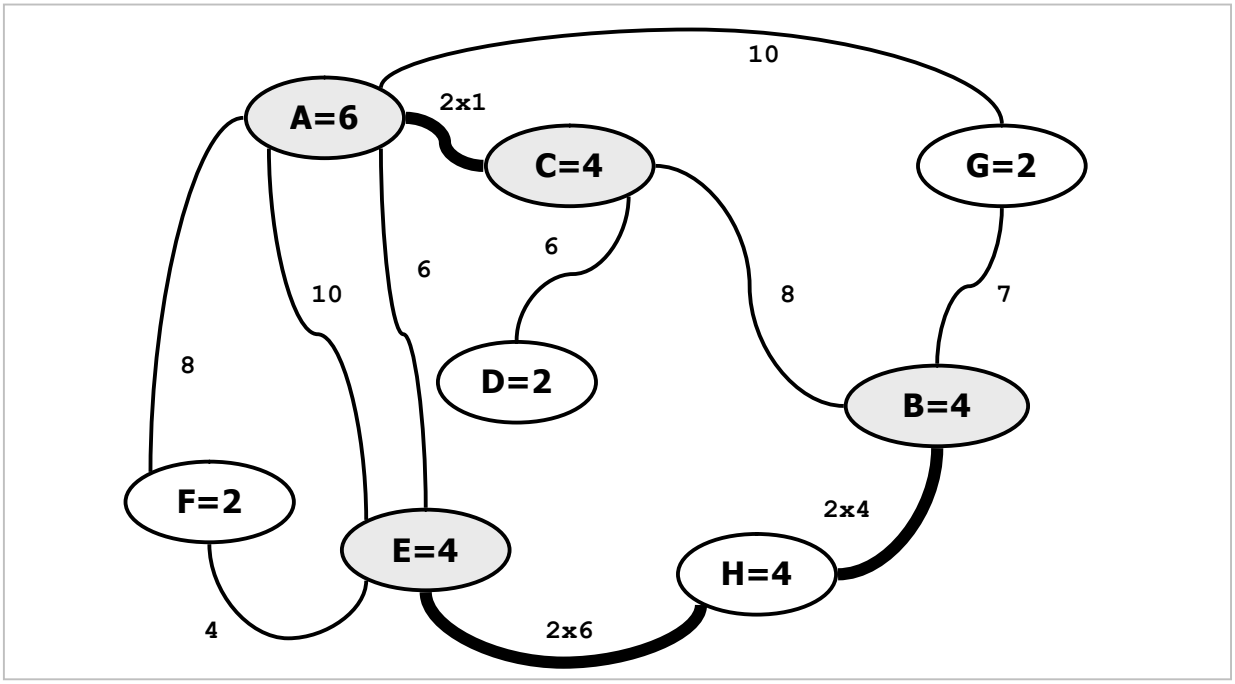


Рис. 30-4 — Граф после увеличения степеней некоторых вершин и рёбер

На рис. 30-5 показан порядок обхода графа из вершины *A*, тут числа рядом со стрелками обозначают последовательность обхода.

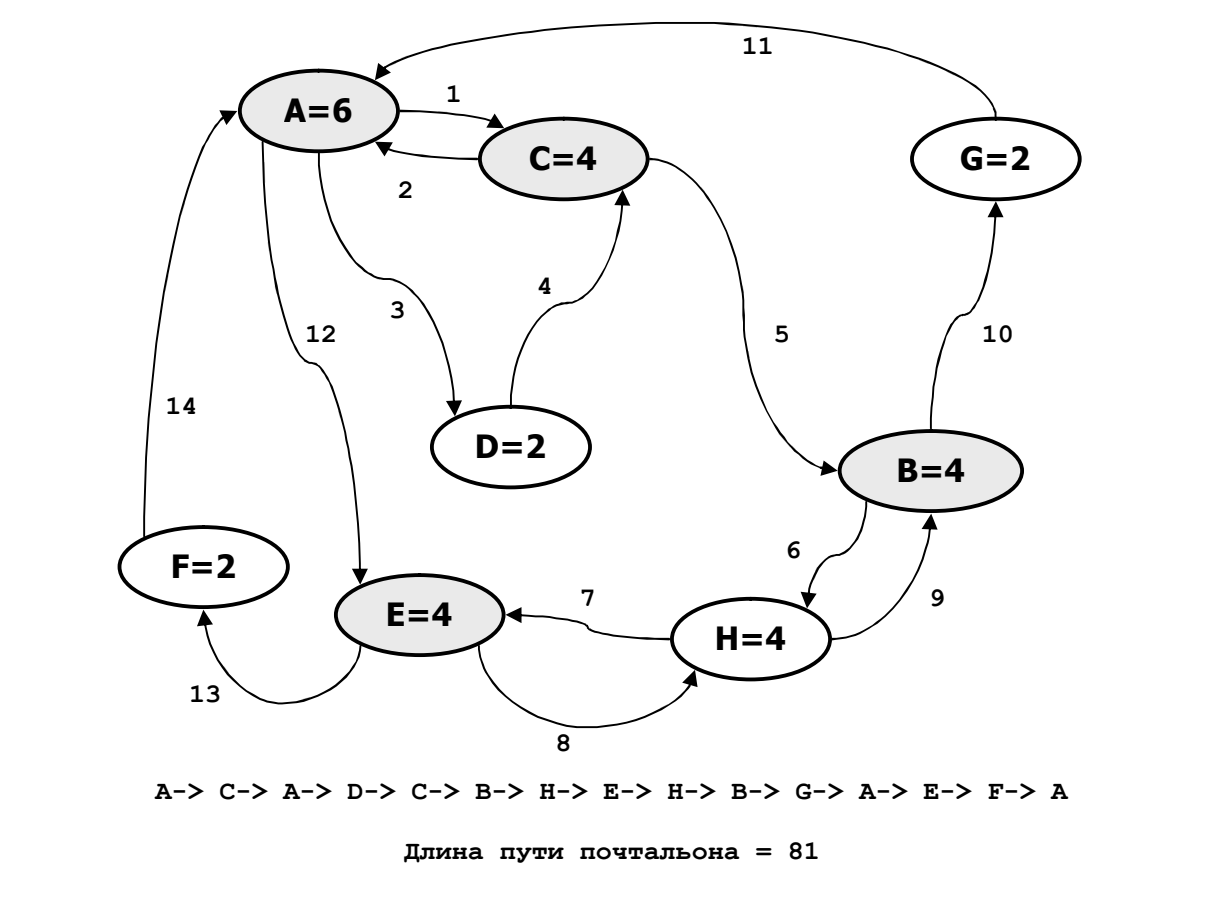


Рис. 30-5 — Порядок обхода рёбер из вершины A (числа указывают порядок)

30.3. Алгоритм

Дадим теперь алгоритм решения задачи почтальона:

- Подсчитать степени всех вершин графа и назначить рёбрам единичные степени.
- Если в графе существуют нечётные вершины, то:
  - Найти пути и расстояния между всеми нечётными вершинами.
  - Построить вспомогательный полный граф, состоящий только из нечётных вершин и рёбер, представляющих кратчайшие пути; найти в этом графе паросочетание минимального веса.
  - Нарастить степени рёбер исходного графа, лежащих на кратчайших путях между вершинами оптимальных пар, а также нарастить степени инцидентных им вершин.
- Построить Эйлеров цикл с учётом степеней рёбер (их кратности).

Алгоритм реализован в методе **GenPostPathUndir**, текст которого дан ниже. Здесь для хранения текущих степеней вершин и рёбер отведены поля **TNode.mLimit** и **TLink.mLimit**.

Листинг 30-1 — Метод решения задачи почтальона на неориентированном графе

```
function TGraph.GenPostPathUndir(var aCost: integer): TBuffer;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Изменение степени ребра (двух полей TLink.mLimit)

  procedure ModifyLimit(aLink: TLink; aDelta: integer);
  var L: TLink; // встречный линк
  begin
    Inc(aLink.mLimit, aDelta);              // прямой линк
    L:= aLink.mDest.GetLink(aLink.mOwner);  // встречный линк
    Inc(L.mLimit, aDelta);
  end;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества вершин с нечётной степенью
  // и предварительная подготовка полей TNode.mLimit

  function GenOddNodes: TSet;
  var Node: TNode;
  begin
    Result:= nil;
    // Перебор всех вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Сброс:  mColor= 0;  mPred= mLink= nil;  mDist= mFlow= MaxInt;
      Node.ResetNode;
      // В поле mLimit сохраним количество линков вершины (степень):
      Node.mLimit:= Node.OutGetCnt;  //Node.mLnkOut.GetCount;
      // Если количество рёбер нечётное, заносим в множество результата:
      if Odd(Node.mLimit) then begin
        if not Assigned(Result) then Result:= CreateSet;
        Result.Insert(Node);
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Установка степеней рёбер в полях TLink.mLimit
  // и степеней вершин в полях TNode.mLimit

  procedure LimitsPrepare;
  var OddNodes: TSet;     // множество вершин с нечётной степенью
        Pairs : TCostSet; // паросочетание минимального веса
         Pair : TPair;    // пара вершин
           Gr : TGraph;   // вспомогательный граф
      Ni, Nj  : TNode;    // вершины исходного графа
      N2i,N2j : TNode;    // копии вершин исходного графа во вспомогательном
         NFin : TNode;    // конечная вершина в кратчайшем пути
         Dist : integer;  // кратчайшее расстояние между вершинами
         Link : TLink;
         i, j : integer;
```

```
begin
  // Устанавливаем степени всех рёбер в единицу:
  SetLinksLimit(1);
  // Собираем множество вершин с нечётными степенями:
  OddNodes:= GenOddNodes;
  // Если таких вершин нет, то выход:
  if not Assigned(OddNodes) then Exit;
  // Создаём вспомогательный граф:
  Gr:= TGraph.Create('Odd Nodes:', // имя графа произвольно
                     false,        // граф не ориентирован
                     false,        // вершины не нагружены
                     true          // рёбра нагружены
                    );
  // И вставляем в него копии вершин с нечётными степенями:
  Ni:= OddNodes.GetFirst as TNode;
  while Assigned(Ni) do begin
    Gr.InsertNode(Ni.Copy as TNode);
    Ni:= OddNodes.GetNext as TNode;
  end;
  // Для определения взаимных расстояний строим карту дальних указателей
  // в исходном графе:
  Self.InitMap_Floyd;
  // Строим рёбра между всеми вершинами вспомогательного графа
  // (создаём полный граф).
  // Длина рёбер устанавливается равной длине кратчайшего пути
  // между вершинами исходного графа
  for i:= 1 to OddNodes.GetCount-1 do begin
    Ni:= OddNodes.GetItem(i) as TNode;     // вершина из множества нечётных
    N2i:= Gr.mNodes.GetItem(i) as TNode;   // её копия в графе
    // Перебор последующих после Ni вершин:
    for j:= i+1 to OddNodes.GetCount do begin
      Nj:= OddNodes.GetItem(j) as TNode;   // вершина из множества нечётных
      N2j:= Gr.mNodes.GetItem(j) as TNode; // её копия во вспомогат. графе
      Dist:= Ni.GetFarLink(Nj).mDist; // кратчайшее расстояние в исх. графе
      Gr.SetLink(N2i, N2j, Dist);     // назначим длине ребра во вспомогат.
    end;
  end;
  // Во вспомогательном графе
  // находим паросочетание с минимальным весом:
  Pairs:= Gr.GenPairs_Edmonds (pMinW);

  // Для каждой из полученных пар увеличиваем степени mLimit
  // вдоль соответствующих кратчайших путей:
  Pair:= Pairs.mSet.GetFirst as TPair;
  while Assigned(Pair) do begin
    // Pair.mSet содержит ровно две вершины пары.
    // Стартовая вершина цепи в исходном графе:
    Ni:= mNodes.GetObject(Pair.mSet.GetFirst as TNode) as TNode;
    // Конечная вершина цепи в исходном графе:
    NFin:= mNodes.GetObject(Pair.mSet.GetNext as TNode) as TNode;
    repeat
      Inc(Ni.mLimit);          // степень+1 в начальной (промежут.) вершине
      Nj:= Ni.GetNear(NFin);   // следующая вершина на кратчайшем пути
      Link:= Ni.GetLink(Nj);   // линк на промежуточную вершину
      ModifyLimit(Link, +1);   // степень+1 линка
      Ni:= Nj;                 // продвижение к следующей вершине
    until Ni = NFin;           // пока не достигнута конечная
    Inc(Ni.mLimit);            // степень+1 в конечной вершине
    Pair:= Pairs.mSet.GetNext as TPair;  // следующая пара
  end;
  // Очистка памяти:
  Pairs.ClrAndDestroy;     // пары
```

```
    Pairs.Free;              // множество пар
    DoneMap;                 // освобождаем карту дальних связей
    Gr.Free;                 // вспомогательный граф
    OddNodes.Free;           // множество вершин с нечётной степенью
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление кольцевых маршрутов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк
      // (степень которого Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл и путь:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (степень больше нуля):
      Result.Put(Link);         // помещаем линк в буфер результата
      Dec(aNode.mLimit);        // уменьшаем степень текущей вершины
      aNode:= Link.mDest;       // продвигаемся к следующей вершине
      Dec(aNode.mLimit);        // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue); // накопление стоимости (длины пути)
      ModifyLimit(Link, -1);    // уменьшаем степень ребра
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;
    Link: TLink;
    Count: integer; // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathUndir }

  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= 0;                // накопитель стоимости
  LimitsPrepare;            // подготовка полей mLimit (пределов посещ.)
  Node:= NodeFirst;         // построение начинаем с первой вершины

  // Строим кольцевые маршруты,
  // пока не исчерпаны степени всех вершин и рёбер
  repeat
    // Строим кольцевой маршрут из вершины Node и добавляем в результат:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;        // счётчик для прокрутки буфера Result
    while Count>0 do begin
      Link:= Result.Get as TLink;   // берём линк из начала буфера
      Result.Put(Link);             // и помещаем в конец
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то стоп.
      // Построение очередного цикла начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
```

```
    // Выход из цикла, когда степени всех вершин и рёбер исчерпаны:
  until not Assigned(Node);
end;

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);        // да, возвращаем назад в буфер
      Break;                    // и прекращаем цикл
    end;
    Result.Put(Link);           // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;
```

Поясним некоторые технические моменты. Процедура **ModifyLimit** изменяет степень ребра, модифицируя поле **mLimit** сразу и в прямом, и в обратном линках ребра.

Процедура **LimitsPrepare** выполняет работу, предваряющую поиск циклов, — превращает нечётный граф (если он таков) в чётный. Здесь подсчитываются степени вершин, и, если надо, строится вспомогательный граф, после чего находится минимальное паросочетание и корректируются степени рёбер. Отметим, что операторы:

```
      Ni:= mNodes.GetObject(Pair.mSet.GetFirst as TNode) as TNode;

      NFin:= mNodes.GetObject(Pair.mSet.GetNext as TNode) as TNode;
```

находят вершины в исходном графе, соответствующие их копиям во вспомогательном графе.

Процедура **AddPath** добавляет к буферу результата **Result** очередной цикл, начинающийся из вершины **aNode**. Объединение циклов происходит в теле метода, причём очередная часть цикла вставляется в нужное место прокруткой буфера **Result**.

30.4. Испытание

Следующей программой испытаем метод, решающий задачу почтальона в неориентированном графе.

Листинг 30-2 — Программа для решения задачи почтальона на неориентированном графе

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
```

```
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res : TBuffer;
    Cost: integer;

begin
    Gr:= TGraphChars.Load('Test.txt');
    Gr.Expo;
    Res:= Gr.GenPostPathUndir(Cost);
    Res.Expo;
    Res.Free;
    Writeln('Cost= ', Cost);
    Readln;
end.
```

На рисунке представлен граф для проверки метода.

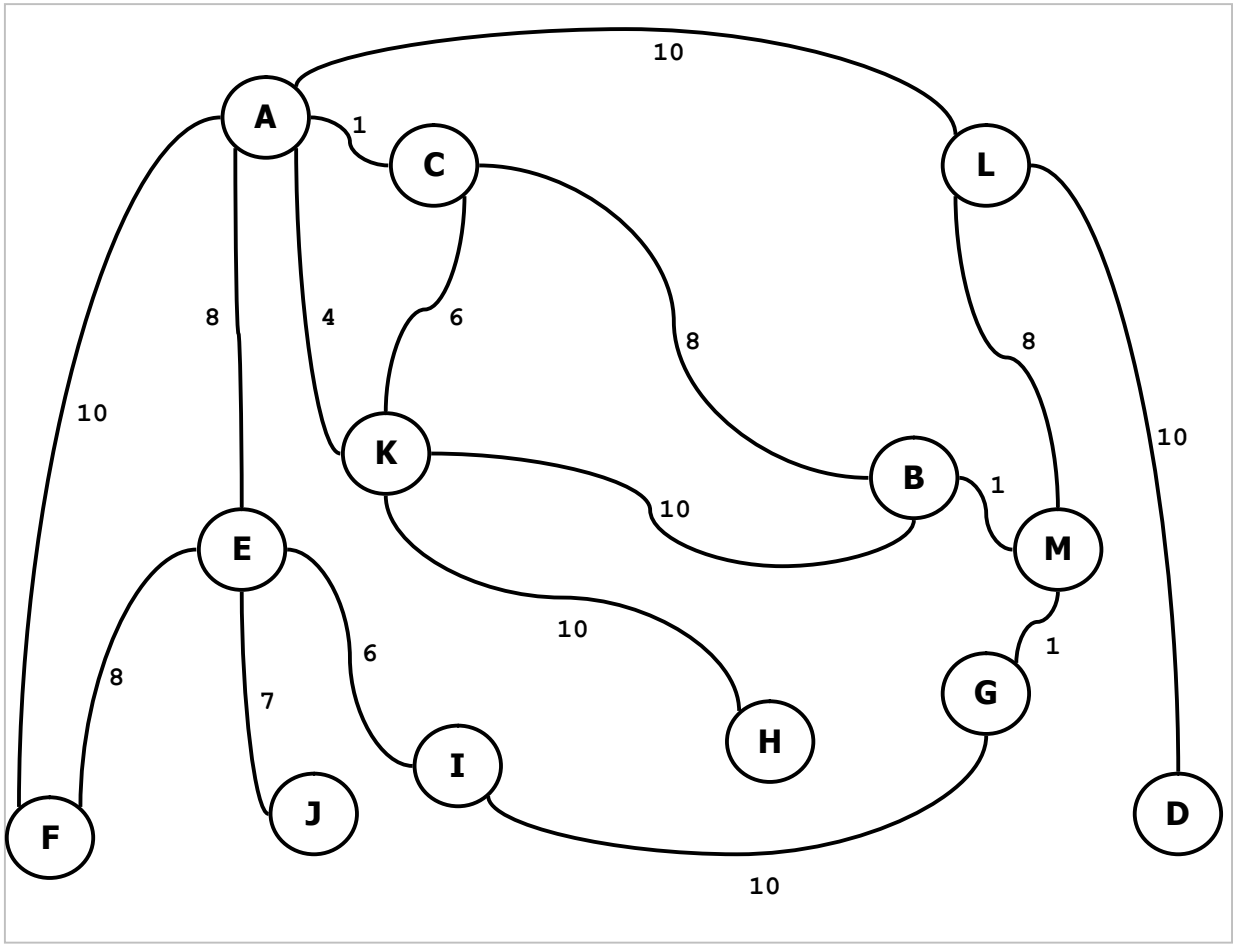


Рис. 30-6 — Граф для проверки метода, решающего задачу почтальона

Текстовое представление графа дано ниже:

```
Граф для проверки метода решения задачи почтальона
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
13 - количество вершин
A B C D E F G H I J K L M
A -> C=1 E=8 F=10 K=4 L=10
B -> C=8 K=10 M=1
C -> A=1 B=8 K=6
D -> L=10
E -> A=8 F=8 I=6 J=7
F -> A=10 E=8
G -> I=10 M=1
H -> K=10
I -> E=6 G=10
J -> E=7
K -> A=4 B=10 C=6 H=10
L -> A=10 D=10 M=8
M -> B=1 G=1 L=8
```

В результате получен цикл стоимостью 159 единиц, содержащий 25 рёбер:

```
A-> C-> A-> E-> J-> E-> A-> F-> E-> I-> G-> M-> B-> C-> K-> H-> K-> A-> K->
B-> M-> L-> D-> L-> A
```

30.5. Итоги

- В задачах Эйлера и почтальона на неориентированных графах ищутся (кратчайшие) циклы обхода всех рёбер графа.
- В задаче Эйлера надо пройти по всем рёбрам графа ровно один раз. Она имеет решение только для чётных связных графов, то есть таких, в которых все вершины имеют чётные степени.
- Задача почтальона решается на произвольных связных графах, она состоит в поиске кратчайшего цикла, включающего все рёбра графа, при этом некоторые рёбра в цикле могут повторяться.
- Решение задачи почтальона сводится к решению задачи Эйлера путём повышения степеней существующих рёбер, и превращения тем самым нечётного графа в чётный.
- Для поиска удваиваемых (утраиваемых, и т.д.) рёбер строится вспомогательный полный граф и отыскивается на нём паросочетание минимального веса.

30.6. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 227
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 219 Стр. 227
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 31
Задача почтальона на орграфе

Продолжим рассказ о китайском почтальоне. Он разбогател, купил автомобиль, и стал развозить на нём почту. Увы, узкие улочки тамошнего городка допускают лишь одностороннее движение. В заботах об экономии топлива почтальон ищет кратчайший кольцевой маршрут (цикл), проходящий по всем улицам городка хотя бы по разу. Иными словами, решает задачу предыдущей главы на ориентированном графе.

Выясним *необходимое* условие её решения. Очевидно, что граф должен быть *сильно связным*, то есть таким, чтобы все вершины в нём были взаимно достижимы. Будем считать, что оно соблюдается. Подробно связность графа обсуждалась в главе 16.

Самым подходящим для почтальона маршрутом было бы однократное прохождение каждой дуги графа по циклу Эйлера. Но для этого число входов в каждую вершину должно совпадать с числом выходов из неё. Для ориентированного графа это означает, что количество входящих в каждую вершину дуг должно совпадать с количеством исходящих. То есть, должны совпадать полу-степени входа и выхода каждой из вершин. Графы, где это условие соблюдается, называются *симметричными*.

Но почтальону не повезло: орграф, с которым он имел дело, был не симметричным. Здесь по аналогии с решением задачи на неориентированном графе, следует найти множество дуг, которые почтальону придётся проезжать дважды, трижды, и т.д.

31.1. Идея решения

Введём несколько терминов. Назовём *степенью вершины* суммарное количество дуг, входящих и исходящих из вершины, то есть, сумму полустепеней исхода и захода.

Степенью дуги назовём количество разрешённых посещений этой дуги, изначально степени всех дуг составляют единицу. Подобно задаче почтальона на неориентированном графе, в ходе поиска циклов степени вершин и дуг будут уменьшаться.

Назовём *асимметрией* вершины разность между количеством входящих и исходящих дуг. Положительная асимметрия означает, что полу-степень входа превышает полу-степень исхода.

Проведём мысленный эксперимент, и покажем его на следующем орграфе (рис. 31-1). Допустим, что за каждой улицей городка закреплён свой почтальон, а в каждой вершине графа размещено почтовое отделение. Поутру почтальоны

приходят каждый в *своё* отделение, садятся в служебные авто, обслуживают *свои* улицы, затем оставляют автомобили уже в других почтовых отделениях на другом конце улицы, и возвращаются налегке пешком. В сумме все они делают ту же работу, что должен выполнить один почтальон.

Ввиду асимметрии графа, в вершинах с положительной асимметрией автомобилей останется больше, чем было утром. Соответственно в вершинах с отрицательной асимметрией их останется меньше. Чтобы восстановить исходное количество авто в вершинах, специальная ночная бригада перегоняет часть автомобилей из мест скопления туда, где их не хватает, причём делает это *кратчайшими* маршрутами. Эта задача равнозначна поиску *потока минимальной стоимости* из нескольких источников в несколько стоков, где единицами потока являются перегоняемые автомобили. Решив задачу оптимального распределение этого потока, найдём дуги, которые неоднократно посетит почтальон, работающий в одиночку.

Обратимся к рис. 31-1, здесь асимметрия чёрной вершины *A* составляет **4-1=3**, у трёх серых вершин *B*, *C*, и *E* асимметрия равна **-1**, а прочие вершины симметричны.

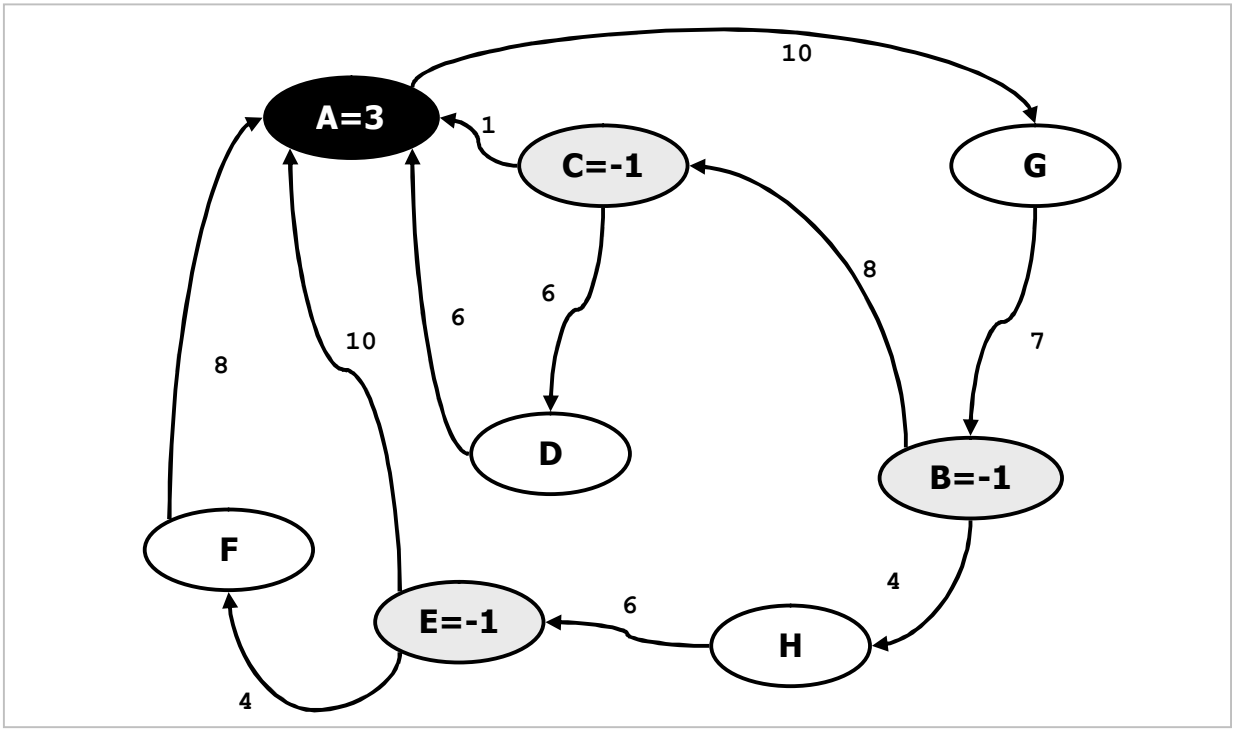


Рис. 31-1 — Орграф с четырьмя асимметричными вершинами

После почтовой смены в чёрной вершине окажется на три автомобиля больше, а в трёх серых будет недоставать по одному авто. Здесь надо переправить поток из чёрной вершины в серые, и общий объём потока составит **3** единицы. Способ решения таких задач уже известен. Введём в граф две вспомогательные вершины: общий исток *S* и общий сток *T*. Соединим общий исток *S* дугами нулевой стоимости со всеми истоками в графе (чёрными вершинами). Этим дугам назначим пропускную способность, равную «мощности» соответствующего истока.

Аналогично поступим со стоками, соединяя их с искусственным стоком T. Всем прочим дугам графа назначим неограниченную пропускную способность, а цену (вес дуги) сохраняем. Так получим граф, показанный на рис. 31-2.

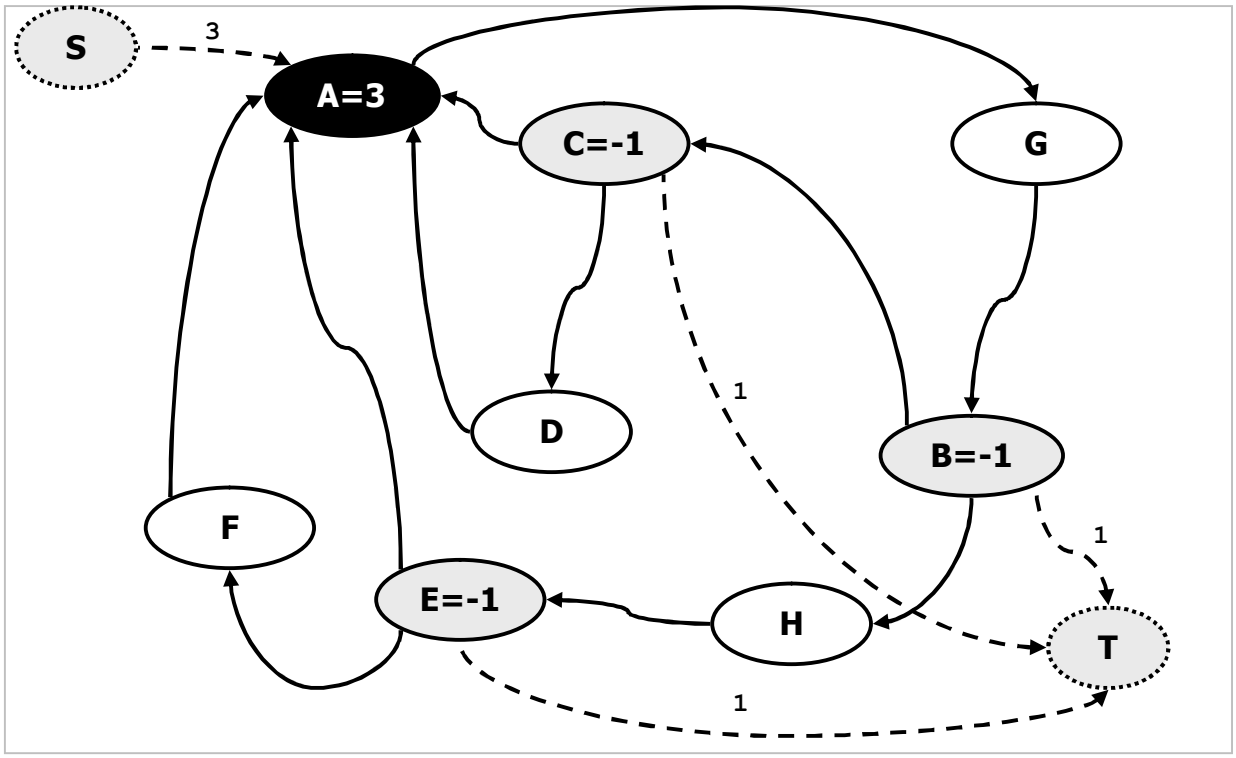


Рис. 31-2 — Дополнение орграфа истоком, стоком, и вспомогательными дугами

Распределение потока от S к T показано на рис. 31-3.

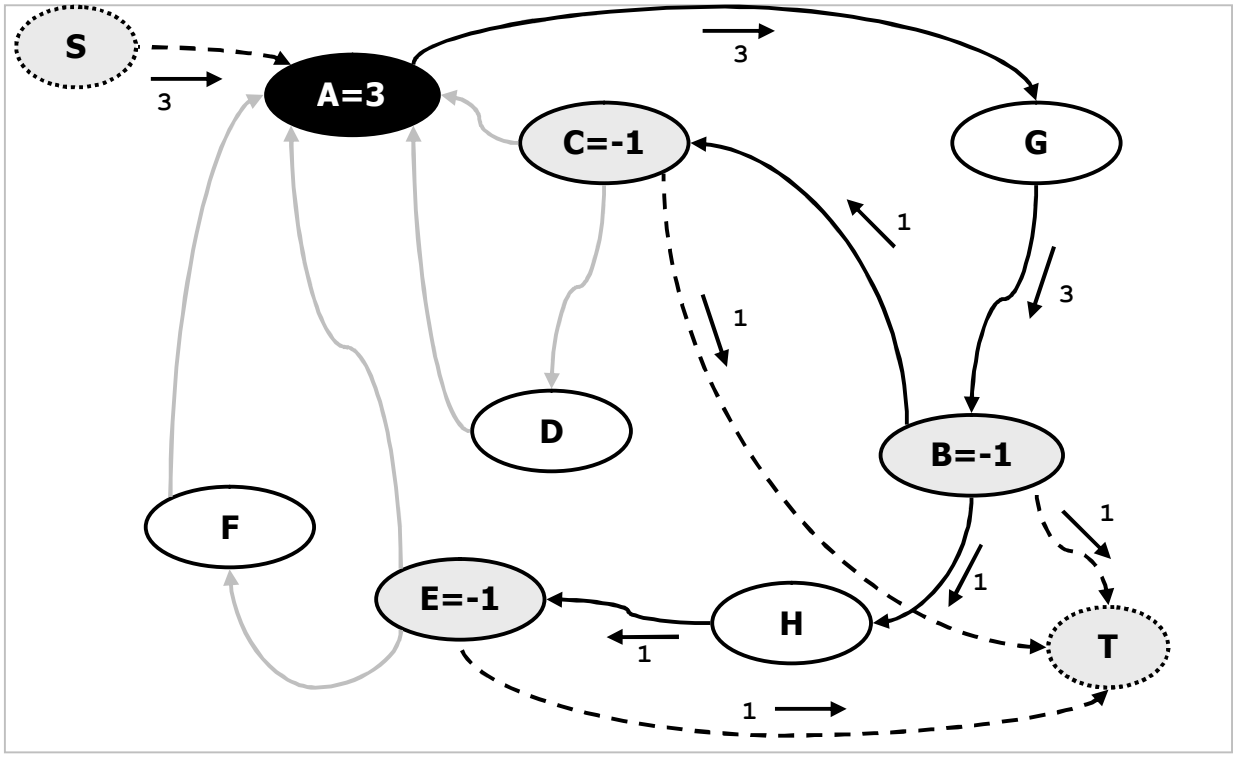
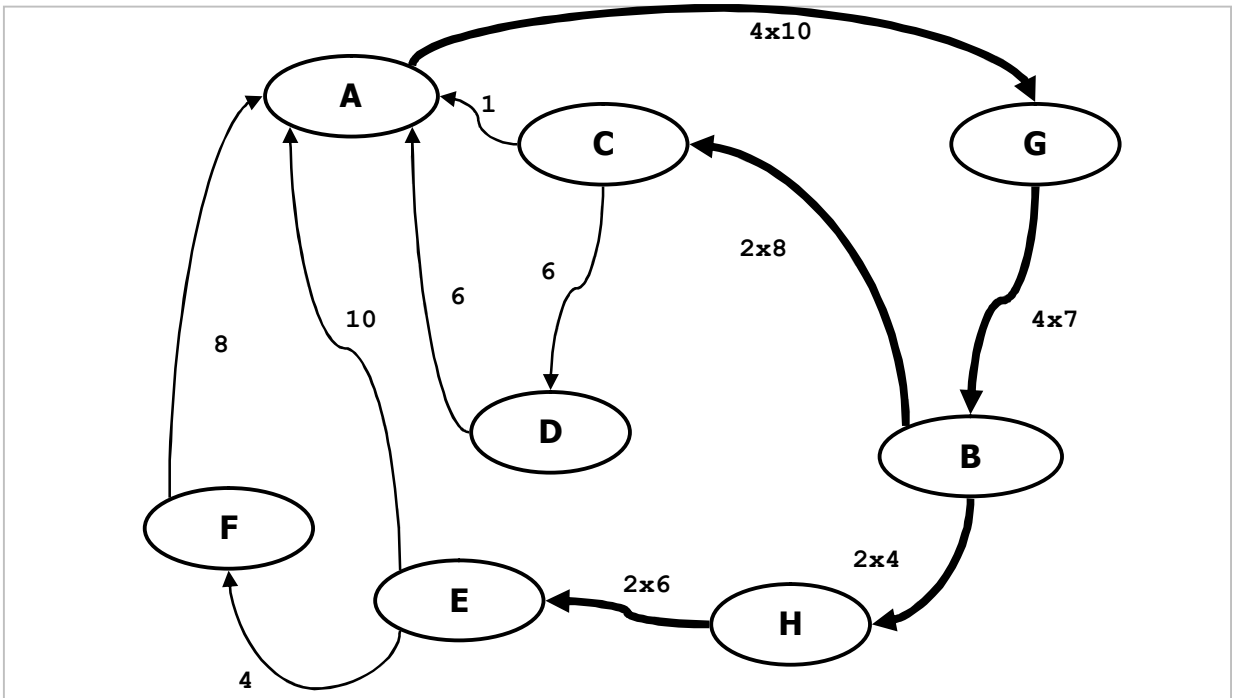


Рис. 31-3 — Распределение потока

Распределение потока показывает, что дуги *A-G* и *G-B* почтальону надо посетить ещё по три раза (всего по 4), а дуги *B-C*, *B-H* и *H-E* — дважды. Увеличив степени всех вершин с учётом найденного потока, и удалив вспомогательные дуги и вершины, получим граф, показанный на рис. 31-4.



A-> G-> B-> C-> A-> G-> B-> C-> D-> A-> G-> B-> H-> E-> A-> G-> B-> H-> E-> F-> A
Стоимость цикла = 139

Рис. 31-4 — Скорректированные степени дуг и кратчайший цикл

Далее на этом графе находим циклы Эйлера так же, как это делалось на неориентированном графе, результат показан на том же рисунке.

31.2. Алгоритм решения задачи почтальона на орграфе

* Вычислить начальные степени и асимметрию всех вершин. Назначить всем дугам единичные степени.
* Если граф оказался несимметричным, то:
  - Создать две вспомогательные вершины: искусственный исток *S* и сток *T*.
  - Соединить исток *S* со всеми положительно асимметричными вершинами, пропускную способность дуг назначить равной асимметрии этих вершин.
  - Соединить отрицательно асимметричные вершины с искусственным стоком *T*, пропускную способность дуг задать равной асимметрии этих вершин по модулю (положительной).
  - Найти распределение по дугам графа известного потока минимальной стоимости от *S* к *T*.

- Прибавить значения потоков в дугах к текущим степеням дуг и соответственно увеличить степени вершин.

* Найти кратчайший цикл Эйлера с учётом полученных значений степеней дуг и вершин.

Текст метода представлен в следующем листинге.

Листинг 31-1 — Метод для задачи почтальона на орграфе

```
function TGraph.GenPostPathDir(var aCost: integer): TBuffer;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подготовка полей mLimit в дугах и вершинах графа.
  // Возвращает стоимость дополнительного потока

  function LimitsPrepare: integer;
  var Node : TNode;     // текущая вершина
      Link : TLink;     // текущая дуга
      Flag : boolean;   // признак наличия асимметричных вершин
      S, T : TNode;     // искусственные исток и сток
      Flow: integer;    // величина корректирующего потока
  begin
    Result:= 0;
    // Устанавливаем пределы посещения всех дуг в единицу:
    SetLinksLimit(1);
    // Для всех вершин определяем разности полустепеней входа и выхода,
    // а также степени вершин TNode.mLimit:
    Flag:= false;         // признак асимметрии графа
    // Перебор вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        mLimit:= InGetCnt + OutGetCnt;  // степень вершины
        mDist:= InGetCnt - OutGetCnt;   // асимметрия
        Flag:= Flag or (mDist<>0); // признак наличия асимметричных вершин
      end;
      Node:= NodeNext;
    end;
    // Если все вершины симметричны (все полустепени совпадают),
    // то выход из процедуры:

    if not Flag then Exit;

    // Для несимметричного графа строим поток минимальной стоимости
    // с величиной, равной суммарной ассиметрии
    // Создаём:
    S:= TNode.Create(0, Self); // вспомогательный исток
    T:= TNode.Create(0, Self); // вспомогательный сток

    Flow:= 0; // здесь подсчитаем суммарный поток
    // Обработка всех вершин, кроме S и T:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Во всех дугах формируем данные для вычисления потока:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mLow:= 0;             // минимальный поток = 0
        Link.mHigh:= MaxInt;       // максимальный не ограничен
        Link:= Node.OutLinkNext;
      end;
      // Если вершина несимметрична (mDist <> 0),
```

```
      // соединяем её с искусственным истоком либо стоком:
      if Node.mDist < 0 then begin
        // создаём дугу из вершины во вспомогательный сток:
        Link:= Node.MakeLink(T, 0); // создаём линк
        Link.mHigh:= -Node.mDist;   // пропускная способность дуги
      end else if Node.mDist > 0 then begin
        // Накапливаем требуемый поток:
        Inc(Flow, Node.mDist);      // асимметрия вершины
        // создаём дугу из вспомогательного истока в вершину:
        Link:= S.MakeLink(Node, 0); // создаём линк
        Link.mHigh:= Node.mDist;    // пропускная способность дуги
      end;
      Node:= NodeNext;
    end; // while
    // Вставляем вспомогательные вершины в граф:
    mNodes.Insert(S);          // исток
    mNodes.Insert(T);          // сток
    // Распределяем поток Flow по дугам и подсчитываем его стоимость:
    Result:= CalcMinCostFlow(S, T, Flow);
    // Удаляем из графа и освобождаем:
    RemoveNode(S);  S.Free;     // вспомогательный исток
    RemoveNode(T);  T.Free;     // вспомогательный сток

    // Добавляем полученные потоки в дугах к степеням дуг и вершин
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Flow:= Link.mFlow;  // поток в дуге
        // При наличии потока
        // добавляем его к степеням дуги и двух смежных вершин:
        if Flow > 0 then begin
          Inc(Link.mLimit, Flow);          // + к дуге
          Inc(Link.mOwner.mLimit, Flow);   // + к источнику дуги
          Inc(Link.mDest.mLimit, Flow);    // + к приёмнику дуги
        end;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление частичных циклов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк (Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (лимит больше нуля):
      Result.Put(Link);          // помещаем линк в буфер результата
      Dec(aNode.mLimit);         // уменьшаем степень текущей вершины
      aNode:= Link.mDest;        // продвигаемся к следующей вершине
      Dec(aNode.mLimit);         // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue);   // накопление стоимости (длины)
      Dec(Link.mLimit);          // уменьшаем степень дуги
    end;
```

```
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
var Node: TNode;      // текущая вершина
    Link: TLink;      // текущая дуга
    Count: integer;   // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathDir }
  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= -1;     // Стоимость цепи на случай отсутствия решения

  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;

  aCost:= 0;           // накопитель стоимости
  LimitsPrepare;       // подготовка степеней mLimit вершин и рёбер
  Node:= NodeFirst;    // построение начинаем с любой вершины

  // Цикл добавления циклов к буферу Result:
  repeat
    // Строим циклы из текущей вершины Node
    // и добавляем в буфер Result:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;         // счётчик для прокрутки буфера
    while Count>0 do begin
      Link:= Result.Get as TLink;    // берём линк из начала буфера
      Result.Put(Link);              // и помещаем в конец
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то
      // построение очередного цикла начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
    // Выход, если степени всех вершин исчерпаны
  until not Assigned(Node);

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);         // да, возвращаем назад в буфер
      Break;                     // и прекращаем цикл
    end;
    Result.Put(Link);            // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;
```

Этот метод сходен с методом для неориентированного графа: циклы строятся одинаково. Отличие состоит лишь в предварительной подготовке степеней вершин и дуг.

31.3. Испытание

Для испытания служит следующая программа.

Листинг 31-2 — Программа для решения задачи почтальона на ориентированном графе

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res : TBuffer;
    Cost: integer;

begin
    Gr:= TGraphChars.Load('Test.txt');
    Gr.Expo;
    Res:= Gr.GenPostPathDir(Cost);
    Res.Expo;
    Res.Free;
    Writeln('Cost= ', Cost);
    Readln;
end.
```

Проверим действие метода на графе, показанном на рис. 31-1, текстовое представление графа дано ниже:

```
Орграф для задачи почтальона
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A B C D E F G H
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

В результате получен цикл длиной **139** единиц:

```
A-> G-> B-> C-> A-> G-> B-> C-> D-> A-> G-> B-> H-> E-> A-> G-> B-> H-> E->
F-> A
```

31.4. Итоги

- В задаче Эйлера надо пройти по всем дугам орграфа ровно один раз; она имеет решение только на симметричных сильно связных орграфах.
- Задача почтальона решается на произвольных связных орграфах, она состоит в поиске кратчайшего цикла, включающего все дуги, при этом некоторые дуги в цикле могут повторяться.
- Задача почтальона сводится к задаче Эйлера путём повышения степеней существующих дуг, — так несимметричный орграф превращается в симметричный.
- Для поиска удваиваемых (утраиваемых, и т.д.) дуг строится вспомогательный орграф, в котором отыскивается распределение по дугам потока минимальной стоимости.

31.5. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 227
	8	Липский В.	Комбинаторика для программистов	
√	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 219 Стр. 227
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 32
Разомкнутая задача Гамильтона

Задачи о посещении *всех* вершин графа сочетают простоту постановки и сложность решений. Но обо всём по порядку.

32.1. Цикл и пути коммивояжёра

Отвлечёмся от своих забот и вникнем в бизнес кочующего торговца — коммивояжёра. Он периодически доставляет товары в ряд городов, связанных сетью дорог, причём не все эти города связаны напрямую, иные доступны лишь транзитом через другие города. Источник его товара расположен в одном из городов, и торговцу надо избрать *кратчайший* маршрут, ведущий через *все* города и возвращающий его в исходную точку — такой маршрут назовём *циклом*. Данная проблема известна как *замкнутый* вариант задачи о коммивояжёре.

Пусть теперь источники товара расположены в двух городах: X и Y. Тогда торговец будет следовать из X в Y, посещая по ходу *все* прочие города. Достигнув конечной точки и запасшись товаром, он последует в обратном направлении от Y к X, посещая опять те же самые города. Поиск кратчайшего пути между двумя городами с посещением *всех* остальных городов известен как *разомкнутый* вариант задачи о коммивояжёре.

А где лучше расположить эти источники товара? То есть, выбрать города X и Y так, чтобы путь между ними через все прочие города оказался минимальным? — таково дополнение к *разомкнутому* варианту задачи о коммивояжёре.

32.2. Цикл и пути Гамильтона

К сожалению, не все торговцы кристально честны, иным лучше повторно не появляться там, где продан их товар. То же относится и к страховым агентам. Однако не всякая сеть наземных дорог позволяет избежать нежелательных встреч: посещение *всех* городов может потребовать *повторного* визита в некоторые из них. И потому из гуманных соображений переместим место действия в страну, где из любого города можно добраться до любого другого, минуя прочие города. Однако в этом полном насыщенном связями графе, прямое ребро или дуга между вершинами не всегда короче суммы нескольких промежуточных дуг, то есть, правило треугольника здесь не гарантируем.

Применительно к этому *насыщенному* графу введём несколько терминов, а также рассмотрим ещё три варианта задач о коммивояжёре, известных как задачи Гамильтона (Гамильтон — не торговец, а известный математик).

Цикл Гамильтона — это замкнутый путь, проходящий через *все* вершины графа ровно по *одному* разу. Нас будет интересовать *кратчайший* цикл.

Путь Гамильтона — это кратчайший путь, проходящий между двумя заданными вершинами с посещением всех остальных ровно по *одному* разу.

Дополнение к задаче о пути Гамильтона — поиск такой *пары вершин*, Гамильтонов путь между которыми будет кратчайшим среди всех возможных пар.

Иногда в задачах коммивояжёра и Гамильтона ищут не *кратчайшие*, а *длиннейшие* пути. В таких случаях задачу сводят к поиску опять же кратчайших путей, инвертируя длины дуг и рёбер. Поэтому далее всегда будем искать *кратчайшие* пути.

32.3. Решаемость задач

Итак, задачи *коммивояжёра* от задач *Гамильтона* отличает лишь то, что коммивояжёру *разрешено* повторное посещение вершин, а Гамильтону — *нет*. Требования к циклам и путям коммивояжёра мягче, и можно ожидать, что задачи коммивояжёра всегда имеют решения на связном неориентированном графе или сильно связанном орграфе. Для задач Гамильтона критерии решаемости сложнее, но мы не станем вникать в эту тему. Во-первых, потому, что нас будут интересовать главным образом полные графы, а на таких графах задачи Гамильтона всегда решаются. А во-вторых, при отсутствии решения на неполном графе, алгоритмы обнаружат это естественным путём.

На рис. 32-1 показан сильно связанный орграф, содержащий только цикл коммивояжёра. Очевидно, что при обходе всех вершин графа здесь не миновать повторного посещения вершин *B* и *C*.

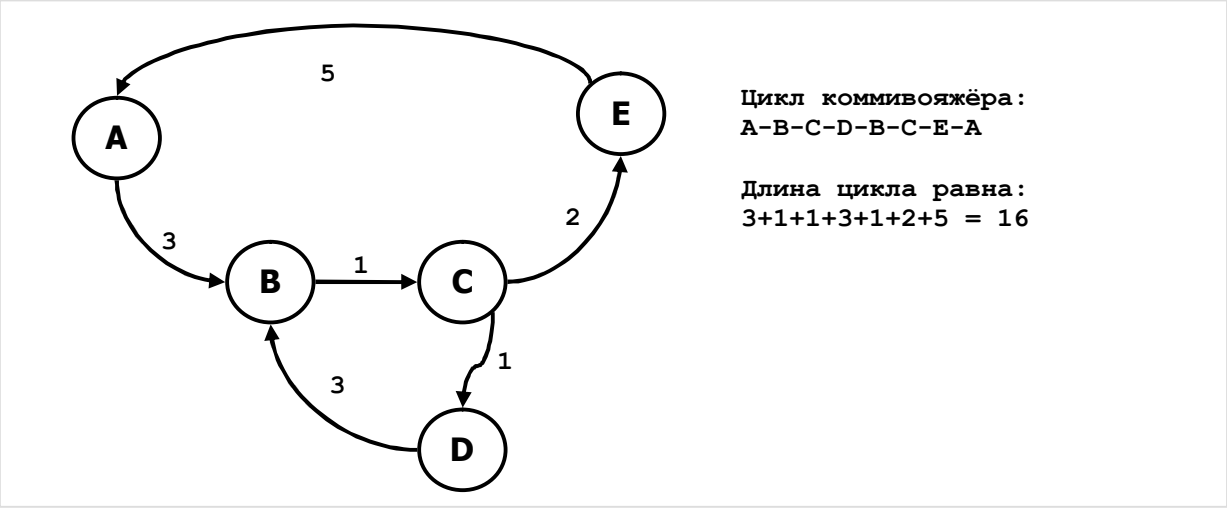
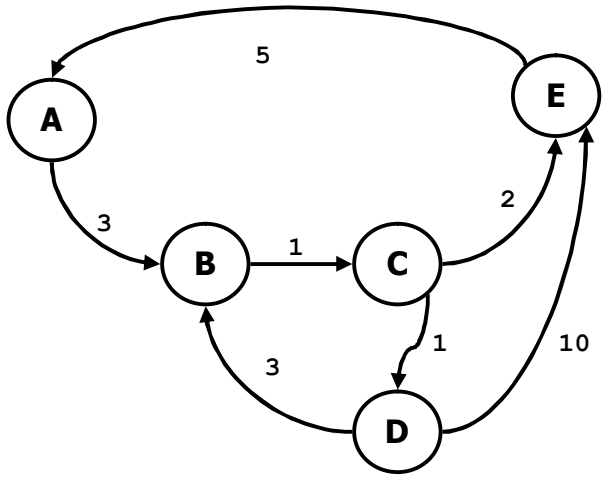


Рис. 32-1— Сильно связанный орграф, не содержащий цикл Гамильтона

На рис. 32-2 показан похожий граф, но с добавленной дугой *D-E*, что позволило провести в нём ещё и цикл Гамильтона (ни одна промежуточная вершина не посещена дважды). Здесь Гамильтонов цикл оказался длиннее кратчайшего цикла коммивояжёра, и потому честному торговцу выгодней повторно посетить города *B* и *C*. Стало быть, если в графе существует

Гамильтонов цикл или путь, то существует и цикл (путь) *коммивояжёра*, причём второй не длиннее первого.



Кратчайший цикл коммивояжёра:
A-B-C-D-B-C-E-A

Длина цикла равна:
3+1+1+3+1+2+5 = 16

Цикл Гамильтона:
A-B-C-D-E-A

Длина цикла равна:
3+1+1+10+5 = 20

Рис. 32-2 — Орграф, содержащий и цикл коммивояжёра, и цикл Гамильтона

32.4. Связь между задачами *коммивояжёра* и *Гамильтона*

Математика щедра на парадоксы! Один из них состоит в том, что решать задачи *коммивояжёра* будем через... решение *Гамильтоновых* задач! Позвольте, но ведь сказано, что путь Гамильтона существует не во всяком графе! Дело в том, что *связанный* граф, на котором решается задача коммивояжёра, можно свести к эквивалентному *полному* графу, соединив все его вершины виртуальными рёбрами или дугами. Каждой такой мнимой дуге или ребру соответствует *цепь* из дуг или рёбер исходного графа, соединяющая инцидентные мнимому ребру вершины кратчайшим путём. Решив на таком полном графе задачу Гамильтона, решение для коммивояжёра получим обратной заменой мнимых дуг (рёбер) цепочками реальных дуг (рёбер).

Так, к примеру, отсутствующую дугу *D-E* на рис. 32-1 можно заменить виртуальной дугой *D-E* такой, что длина её составит **3+1+2=6**, что эквивалентно цепочке дуг *D-B-C-E*. Связав попарно все вершины графа, породим *полный* граф, на котором гарантировано решается Гамильтонова задача.

Упомянутые тут несложные замены будут реализованы в главе 34, где дано решение задач коммивояжёра. А сейчас устремим усилия на решение двух вариантов задачи *Гамильтона* : *разомкнутой* (Гамильтонов путь), и *замкнутой* (Гамильтонов цикл). При внешнем сходстве формулировок, методы решения этих задач различны, и потому в этой главе уделим внимание только *разомкнутой* задаче, а Гамильтоновым *циклом* займёмся в следующей главе.

Итак, формулировка задачи о кратчайшем *Гамильтоновом пути* такова: дан граф и две его вершины, необходимо найти *кратчайший* путь из одной вершины в другую, проходящий через *все* прочие вершины графа ровно по одному разу.

32.5. Ориентиры

Отметим, что в равной степени и *быстрого*, и *точного* решения этой задачи не найдено, и потому рассмотрим несколько алгоритмов, в разной степени отвечающих этим двум требованиям. Оттолкнёмся от простых решений, которые будут ориентирами, в сравнении с которыми будем оценивать эффективность более изощрённых алгоритмов. Эти три простых алгоритма таковы: 1) случайный выбор, 2) жадный алгоритм и 3) полный перебор.

32.5.1. Случайный выбор маршрута

Этот абсурдный алгоритм обладает лишь одним достоинством — он очень быстр. Суть его соответствует названию: на пути от исходной вершины к конечной будем следовать «куда глаза глядят», переходя по первой попавшейся дуге в ещё не посещённую вершину (посещённые вершины метим). Очень вероятно, что при испытании на большом количестве графов средняя длина маршрута будет близка к средней длине дуги графа, умноженной на количество дуг. Результаты, полученные случайным выбором, будут основой для оценки других алгоритмов. Реализацию случайного выбора объединим в одном методе с *жадным* алгоритмом.

32.5.2. Жадный алгоритм

В жадном алгоритме выбирается не произвольная, а *кратчайшая* дуга к ещё не посещённой вершине. Скорость алгоритма лишь немногим уступает случайному выбору, но результат существенно лучше (числовые сравнения даны в конце главы). Ниже следует листинг метода, реализующего, и случайный выбор, и жадный алгоритм, — вариант выбирается через параметр **aGreed**.

Листинг 32-1 — Случайный выбор и жадный алгоритм

```
function TGraph.GenHamPath_Greed(aStart, aFin: TNode;
                                 var aCost: integer;
                                 aGreed: Boolean // TRUE - жадный алгоритм
                                 ): TBuffer;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение первого встретившегося белого линка (случайного)
  function GetAnyLink(aNode: TNode): TLink;
  begin
    Result:= aNode.OutLinkFirst;
    while Assigned(Result) and
          (Result.mDest.mColor <> CWhite)
      do Result:= aNode.OutLinkNext;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение ближайшего белого линка
  function GetBestLink(aNode: TNode): TLink;
  var  Link: TLink;
       BestCost: integer;
  begin
    Result:= nil;
    BestCost:= MaxInt;
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
```

```
      if (Link.mDest.mColor = CWhite) and
         (Link.mValue < BestCost) then begin
        BestCost:= Link.mValue;
        Result:= Link;
      end;
      Link:= aNode.OutLinkNext;
    end; // while
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    Cost: integer;    // текущая стоимость цепи
    Count: integer;   // счётчик неокрашенных вершин
    Node: TNode;      // текущая вершина
    L : TLink;        // текущий линк

begin { GenHamPath_Greed }
  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;               // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  ResetNodes;            // очистка вспомогательных полей
  aFin.mColor:= CBlack; // красим конечную вершину
  Cost:= 0;              // накопленная стоимость = 0

  Count:= mNodes.GetCount-2; // счётчик неокрашенных вершин
  Node:= aStart;
  while Count > 0 do begin
    Node.mColor:= CBlack;
    Result.Put(Node);
    // перебираем соседние неокрашенные вершины
    if aGreed
      then L:= GetBestLink(Node) // ищем ближайшую неокрашенную
      else L:= GetAnyLink(Node); // ищем любую неокрашенную
    // Если неокрашенных нет, прервать цикл
    if not Assigned(L) then break;
    // Нашли неокрашенную:
    Inc(Cost, L.mValue);     // накапливаем стоимость
    Dec(Count);              // счётчик - 1
    Node:= L.mDest;          // следующая вершина
  end; // while

  if Count=0 then begin
    // Здесь пройдены все вершины, кроме последней
    Result.Put(Node);        // заносим в результат предпоследнюю
    L:= Node.GetLink(aFin); // линк из предпоследней в последнюю
    if Assigned(L) then begin
      Result.Put(aFin);
      Inc(Cost, L.mValue);  // накапливаем стоимость
      aCost:= Cost;         // возвращаем стоимость
    end else Result.Clear;  // если нет завершающего линка
  end;
end;
```

32.5.3. Полный перебор маршрутов

Хотя жадному алгоритму присущи зачатки интеллекта, однако он, подобно начинающему шахматисту, видит лишь на шаг вперёд и лишён прозорливости, ведущей к точному решению. Найти это решение на первый взгляд не так уж сложно: достаточно перебрать все возможные пути от исходной вершины к конечной. Рекурсивный алгоритм такого перебора представлен ниже, — это

типичный обход графа в глубину, по ходу которого накапливается стоимость очередного пути. Возврат на предыдущий уровень рекурсии случается, когда:

a) достигнута конечная вершина, и путь построен, — здесь запоминаем лучший на текущий момент результат;

b) накопленная стоимость пути превысила лучшую на данный момент, и тогда нет смысла углубляться далее, и следует отступить на уровень выше.

Листинг 32-2 — Поиск минимального пути рекурсивным перебором

```
function TGraph.GenHamPath_Full(aStart, aFin: TNode;
                              var aCost: integer): TBuffer;
var
    Cost: integer;    // текущая стоимость пути
    Count: integer;   // счётчик окрашенных вершин
    Node: TNode;      // текущая вершина
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура сохранения текущего решения.
  // Вызывается после окраски всех вершин.
  // aFin - последняя окрашенная вершина
  procedure SaveResult;
  begin
    aCost:= Cost;          // сохраняем стоимость
    Result.Clear;          // очищаем буфер результата
    Node:= aFin;           // заполняем буфер с последней вершины
    repeat
      Result.Push(Node);   // заносим в буфер в обратном порядке
      Node:= Node.mPred;   // предыдущая вершина
    until Node=aStart;     // пока не достигнем начальной вершины
    Result.Push(Node);     // заносим конечную вершину
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура поиска в глубину

  procedure Local(aNode: TNode);

  var L : TLink;  // текущий исходящий линк
  begin { Local }
    aNode.mColor:= CBlack;  // красим данную вершину
    Inc(Count);             // и наращиваем счётчик окрашенных
    // Все вершины (кроме последней) окрашены?
    if Count < mNodes.GetCount-1 then begin
      // Нет, ищем соседнюю неокрашенную перебором исходящих линков
      L:= aNode.OutLinkFirst;
      while Assigned(L) do begin
        if (L.mDest <> aFin) and (L.mDest.mColor = CWhite) then begin
          // Нашли неокрашенную и не последнюю:
          Inc(Cost, L.mValue);        // накапливаем стоимость
          if Cost < aCost then begin  // если стоимость ниже минимальной
            L.mDest.mPred:= aNode;    // то метим предыдущей вершиной
            Local(L.mDest);           // и рекурсивно вызываем эту же проц.
          end;
          Dec(Cost, L.mValue);        // восстанавливаем стоимость
        end;
        L:= aNode.OutLinkNext;        // следующий исходящий линк
      end;
    end else begin
      // Здесь Count = mNodes.GetCount-1 -- все вершины, кроме aFin, окрашены
```

```
      // Ищем замыкающий линк на вершину aFin
      L:= aNode.OutLinkFirst;
      while Assigned(L) and (L.mDest<>aFin)
        do L:= aNode.OutLinkNext;
      if Assigned(L) then begin
        // Замыкающий линк найден:
        Inc(Cost, L.mValue);       // накапливаем стоимость
        L.mDest.mPred:= aNode;     // метим предыдущей вершиной
        if Cost < aCost            // если она меньше минимальной
          then SaveResult;         // сохраняем путь и его стоимость
        Dec(Cost, L.mValue);       // восстанавливаем стоимость
      end;
    end;
    // При выходе восстанавливаем счётчик вершин и цвет
    Dec(Count);
    aNode.mColor:= CWhite;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenHamPath_Full }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;               // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  aCost:= MaxInt;// текущая минимальная стоимость = Maxint
  ResetNodes;            // очистка вспомогательных полей
  Cost:= 0;              // накопленная стоимость = 0
  Count:= 0;             // счётчик окрашенных вершин = 0
  Local(aStart);         // вызов рекурсивной процедуры
  // Если путь не обнаружен, возвращаем минус 1
  if aCost = MaxInt then aCost:=-1;
end;
```

Понятно, что простота и точность алгоритма чем-то оплачена. Чем? — временем, разумеется. Количество возможных путей из вершины *X* в вершину *Y* на полном графе с *N* вершинами составляет порядка *(N-2)!*, и потому время полного перебора с ростом размера графа растёт с той же чудовищной факториальной крутизной. Нынешние компьютеры заметно «притормаживают» уже при *N* = `15...20`, а на третьем десятке перед задачей капитулирует даже самый мощный из них (точнее, его пользователь).

Подведём промежуточный итог. С одной стороны, имеем очень *быстрый* жадный алгоритм, находящий Гамильтонов путь, пусть и не идеальный. С другой стороны, есть алгоритм полного перебора, дающий *точный* результат. К сожалению, из-за крайней медлительности полный перебор не применим к сколь-нибудь крупным графам. Хотелось бы найти относительно *быстрый*, и по возможности *точный* алгоритм поиска минимального Гамильтонова пути, — этому посвятим остаток главы.

32.6. Покрывающие деревья и штрафование вершин

Улучшенные методы решения Гамильтоновых задач на *неориентированных* графах и *орграфах* принципиально отличаются. И потому метод для *орграфов*

отложим на будущее, здесь рассмотрим построение кратчайшего Гамильтонова пути на *неориентированных* графах. Метод будет основан на построении минимального покрывающего дерева (остова) весьма быстрым алгоритмом *Прима*, подробно изложенном в главе 23.

Однако причём тут *Прим*? Дело в том, что искомый Гамильтонов путь между вершинами *X* и *Y* является и одним из *покрывающих деревьев* графа. Пусть на некотором графе построен минимальный остов, и случилось так, что вершины *X* и *Y* в этом дереве имеют степени **1**, а все прочие вершины — степени **2** (степень вершины — это количеству инцидентных ей рёбер). Это значит, что образовалась цепочка рёбер от *X* к *Y*, которая и будет кратчайшим Гамильтоновым путём. Конечно, столь удачное совпадение маловероятно, однако эту вероятность можно повысить, пустившись на невинный «обман» алгоритма Прима, известный как *метод штрафования вершин*.

Для дальнейших пояснений введём в рамках этой главы несколько терминов. Назовём *n*-вершиной покрывающего дерева, вершину, степень которой в этом дереве равна *n*. Тогда **1**-вершины назовём *листьями*, **2**-вершины — *суставами*, а вершины степеней **3** и более — *узлами* (рис. 32-3).

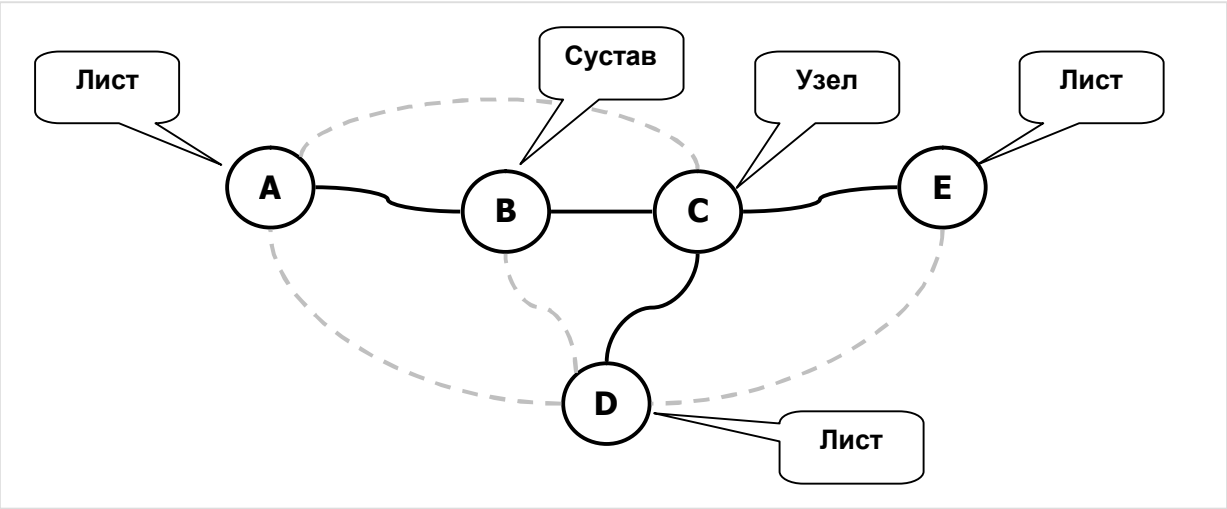


Рис. 32-3 — Остовное дерево с тремя листьями, одним суставом и одним узлом степени 3

В этих терминах конечную цель можно выразить так: в Гамильтоновом пути вершины *X* и *Y* должны быть *листьями*, а все прочие вершины — *суставами*. Никаких *узлов* в этом дереве быть не должно. Однако алгоритм Прима для минимального покрывающего остова не даёт таких гарантий, и потому предстоит модифицировать его.

Обратите внимание на особенность алгоритма Прима: в ходе построения минимального остова вершины, плотно окружённые ближними соседями, «притягивают» их к себе, обрастая большим числом рёбер, и становясь *узлами* дерева. Наоборот, отдалённые от прочих вершины соединяются с деревом лишь одним ребром, становясь *листьями*. Поскольку общее количество рёбер в дереве

фиксировано и равно *N-1* (*N* — количество вершин), то чем больше *листьев* в дереве, тем больше в нём *узлов*. Таким образом, по количеству листьев *L* можно судить, насколько построенный остов отличается от пути Гамильтона. Разность *L-2* — количество листьев за вычетом двух — назовём *невязкой*. Для Гамильтонова пути невязка будет равна нулю. Обсудим пути снижения этой *невязки*.

Излагаемую далее технику называют *методом штрафования вершин*, хотя, что это объясняет? Чем провинились вершины? Мы дадим несколько иную — геометрическую — интерпретацию этого приёма. И пусть она не точна количественно, зато вполне наглядна. Рассмотрим граф, все вершины и рёбра которого лежат в одной плоскости (рис. 32-4).

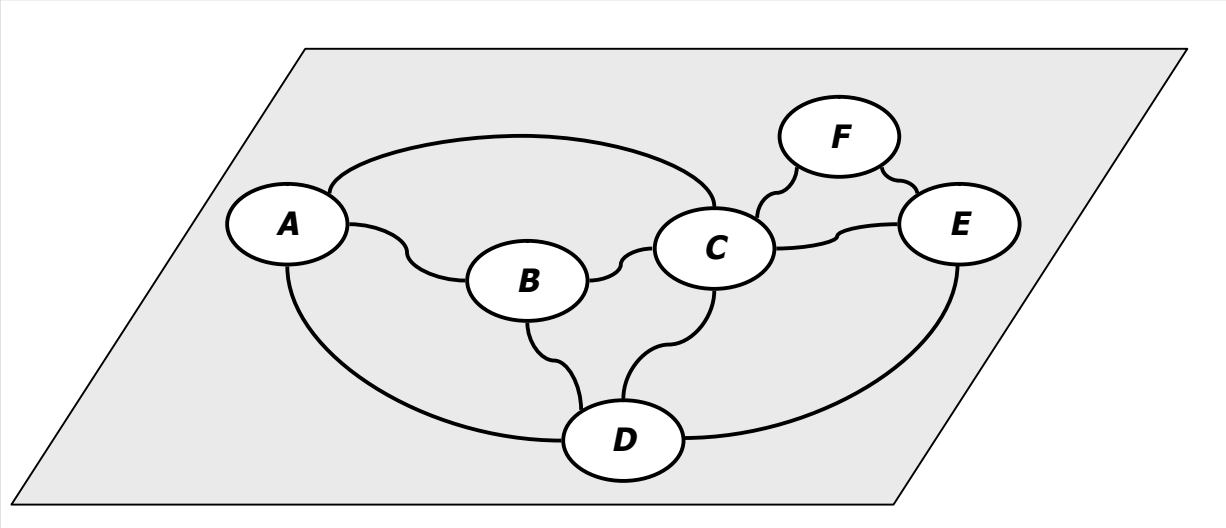


Рис. 32-4 — Граф, вершины у рёбра которого лежат на плоскости

Здесь к вершине *C* плотно прижато несколько соседей, и потому алгоритм Прима с большой вероятностью построит дерево с узлом в вершине *C*, — такое дерево не будет Гамильтоновым путём (рис. 32-5).

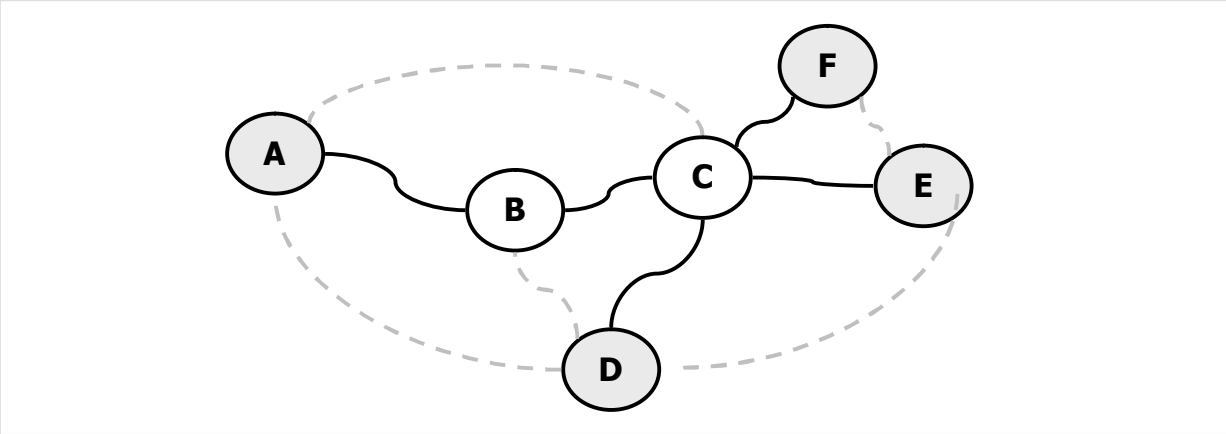


Рис. 32-5 — Минимальное покрывающее дерево с узлом C и четырьмя листьями

Теперь слегка деформируем граф, приподняв вершину *C* над плоскостью так, чтобы длина инцидентных ей рёбер увеличилась на одну и ту же величину **D**, как показано на рис. 32-6.

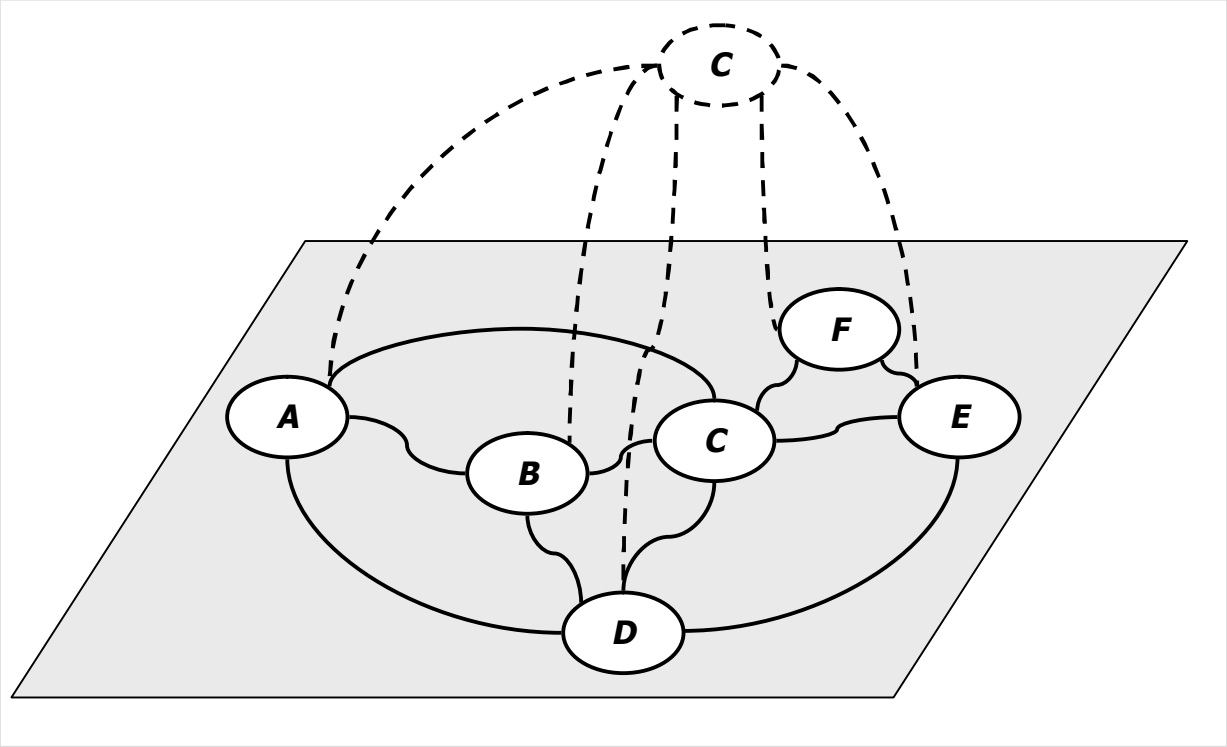


Рис. 32-6 — Граф с оштрафованной (приподнятой) вершиной C

«Скормив» этот деформированный граф алгоритму Прима, с некоторой вероятностью получим результат, показанный на рис. 32-7, — здесь покрывающее дерево выродилось в Гамильтонов путь. Случилось так потому, что, удлинение рёбер, инцидентных приподнятой вершине, сделало их менее привлекательными для Прима. И потому Прим оставил в дереве лишь кратчайшее из рёбер, зато привлёк к остову другие рёбра. Так бывший узел C стал листом, а бывшие листы D, E и F превратились в суставы.

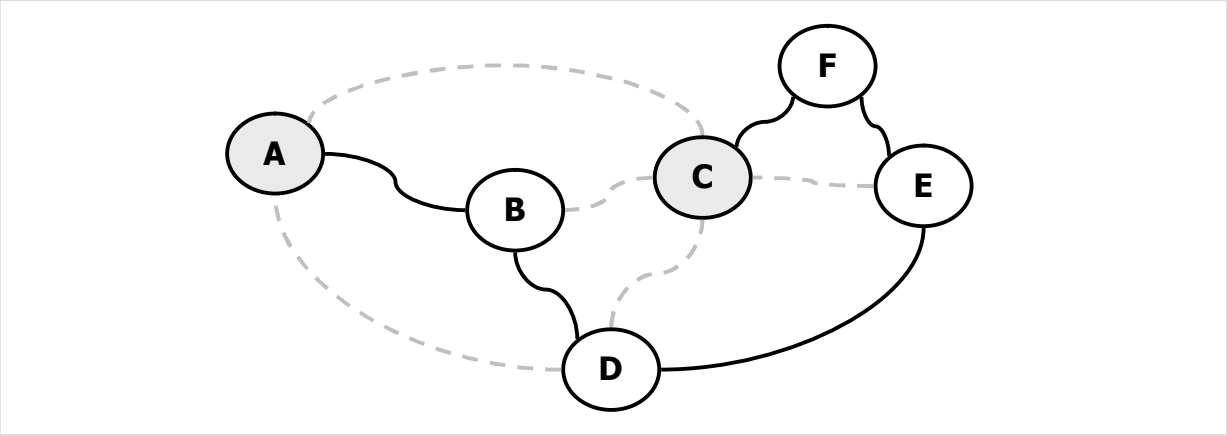


Рис. 32-7 — Покрывающее дерево, построенное после деформации исходного графа (A и C – листы, остальные вершины — суставы)

Итак, деформация исходного графа удлинением рёбер, инцидентных некоторому узлу, может изменить конфигурацию минимального покрывающего дерева, приблизив его форму к искомому Гамильтонову пути. Разумеется, что стоимость деформированного дерева исказится, — она увеличится. Но по

исходным весам рёбер всё ещё можно определить истинную стоимость этого подграфа. Настала пора ответить на следующие вопросы:

- Как можно «удлинять» рёбра, не нарушая их исходной длины? Исходная длина нужна для определения стоимости Гамильтонова пути.
- Почему поднятие вершин и «удлинение» рёбер в целом ведёт к решению задачи?
- На какую высоту «приподнимать» вершины (удлинять рёбра), чтобы найти решение по возможности быстро?

32.6.1. Способ «удлинения» рёбер

«Удлинять» рёбра «поднятием» вершины на некоторую высоту будем посредством так называемого *штрафа*, хранимого в поле вершины. «Штрафная» длина некоторого ребра *X-Y* определится сложением его истиной длины со штрафами двух его инцидентных вершин:

Штрафная длина *X-Y* **= Истиная длина** *X-Y* **+ штраф** *X* **+ штраф** *Y*

В нашей программе штраф вершины будет храниться в поле **TNode.mDist**.

32.6.2. Сортировка остовных деревьев

Теперь покажем, как и почему штрафование вершин ведёт, либо к точному, либо к близкому к нему решению. Мысленно представим все возможные остовы графа, построенные при нулевых штрафах. Расположим эти остовы в порядке их стоимости, начиная с «лёгких», и понаблюдаем за тем, как штрафование влияет на этот рейтинг-лист. Оштрафуем вершину *X* на величину **D**. Разумеется, что это увеличит веса всех деревьев, но в разной степени, и потому их порядок в рейтинге изменится. Меньше других (на **1•D**) «потяжелеют» деревья, в которых оштрафованная вершина *X* является листом (1-вершиной). На **2•D** увеличатся веса деревьев, где она является суставом (2-вершиной), и так далее. И тогда в нашем воображаемом рейтинге остовы, где оштрафованная вершина является *узлом*, опустятся, а те, где она состоит *листом* или *суставом* — соответственно поднимутся.

Далее рассмотрим, каковы будут последствия от штрафа начальной и конечной вершин *X* и *Y*. Напомню, что по условиям задачи они должны стать крайними вершинами, то есть, листьями. Огромный штраф «отдалит» эти вершины так далеко от прочих вершин, что модифицированный алгоритм Прима в легчайшем остове сделает их *листьями*, и соответствующая группа деревьев, где эти вершины являются листьями, *поднимется* в топ нашего рейтинга, — в этом состоит первый шаг будущего алгоритма.

Во «всплывшей» вверх группе деревьев окажется искомый нами *кратчайший* Гамильтонов путь, а также другие Гамильтоновы пути и узловатые образования с

лишними листьями. Дабы отделить Гамильтоновы пути от других деревьев — приподнять их в рейтинге — продолжим аккуратно штрафовать оставшиеся узлы, то есть, вершины со степенями более двух. Пусть во вновь полученном легчайшем дереве найден такой узелок, и оштрафован на величину **D**. Тогда все деревья, где степень этой вершины окажется больше двух, опустятся в рейтинге, а те, где она равна двум или меньше, — поднимутся. И у алгоритма Прима появится шанс построить минимальный остов, где бывший *узел* станет *суставом*, а количество *листьев* уменьшится. Отметим, что в результате этого штрафования веса всех Гамильтоновых путей увеличатся на одну и ту же величину **2•D**, и потому кратчайший из них в своём семействе по-прежнему останется кратчайшим. В конце концов, после нескольких таких итераций на вершине рейтинга может оказаться остов, все вершины которого, за исключением двух крайних, будут суставами. Если это случится, то данное дерево окажется Гамильтоновым путём, причём кратчайшим.

32.6.3. Выбор шага штрафования и сходимость

Итак, штрафованием узлов и построением минимального остова можно поднять искомую Гамильтонову цепь на вершину «рейтинга» остовных деревьев. И хотя алгоритм Прима весьма быстр, неоправданное его повторение существенно замедлит работу. Каково оптимальное приращение штрафа? Слишком осторожный штраф не распрямит узелок, и придётся повторно штрафовать его, обращаясь затем к алгоритму Прима. Слишком крупный штраф превратит оштрафованный узел в излишний лист, а узел завяжется в другой вершине. С целью вычисления оптимального штрафа каждого узла рассмотрим следующую идею.

Рассмотри рис. 32-8, где дано дерево, полученное на некотором этапе алгоритмом Прима. Здесь для узлов *B* и *C* надо вычислить минимальные штрафы. Будем поочерёдно «разрывать» рёбра, инцидентные узлу, а образовавшиеся при этом два поддерева графа соединять другими рёбрами, не инцидентными штрафуемому узлу, так, чтобы приращение стоимости вновь образованного дерева в итоге оказалось минимальным. Например, после разрыва ребра *B-C* образовавшиеся поддеревья можно соединять рёбрами *A-C*, *A-F*, *D-C*, *D-E*.

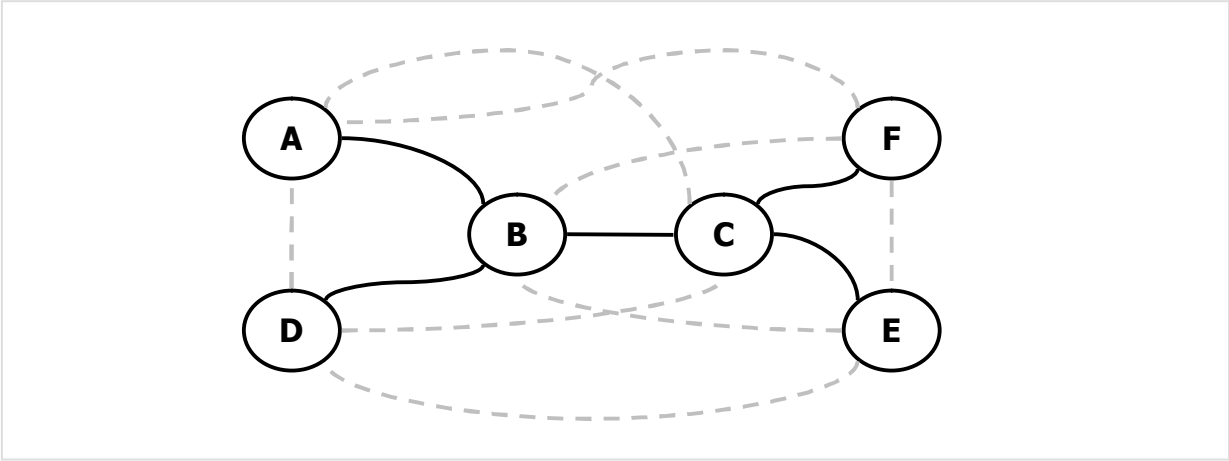


Рис. 32-8 — Остовное дерево с двумя узлами в вершинах B и C

Пусть кратчайшим из них оказалось ребро *A-C* (рис. 32-9), тогда минимальное приращение штрафа для разрыва ребра *B-C* составит:

```
Delta(B-C) = Cost(A-C) - Cost(B-C),
```

где **Cost** — функция, вычисляющая стоимость ребра с учётом текущих штрафов инцидентных ему вершин. Теперь, если оштрафовать вершину *B* на величину **Delta**, то штрафная цена ребра *B-C* сравняется со штрафной ценой ребра *A-C*, и оба они станут равно привлекательными для алгоритма Прима.

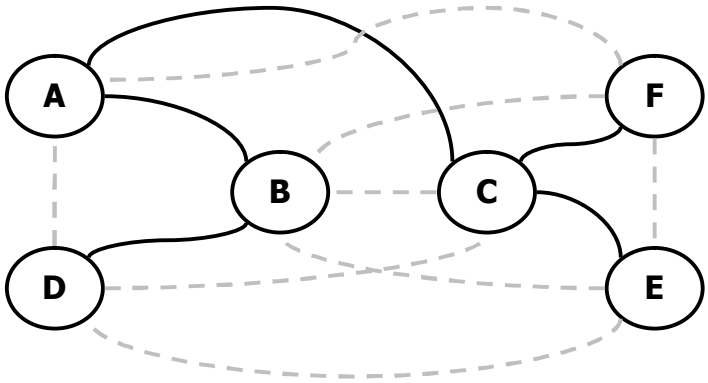


Рис. 32-9 — Разрыв ребра B-C и соединение поддеревьев ребром A-C

Так поочерёдно находятся приращения штрафов и для разрыва рёбер *A-B* и *B-D*, после чего выбирается минимальное из них, и применяется в качестве приращения штрафа вершины *B*. Аналогично ищется приращение штрафа для вершины *C*.

Обработав так все *узлы* текущего дерева, и назначив им новые штрафы, мы вправе ожидать, что следующее применение алгоритма Прима породит остов с меньшим количеством узлов и листьев. Увы, ожидание это оправдывается не всегда ввиду сложного взаимного влияния штрафов разных вершин. Невязка *L-2* порой не только не уменьшается, но даже увеличивается. И всё же после нескольких итераций тенденция к уменьшению невязки проявляется. Но только тенденция. Со временем может наблюдаться эффект кочующих узлов: после очередной итерации узел переходит с одной вершину на другую, а затем обратно, что напоминает многократное повторение позиций в шахматной партии. Ничья, равносильная поражению? — прочь капитуляцию! Выход найдём, если решим ещё *два* вопроса: когда прекратить итерации, и что делать с оставшимися узлами.

32.6.4. «Разрубание» узлов

Хотя разумное штрафование вершин и ведёт к постепенному повышению рейтинга искомого нами минимального Гамильтонова пути, однако он может так и не «всплыть» на вершину рейтинга, дав тем самым точное решение. Вместо него на верхушке оказываются деревья, по форме и весу близкие к искомому

Гамильтонову пути. Почему бы здесь не «развязать» оставшиеся узелки что называется «разрубив» их? Сделаем это способом, похожим на тот, что применён для вычисления штрафов. Будем поочерёдно разрывать рёбра, инцидентные узлу, а получившиеся поддеревья соединять лист к листу через кратчайшие рёбра.

На рис. 32-10 дан пример дерева с двумя узлами *B* и *C*, и двумя избыточными листьями *D* и *E* (здесь невязка *L-2* равна двум).

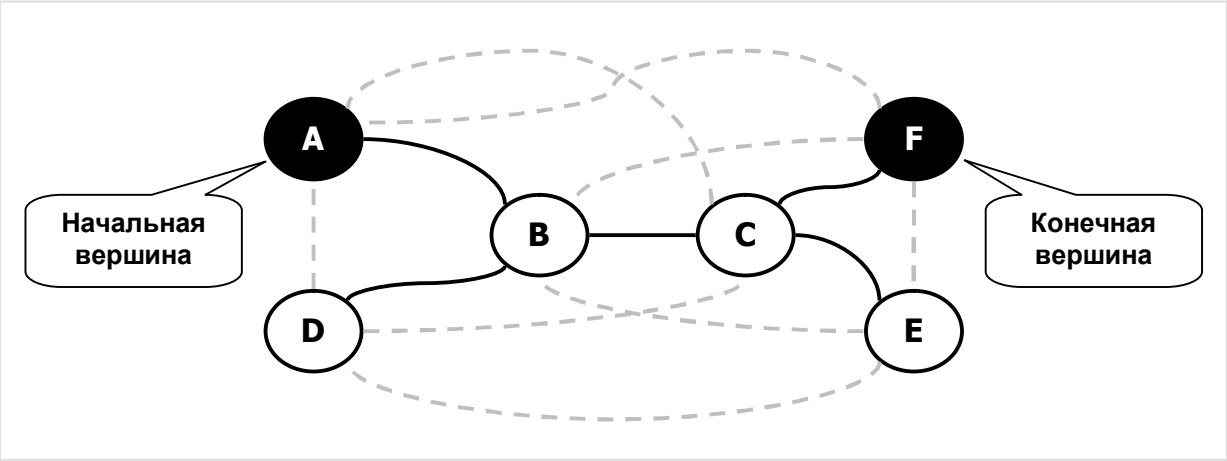


Рис. 32-10 — Исходное дерево с двумя узлами (невязка равна двум)

Для оптимального «разрубания» узла *B* поочерёдно разрываем инцидентные ему рёбра, а получающиеся при этом поддеревья соединяем «лист в лист». Так, после разрыва ребра *A-B* соединим поддерево *A* с остальными вершинами, и здесь кратчайшим мостом может оказаться, например, ребро *A-D* (рис. 32-11).

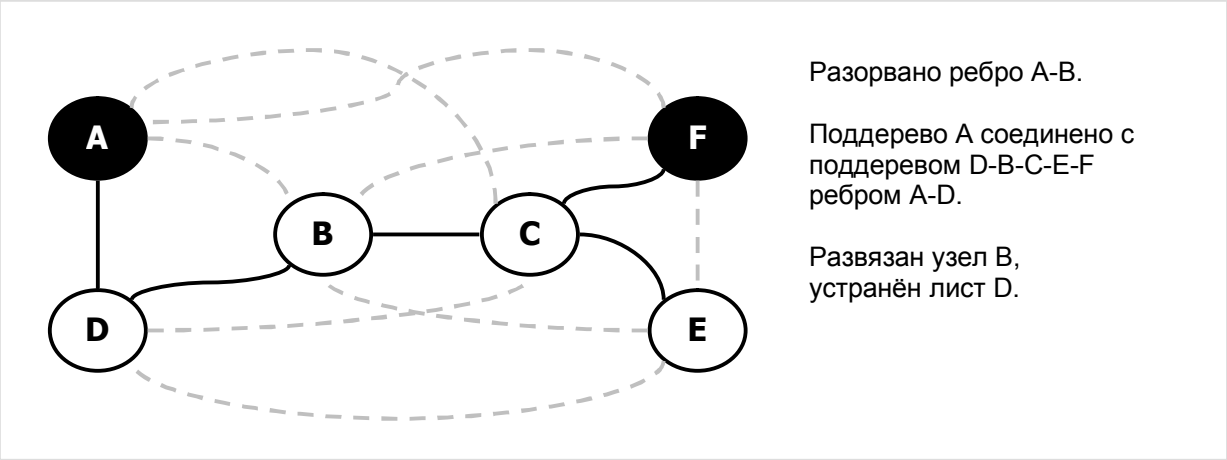


Рис. 32-11 — Разрыв ребра A-B и включение ребра A-D

В другой ситуации, после разрыва ребра *B-C*, кратчайшим соединением листов может оказаться ребро *D-E* (см. рис. 32-12 — здесь устраняются сразу два узла и два листа). Перебрав, таким образом, все варианты разрывов и соединений, выберем тот, который даёт наименьшее приращение стоимости. Аналогично устраняется узел *C*.

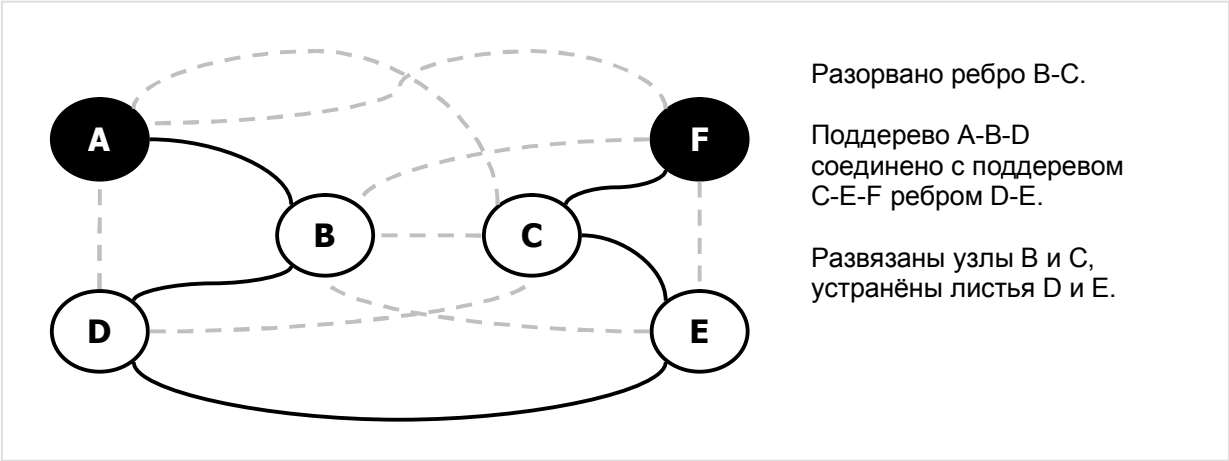


Рис. 32-12 — Разрыв ребра B-C и включение ребра D-E

Практика показала, что такое «силовое вмешательство» даёт цепь, весьма близкую к кратчайшему Гамильтонову пути. После некоторого количества итераций (на разных остовах) эти манипуляции дают либо точный, либо весьма близкий к нему результат.

32.6.5. Счётчики штрафования

Теперь обратимся ко второй проблеме: когда прекратить итерации и удовлетвориться полученным решением? Здесь возможны разные стратегии, и одна из них — простейшая — состоит в применении *счётчиков* штрафования вершин. Поскольку штрафование, в конечном счёте, ведёт к блужданию узлов, есть смысл ограничить число штрафов, налагаемых на каждую вершину. Учредим для каждой вершины счётчик, и при каждом штрафовании будем уменьшать его вплоть до нуля. Вершину с нулевым счётчиком далее не штрафуем. Когда после некоторого числа итераций все *узлы* исчерпают свои счётчики, наложение штрафов прекратится (хотя запас штрафов в *листах* и *суставах* может оставаться).

Исходные значения счётчиков можно задать *однократно*, но угадать это начальное значение непросто. Можно сделать иначе: после каждой удачной итерации, приводящей либо к снижению невязки, либо к снижению стоимости очередного промежуточного решения, *возобновлять* все счётчики, занося туда небольшие числа. Тем самым алгоритму даётся ещё несколько попыток улучшить решение. Рано или поздно, когда невязка и стоимость дерева перестанут снижаться, возобновление счётчиков прекратится, и в узлах дерева они будут исчерпаны, что приведёт к прекращению попыток.

В данном алгоритме стратегия обновления счётчиков учтёт также и размер графа, поскольку в больших графах может потребоваться больше попыток штрафования. И потому изначально все счётчики устанавливаются равными **Count** $= N$, где N — число вершин графа. Затем после каждой удачной итерации счётчики всех верши возобновляются новым значением, вычисляемым из предыдущего по формуле:

Count = 2 + 3•Count/4

Возобновляемое значение счётчиков, таким образом, асимптотически снижается.

32.6.6. Алгоритм в целом

Обсудив основные идеи метода штрафования вершин, представим порядок действий в целом.

1. Установить начальное состояние всех вершин: цвет белый, штрафы нулевые, счётчики штрафования ненулевые, приращение счётчиков **`Count:= N`**.

2. Для исходной и конечной вершин назначить столь большие штрафы, чтобы гарантировано превратить их в листья.

3. Начало цикла.

    a. Методом Прима (модифицированным) построить покрывающей остов минимальной стоимости. Если дерево не получено, то решения нет.

    b. Подсчитать степени вершин в остове и невязку *L-2* через число лишних листьев.

    c. Исходя из текущей конфигурации остова, подсчитать новые штрафы в узлах и количество штрафуемых узлов **`Nodes`**. Здесь **`Nodes`** будет равно нулю, когда невязка равна нулю, или когда исчерпаны счётчики штрафования в узлах.

    d. Если в остове остались узлы, «разрубить» их, получив очередной Гамильтонов путь, близкий к искомому или совпадающий с ним. Если путь оказался лучше (дешевле) предыдущего, то запомнить его.

    e. Если удалось улучшить решение или снизить невязку, то переустановить счётчики штрафования для всех вершин новым значением: **`Count = 2 + 3•Count/4`**.

    f. Если количество оштрафованных вершин **`Nodes`** достигло нуля (см. пункт **`c`**), завершить цикл.

4. Если решение существует, то преобразовать множество рёбер дерева в Гамильтонов путь (последовательность вершин).

В следующем пункте рассмотрим код, реализующий этот алгоритм. Он, наряду с ранее упомянутыми параметрами, будет принимать ещё один параметр: так называемую *исключённую* вершину. Будучи отличен от **`NIL`**, параметр указывает на вершину, исключаемую из обработки алгоритмом. Вернёмся к пояснению этого параметра в следующей главе при поиске цикла Гамильтона.

32.7. Ускоренный приближённый метод GenHamPathStrip

32.7.1. Вспомогательный объект

Прежде, чем представить метод, рассмотрим вспомогательный объект **THamLink**, необходимый для составления минимального остова с учётом штрафов вершин (листинг 32-3). Он будет элементом приоритетной очереди, упорядоченной в порядке не убывания «штрафной» длины рёбер.

Листинг 32-3 — Вспомогательная функция
и объект для представления оштрафованных рёбер

```
// Вспомогательная функция для вычисления стоимости линка
// с учётом штрафов инцидентных вершин

function CalcValue(aLink: TLink): integer;
begin
  with aLink do Result:= mValue + mOwner.mDist + mDest.mDist;
end;


type  // THamLink -- вспомогательный линк для построения остова

  THamLink = class (TItem)
    mValue: integer; // стоимость с учётом штрафов
    mLink: TLink;    // ссылка на исходный линк
    constructor Create(aLink: TLink);
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;

constructor THamLink.Create(aLink: TLink);
begin
  inherited Create;
  mLink:= aLink;
  // Условная стоимось формируется с учётом штрафов
  mValue:= CalcValue(aLink);
end;

function THamLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  with mLink do begin
    // Для неориентированного графа сравниваем исходную и конечную вершины
    // на предмет встречных линков
    if not mOwner.mOwner.mDirect and  // если не орграф
      (mOwner=(arg as THamLink).mLink.mDest) and
      (mDest=(arg as THamLink).mLink.mOwner)
      then begin
        // Здесь линки направлены встречно, отвергаем дубликат:
        Result:= cmpEq;    Exit;
      end;
  end;
  // Если линки не совпадают, то сортируем по неубыванию стоимости
  Result:= cmpLess;
  if mValue > (arg as THamLink).mValue then Result:= cmpGreate;
end;
```

32.7.2. Метод GenHamPathStrip

В следующем листинге дан метод приближённого ускоренного поиска кратчайшего Гамильтонова пути между вершинами **aStart** и **aFin** с возможным исключением вершины **aStrip**. Функция возвращает последовательность вершин, составляющих путь, и стоимость пути.

Листинг 32-4 — Поиск Гамильтонова пути методом штрафования вершин

```
function TGraph.GenHamPathStrip(aStart,            // начальная вершина
                                aFin,              // конечная вершина
                                aStrip: TNode;     // исключаемая вершина
                                var aCost: integer // длина (стоимость) цепи
                                ): TBuffer;

// Поля вершин TNode используются так:
// mDist    -- текущий штраф
// mPower   -- текущая степень вершины в остове
// mColor   -- текущая окраска
// mRoot    -- принадлежность поддереву
// mFlow    -- для временного хранения приращения штрафа
// mLimit   -- обратный счётчик штрафования (до нуля)

var   Que : TSet; // очередь рёбер по неубыванию стоимости с учётом штрафов
     Tree : TSet; // множество чёрных рёбер покрывающего дерева
   Marked : TBuffer;  // очередь вершин для маркировки поддеревьев

  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка штрафных полей mDist (вызывается единожды)
```

```
procedure ClearFines;
var Node: TNode;       // текущая вершина
begin
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      mDist:= 0;     // штраф
      mLimit:= 1;    // ненулевой предел на штрафование вершины
      mRoot:= nil;   // корневая метка
    end;
    Node:= NodeNext;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Возобновление счётчиков штрафования Node.mLimit
// Вызывается после уменьшения невязки или стоимости

procedure SetLimits(aLimit: integer);
var Node: TNode;
begin
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mLimit:= aLimit; // предел на штрафование вершины (счётчик)
    Node:= NodeNext;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Очистка цвета вершин и рёбер, а также степеней вершин
// (вызывается на каждой итерации перед построением дерева)
// mColor = CWhite    -- цвет белый
// mPower = 0         -- степень вершины в дереве

procedure ResetNodesAndLinks;
var Node : TNode;
    Link : TLink;
begin
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mColor:= CWhite; // цвет
    Node.mPower:= 0;      // степень вершины в покрывающем дереве
    // Сбросить цвета рёбер
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      Link.mColor:= CWhite;
      Link:= Node.OutLinkNext;
    end;
    Node:= NodeNext;
  end;
  // Исключаемую вершину пометить красным:
  if Assigned(aStrip) then aStrip.mColor:= CRed;
end;

// - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Локальная процедура построения минимального покрывающего дерева
// Формирует множество чёрных рёбер дерева и помещает в множество Tree

function GenCover: boolean;

var Node : TNode;      // текущая вершина
    Link : TLink;      // для пометки встречного линка
      HL : THamLink;   // ребро для дерева
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
    // Добавление в очередь Que линков к соседним белым вершинам.
    // Линки HL сортируются в Que по неубыванию штрафной длины

    procedure AddTreeLinks(aNode: TNode);
    var  Link : TLink;
           HL : THamLink;
    begin
      // Обработка исходящих связей
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) do begin
        // Вставить только линки, ведущие к белым вершинам
        if Link.mDest.mColor = CWhite then begin
          // Сконструировать вспомогательный линк:
          HL:= THamLink.Create(Link);
          // И вставить в очередь:
          if not Que.Insert(HL) then HL.Free;
        end;
        Link:= aNode.OutLinkNext;
      end;
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  var N : integer;  // количество присоединяемых вершин

  begin  { GenCover }
    Result:= true;
    // Предварительная очистка:
    Tree.ClrAndDestroy;    // дерево результата
    ResetNodesAndLinks;    // очистка вершин и рёбер: mColor = CWhite
    Node:= aStart;         // начинаем с исходной вершины
    Node.mColor:= CBlack; // красим чёрным
    // Присоединить все вершины графа кроме красной
    // (линков будет на единицу меньше)
    N:= mNodes.GetCount-1;
    if Assigned(aStrip) then Dec(N); // красную обойти стороной
    while Assigned(Node) and (Tree.GetCount < N) do  begin
      // Добавить в очередь Que линки к ближайшим белым вершинам:
      AddTreeLinks(Node);
      // Перебрать линки Que в поиске ближайшей белой вершины
      Node:= nil;                      // искомая вершина пока не найдена
      HL:= Que.GetFirst as THamLink;   // первый линк -- кратчайший
      while Assigned(HL) do begin
        Que.Delete(HL);                // удалить линк из очереди
        // Если линк ведёт к белой вершине, то ближайшая найдена
        with HL.mLink do if mDest.mColor = CWhite
            then Node:= mDest;    // приёмник связи
        if Assigned(Node) then begin
          // Здесь ближайшая белая вершина найдена:
          Node.mColor:= CBlack;      // присоединить её к множеству чёрных
          Tree.Insert(HL);           // вставить линк в результат (дерево)
          HL.mLink.mColor:= CBlack; // и отметить чёрным
          // Отметить чёрным и встречный линк:
          Link:= HL.mLink.GetReverse;
          if Assigned(Link) then Link.mColor:= CBlack;
          Break; // выход из while
        end else begin
          // Этот линк не ведёт к белой вершине, уничтожить его
          HL.Free;
          HL:= Que.GetNext as THamLink; // и взять следующий
        end;
      end; // while
    end; // while
    // Если остов не построен, то граф не связан, и решения не существует
```

```
    if Tree.GetCount < N then begin
      Tree.ClrAndDestroy;
      Result:= false;
    end; // if
    Que.ClrAndDestroy;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт степеней всех вершин, помещение их в поля TNode.mPower
  // и подсчёт невязки через количество листьев

  function CalcPowerNodes: integer;
  var HL : THamLink;   // текущий линк дерева Tree
      Node: TNode;
  begin
    // Предварительно очистить поля mPower
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mPower:= 0;  // степень вершины в покрывающем дереве
      Node:= NodeNext;
    end;
    // Перебор линков в дереве Tree:
    HL:= Tree.GetFirst as THamLink;
    while Assigned(HL) do begin
      // наращиваем степени вершин дерева :
      Inc(HL.mLink.mOwner.mPower);    // в источнике дуги
      Inc(HL.mLink.mDest.mPower);     // в приёмнике дуги
      HL:= Tree.GetNext as THamLink;
    end;
    // Подсчитать невязку как сумму листьев минус 2
    Result:=-2;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mPower=1 then Inc(Result);
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Маркировка поддерева из данной вершины (метим поле mRoot).
  // Дерево обходим в ширину по чёрным линкам.
  // Маркированные аершины красим серым.

  procedure Mark(aNode: TNode);
  var   Link : TLink;
        Node : TNode;
  begin
    Marked.Clear;          // очередь вершин для маркировки поддеревьев
    Marked.Put(aNode);     // занести в очередь корневую вершину
    aNode.mColor:= CGray; // и окрасить серым
    // Пока очередь не пуста:
    while Marked.GetCount > 0 do begin
      Node:= Marked.Get as TNode;   // выбрать вершину из очереди
      Node.mRoot:= aNode;           // и отметить её в поле mRoot
      Link:= Node.OutLinkFirst;
      // Поместить в очередь соседние вершины, связанные чёрными линками:
      while Assigned(Link) do begin
        // Если линк чёрный
        if (Link.mColor = CBlack) and
           // и цвета инцидентных вершин не совпадают
           (Node.mColor <> Link.mDest.mColor) then begin
          // то присоединить соседа:
          Marked.Put(Link.mDest);     // в очередь
          Link.mDest.mColor:= CGray;  // и окрасить серым
```

```
        end;
        Link:= Node.OutLinkNext;
      end;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вычисление минимального штрафа в результате разрыва ребра aLink
  // Вызывается из CalcNodeFine
  // Возвращает штраф и признак успеха aStatus

  function CalcLinkFine(aLink: TLink; var aStatus: boolean): integer;
  var Link : TLink;
      Node : TNode;
      Fine : integer;
  begin
    Result:= MaxInt;  aStatus:= false;
    // В текущий момент все вершины окрашены чёрным, а линки так:
    // принадлежащие дереву - чёрным, разорваный - серым, прочие - белым
    // Пометить вершины двух поддеревьев (поля mRoot)
    Mark(aLink.mOwner);  // поддерево mOwner
    Mark(aLink.mDest);   // поддерево mDest
    // Найти ребро минимального веса, соединяющее два поддерева
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Восстановить цвет после маркировки (кроме красной вершины):
      if Node <> aStrip then Node.mColor:= CBlack;
      // Если это вершина поддерева mOwner и она не корневая
      if (Node.mRoot = aLink.mOwner) and (Node <> aLink.mOwner) then begin
        // то обрабатать её, перебирая исходящие линки
        Link:= Node.OutLinkFirst;
        while Assigned(Link) do begin
          // Если ребро ведёт в другое поддерево,
          // то запомнить его минимальный вес с учётом штрафа:
          // Link.mDest.mRoot -- метка вершины
          // aLink.mDest -- корень оторванной части
          if Link.mDest.mRoot = aLink.mDest then begin
            Fine:= CalcValue(Link);
            if Result > Fine then begin
              Result:= Fine;  // предварительный результат
              aStatus:= true; // признак наличия результата
            end;
          end;
          Link:= Node.OutLinkNext;
        end;
      end;
      Node:= NodeNext;
    end;
    // Окончательный результат (приращение штрафа)
    // вычислить как разность
    if aStatus then Result:= Result - CalcValue(aLink);
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт минимального приращения штрафа для данной вершины
  // вызывается из CalcFines
  // Возвращает штраф и признак успеха aStatus

  function CalcNodeFine(aNode: TNode; var aStatus: boolean): integer;
  var Link, Reverse: TLink;  // прямой и обратный линки
      Fine: integer;         // вычисленный штраф
      OK: boolean;           // признак успеха
  begin
    Result:= MaxInt;  aStatus:= false;
```

```
    // Поочерёдно "разрываем" линки вершины, входящие в состав дерева
    // (чёрные линки) и строим обходные пути,
    // соединяющие разорванные части дерева
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) and (Result > 0) do begin
      if Link.mColor = CBlack then begin
        // Этот линк принадлежит дереву
        // Временно "разорвать" его и встречный линк (красить серым)
        Link.mColor:= CGray;
        Reverse:= Link.GetReverse;
        if Assigned(Reverse) then Reverse.mColor:= CGray;
        // Вычислить штраф:
        aNode.OutPosPush;   // сохранить позицию перебора
        Fine:= CalcLinkFine(Link, OK); // вычислить штраф
        aNode.OutPosPop;    // восстановить позицию перебора
        // Вновь восстанавить цвета прямого и встречного линков
        Link.mColor:= CBlack;
        if Assigned(Reverse) then Reverse.mColor:= CBlack;
        // Запоминаем минимальное приращение штрафа
        if OK and (Result > Fine) then begin
          Result:= Fine;
          aStatus:= true;
        end;
      end;
      Link:= aNode.OutLinkNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт и установка штрафов для всех вершин
  // Возвращает количество оштрафованных вершин

  function CalcFines: integer;
  var Node: TNode;
      Fine: integer;
      OK: boolean;
  begin
    Result:=0; // для подсчёта оштрафованных вершин
    // Сканировать вершины, степень которых превышает 2,
    // вычислять приращения штрафов и временно помещать в поля mFlow
    // Node.mPower -- степень вершины в дереве
    // Node.mFlow -- сюда временно помещаем приращение штрафа
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= 0;          // mFlow -- приращение штрафа
      if (Node.mPower > 2)     // если это узел
         and (Node.mLimit > 0)// и лимит штрафования не исчерпан
        then begin
        Dec(Node.mLimit); // уменьшить лимит
        Inc(Result);      // нарастить счётчик оштрафованных
        // Вычисляить минимальный штраф, уменьшающий её степень:
        PosPush; // сохранить позицию перебора вершин
        Fine:= CalcNodeFine(Node, OK);
        PosPop;  // восстановить позицию перебора вершин
        if OK then begin
          if Fine = 0 then Fine:=1;  // реальный штраф больше нуля
          Node.mFlow:= Fine;         // mFlow -- приращение штрафа
        end;
      end; // if
      Node:= NodeNext;
    end; // while
    // Нарастить штрафы вершин mDist:= mDist + Node.mFlow
    Node:= NodeFirst;
```

```
    while Assigned(Node) do begin
      Inc(Node.mDist, Node.mFlow);
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вычисление минимального приращения цены
  // в результате разрыва ребра aLink
  // Вызывается из CalcDeltaNode
  // Возвращает кратчайший белый линк, соединяющий листья поддеревьев

  function CalcDeltaLink(aLink: TLink; var aRes: TLink): integer;
  var Link : TLink;
      Node : TNode;
  begin
    Result:= MaxInt;
    aRes:= nil;
    // В текущий момент все вершины окрашены чёрным,
    // линки дерева - чёрным, разорванный линк - серым, остальные - белым
    // Пометить вершины двух поддеревьев (поля mRoot)
    Mark(aLink.mOwner);  // поддерево mOwner
    Mark(aLink.mDest);   // поддерево mDest
    // Найти линк минимального веса, соединяющий два поддерева
    // через вершины со степенями 1 (листья)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Восстановить цвет после маркировки (кроме красной вершины)
      if Node <> aStrip then Node.mColor:= CBlack;
      // Если это лист в дереве mOwner
      if (Node.mRoot = aLink.mOwner) and (Node.mPower < 2)
        then begin
        // то обработать его
        Link:= Node.OutLinkFirst;
        while Assigned(Link) do begin
          // Запомнить кратчайший линк к листу другого поддерева
          if (Link.mDest.mRoot = aLink.mDest)  // в другом поддереве
            and (Link.mDest.mPower < 2)        // и степень меньше 2-х
            and (Result > CalcValue(Link))     // и штрафная длина меньше
          then begin
              Result:= CalcValue(Link);  // промежуточный результат
              aRes:= Link;               // белый линк
          end; // if
          Link:= Node.OutLinkNext;
        end; // while
      end; // if
      Node:= NodeNext;
    end; // while
    // Окончательный результат вычислить как разность
    if Assigned(aRes) then Result:= Result - CalcValue(aLink);
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск подходящей пары линков для снижения степени узла aNode
  // Вызывается из ForceCoverTree
  // Возвращает два линка: aBlack -- разрываемый, aWhite -- замещающий

  procedure FindBlackAndWhite(aNode: TNode; var aBlack, aWhite: TLink);
  var Link : TLink;    // прямой линк
      White: TLink;    // белый линк
      Delta: integer; // текущее приращение
      Minim: integer; // минимальное приращение
  begin
    aBlack:= nil;    aWhite:= nil;
```

```
    Minim:= MaxInt;
    // Поочерёдно "разрывать" линки вершины, входящие в состав дерева
    // (эти линки помечены чёрным)
    // и отыскивать обходные пути,
    // соединяющие разорванные части дерева через листья
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      if Link.mColor = CBlack then begin
        // Этот линк принадлежит дереву, разорвать его (красим серым)
        Link.mColor:= CGray;
        // При разрыве уменьшаются степени инцидентных вершин:
        with Link do begin
          Dec(mOwner.mPower);
          Dec(mDest.mPower);
        end;
        aNode.OutPosPush; // сохр. позицию перебора
        // Вычислить приращение цены
        Delta:= CalcDeltaLink(Link, White);
        aNode.OutPosPop;  // восст. позицию перебора
        // Восстанавить цвет линка
        Link.mColor:= CBlack;
        // При воссоединении чёрного линка
        // восст. степени инцидентных вершин:
        with Link do begin
          Inc(mOwner.mPower);
          Inc(mDest.mPower);
        end;
        // Запомнить минимальное приращение
        if Assigned(White) and (Delta < Minim) then begin
          Minim:= Delta;   // приращение
          aBlack:= Link;   // чёрный линк (в дереве)
          aWhite:= White;  // белый линк (вне дерева)
        end;
      end;
      Link:= aNode.OutLinkNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Удаление из дерева вспомогательных линков
  procedure Tree_Delete(aLink: TLink);
  var HL : THamLink; // линк в дереве Tree
      Rev: TLink;    // встречный линк
  begin
    // Уменьшить степени инцидентных вершин чёрного ребра aLink:
    Dec(aLink.mOwner.mPower);
    Dec(aLink.mDest.mPower);
    // Прямой и встречный линк метить белым
    aLink.mColor:= CWhite;
    Rev:= aLink.GetReverse;  // встречный линк
    if Assigned(Rev) then Rev.mColor:= CWhite;
    // Найти в дереве соответствующий вспомогательный линк
    HL:= Tree.GetFirst as THamLink;
    while Assigned(HL) do begin
      if (HL.mLink = aLink) or (HL.mLink = Rev) then Break;
      HL:= Tree.GetNext as THamLink;
    end;
    // Удалить его из дерева и ликвидировать
    if Assigned(HL) then begin
      Tree.Delete(HL);
      HL.Free;
    end;
  end;
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вставка в дерево вспомогательных линков

procedure Tree_Insert(aLink: TLink);
var HL : THamLink;
    Rev: TLink;
begin
  // Увеличить степени инцидентных вершин:
  Inc(aLink.mOwner.mPower);
  Inc(aLink.mDest.mPower);
  // Прямой и встречный линк метить чёрным
  aLink.mColor:= CBlack;
  Rev:= aLink.GetReverse;
  if Assigned(Rev) then Rev.mColor:= CBlack;
  // Сконструировать вспомогательный линк:
  HL:= THamLink.Create(aLink);
  // И вставить его в дерево:
  if not Tree.Insert(HL) then HL.Free;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Принудительное уменьшение степеней узлов остова

procedure ForceCoverTree;
var Node: TNode;
    Black, White : TLink;
begin
  // Обработать узлы (вершины со степенями более 2)
  // Node.mPower -- степень вершины в дереве
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mPower > 2 then begin
      // Вычислить минимальный штраф, уменьшающий её степень
      PosPush; // сохранить позицию перебора вершин
      FindBlackAndWhite(Node, Black, White);
      PosPop;  // восстановить позицию перебора вершин
      // Если пара рёбер обнаружена
      if Assigned(Black) and Assigned(White) then begin
        // Видоизменить дерево
        Tree_Delete(Black);   // удалить из дерева разрываемый линк
        Tree_Insert(White);   // вставить в дерево соединяющий линк
      end;
    end;
    Node:= NodeNext;
  end; // while
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Форсированное преобразование остова в цепочку
// с вычислением стоимости цепочки

function Force(var aCost: integer): TBuffer;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Локальная функция для переноса цепи
  // из буфера aBuf в результат Result
  // с одновременным подсчётом стоимости цепи

  function MoveResult(aRes: TBuffer): integer;
  var HL : THamLink; // текущий линк дерева из Tree
  begin
    Result:= 0;
    aRes.Clear;
    // Перебор линков в покрывающем дереве:
```

```
      HL:= Tree.GetFirst as THamLink;
      while Assigned(HL) do begin
        aRes.Put(HL.mLink);
        Inc(Result, HL.mLink.mValue); // накопление стоимости
        HL:= Tree.GetNext as THamLink;
      end;
    end;

  var  Delta : integer;  // невязка, вычисляемая по числу листьев

  begin { Force }
    // Повторять, пока невязка больше нуля
    repeat
      // Форсировано снизить степени "узлов"
      ForceCoverTree;
      // Подсчитать степени вершин и невязку через количество листьев
      Delta:= CalcPowerNodes;
    until Delta=0;
    // Перенести цепь из буфера Buf в результат Result
    // и подсчитать стоимость цепи aCost
    Result:= TBuffer.Create;
    aCost:= MoveResult(Result);  //
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Преобразование буфера результата:
  // линки заменяются цепочкой вершин от Start до Fin

  function ConvertLinksToNodes(aLinks: TBuffer): TBuffer;

    // Поиск в буфере aLinks следующей за aNode вершины

    function FindNext(aNode: TNode): TNode;
    var Link: TLink;
        i: integer;
    begin
      Result:= nil;
      // Круговое вращение буфера в поиске нужного линка
      for i:= 1 to aLinks.GetCount do begin
        Link:= aLinks.Get as TLink;
        // Если линк найден, то цикл прекращается
        with Link do begin
          if aNode = mOwner then begin
            Result:= mDest;  Break;
          end;
          if aNode = mDest then begin
            Result:= mOwner; Break;
          end;
        end;
        // Если не тот линк, то вернуть его в буфер
        aLinks.Put(Link);
      end;
    end;

  var  Node: TNode;

  begin { ConvertLinksToNodes }

    Result:= TBuffer.Create;
    // Цепочка начинается с вершины aStart
    Node:= aStart;
    repeat
      Result.Put(Node);       // поместить в буфер
```

```
      Node:= FindNext(Node);  // и найти следующую
    until (Node = aFin) or not Assigned(Node);
    // Цепочка завершается вершиной aFin
    if Assigned(Node) then Result.Put(Node);
    aLinks.ClrAndDestroy;
    aLinks.Free;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    OK : boolean;        // признак связности графа
    Changed: boolean;    // признак снижения стоимости или невязки
    Delta : integer;     // невязка, вычисляемая по степеням вершин
    MinDelta : integer;  // текущая минимальная невязка
    Nodes: integer;      // количество оштрафованных вершин
    Cost: integer;       // текущая стоимость
    BestCost: integer;   // стоимость лучшего промежуточного решения
    Res: TBuffer;        // очередное решение (цепь)
    FineLimit: integer;  // предел штрафования вершины (счётчик)

begin  { TGraph.GenHamPath }

  aCost:= -1; // цена на случай отсутствия решения

  // Создать буфер для линков Гамильтоновой цепи:
  Result:= TBuffer.Create;
  // Если это орграф, то выход:
  if mDirect then Exit;

  // Назначить начальное значение для счётчиков штрафования вершин
  FineLimit:= mNodes.GetCount;

  // Создать:
  Que:= CreateSet;           // множество для очереди линков
  Tree:= CreateSet;          // буфер для линков дерева
  Marked:= TBuffer.Create;   // очередь вершин для маркировки

  BestCost:= MaxInt;  // начальная лучшая стоимость

  // Очистить штрафные поля mFines:
  ClearFines;

  // Назначить максимальные штрафы (mDist) для крайних вершин:
  aStart.mDist:= MaxInt div 4;
  aFin.mDist:= aStart.mDist;

  MinDelta:= MaxInt;        // минимальная невязка

  // Повторять, пока можно оштрафовать вершины
  repeat
    // Построить кратчайшее покрывающее дерево с учётом штрафов:
    OK:= GenCover;
    // Выход, если граф не связан:
    if not OK then Break;
    // Подсчитать невязку Delta как сумму листьев - 2
    Delta:= CalcPowerNodes;
    // Подсчитать новые штрафов
    // на основе текущих степеней вершин (полей mPower)
    // вернуть количество оштрафованных вершин Nodes
    Nodes:= CalcFines;
    Changed:= false;   // признак снижения стоимости или невязки
    // Искать форсированное решение
    Res:= Force(Cost);
```

```
    // Если оно лучше текущего, то запомнить, а иначе удалить
    if Cost < BestCost then begin
      Changed:= true;    // признак снижения стоимости или невязки
      BestCost:= Cost;   // запомнить лучшую стоимость
      aCost:= Cost;      // и текущую стоимость
      Result.Free;       // прежний результат удалить
      Result:= Res;      // а новый сохранить
    end else Res.Free;   // а иначе удалить новый
    // При уменьшении невязки
    if Delta < MinDelta then begin
      Changed:= true;        // признак снижения стоимости или невязки
      MinDelta:= Delta;      // запомнить минимальную невязку
    end;
    // Если уменьшилась стоимость или невязка
    if Changed then begin
      SetLimits(FineLimit); // возобновить количество попыток штрафования
      FineLimit:= 2 + 3*FineLimit div 4; // пересчитать лимит штрафования
    end;
  until Nodes = 0;  // прекратить, если штрафование не выполнялось
  // По окончании преобразовать буфер результата:
  // линки --> в вершины от Start ... до Fin
  if OK then Result:= ConvertLinksToNodes(Result);
  // Очистить память:
  Marked.Free;         // очередь вершин для маркировки
  Tree.ClrAndDestroy; // остовное дерево
  Tree.Free;
  Que.Free;            // очередь линков
end;
```

Далее обсуждаются некоторые детали листинга.

32.7.3. Функция GenCover

Эта функция строит минимальный остов графа с учётом текущих штрафов, наложенных на вершины. Функция возвращает **TRUE**, если остов построен. В случае, когда параметром **aStrip** задана исключаемая вершина, функция заранее окрашивает её в запрещающий красный цвет, и тогда эта вершина вместе с инцидентными ей рёбрами, игнорируются.

32.7.4. Процедура Mark

Вызывается всякий раз при «разрыве» очередного ребра остова для маркировки вершин двух получаемых при этом поддеревьев. Одно из них метится вершиной-источником линка, второе — вершиной приёмником. Параметр процедуры содержит одну из этих вершин.

32.7.5. Функции CalcFines, CalcNodeFine и CalcLinkFine

Подсчитывают оптимальные штрафы соответственно либо всех вершин, либо отдельной вершины, либо отдельного линка.

32.7.6. Функция CalcDeltaLink

Возвращает кратчайший белый линк, соединяющий листья двух поддеревьев в процедуре «разрубания» узлов.

32.7.7. Функция Force и процедура ForceCoverTree

На каждой итерации принудительно «разрубают» узлы дерева.

32.7.8. Функция ConvertLinksToNodes

Вызывается на завершающем этапе для преобразования буфера результата **Tree**, где в ходе решения задачи формируется множество рёбер Гамильтонова пути. Это множество рёбер надо затем превратить в последовательность вершин от **aStart** до **aFin**.

32.8. Метод GenHamPath

Этот метод использует основной метод, и удобен тогда, когда исключаемая вершина **aStrip** не задана (листинг 32-5).

Листинг 32-5 — Функция поиска Гамильтонова пути без исключённой вершины

```
function TGraph.GenHamPath(aStart,              // начальная вершины
                           aFin: TNode;         // конечная вершины
                           var aCost: integer   // длина цепи
                           ): TBuffer;
begin
  Result:= GenHamPathStrip(aStart, aFin, nil, aCost);
end;
```

32.9. Программы для испытаний

Ниже представлены программы для сравнительных испытаний обсуждённых ранее алгоритмов построения Гамильтоновых путей, а именно: 1) случайного перебора, 2) жадного перебора, 3) полного перебора и 4) метода штрафования вершин. Алгоритмы сравниваются по точности результата и скорости. Результаты зависят и от характеристик обрабатываемых графов. Так, например, для графов с одинаковыми рёбрами все алгоритмы сформируют равноценные пути, — этот случай не интересен. Полезней испытать методы на полных графах, длины рёбер которых равномерно распределены на интервале от **1** до **99**.

Первая программа (листинг 32-6) сравнивает эффективность четырёх алгоритмов на отдельных неориентированных графах. Здесь графы формируются случайно, но могут быть взяты также из некоторого файла.

Листинг 32-6 — Программа сравнения алгоритмов на отдельных графах

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res: TBuffer;
    Cost: integer;
    Na, Nb, Nc : TNode;
    Start : TDateTime;
    Nodes, Cost1, Cost2, Time : integer;

// Преобразование буфера вершин в строку

function HamToStr(aBuf: TBuffer): string;
var N: TNode;
    i: integer;
begin
  Result:='';
  for i:= 1 to aBuf.GetCount do begin
    N:= aBuf.Get as TNode;
    aBuf.Put(N);
    Result:= Result+ N.GetName;
    if i<aBuf.GetCount then Result:=Result +'-';
  end;
end;

begin
  repeat
    Write(#10#10'Nodes= '); Readln(Nodes);
    if Nodes < 4 then Break;
    {
    Сформировать полный граф со случайным распределением длин рёбер
    в диапазоне от 1 до 99 (равномерное распределение)
    }
    Gr:= TGraphChars.GenFull(false, 0, 99, Nodes);
    if Nodes < 16 then Gr.Expo;
    {
    В качестве исходной и конечной вершин взять первую (A) и вторую (B)
    В качестве исключаемой вершины может быть взята вершина C
    }
    Na:= (Gr as TGraphChars).GetNode('A');
    Nb:= (Gr as TGraphChars).GetNode('B');
    //Nc:= (Gr as TGraphChars).GetNode('C');
    Nc:= nil;

    Writeln('========== Ham Path ============'#10);
    Start:= Now;
    //Res:= Gr.GenHamPath(Na, Nb, Cost);
    Res:= Gr.GenHamPathStrip(Na, Nb, Nc, Cost1);
    Time:= MilliSecondsBetween(Start, Now);
    Writeln(HamToStr(Res));
    Writeln('Cost Ham  = ', Cost1:4, Time:7, #10);
```

```
    Res.Free;

    if Nodes < 18 then begin
      Writeln('========== Ham Path Full ============'#10);
      Start:= Now;
      Res:= Gr.GenHamPath_Full(Na, Nb, Cost2);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Writeln('Cost Full = ', Cost2:4, Time:7, #10);
      Res.Free;
      Writeln('Ratio= ', 100*Cost1/Cost2 : 7:1);
    end;

    Writeln(#10'========== Ham Path Greed ============'#10);
    Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, true); // жадно
    Writeln('Cost Greed = ', Cost);
    Res.Free;

    Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, false); // случайно
    Writeln('Cost Randm = ', Cost);
    Res.Free;
    Gr.Free;
  until false;
end.
```

Реалистичную картину можно получить обработкой большого количества графов с накоплением статистики, что выполняет следующая тестовая программа (листинг 32-7). Она определяет среднее время работы каждого метода, а также качество результата как отношение стоимости пути к стоимости идеального пути, полученного полным перебором. Для метода штрафования вершин формируется ещё и массив отклонений результата от идеала, — массив пригоден для построения гистограммы.

Листинг 32-7 — Программа сравнительных испытаний на большом числе графов (формируется статистика)

```
{$APPTYPE CONSOLE}
uses
  SysUtils,  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CPc = 2;       // шаг диаграммы в %
      CArray = 20;   // 2*20 = 40 %

var Gr   : TGraph;
    Res  : TBuffer;
    Cost: integer;
    i,k  : integer;

    Start : TDateTime;
    Time  : integer;

    Nodes : integer;  // количество вершин
```

```
    Retry : integer;  // количество повторений

    // Стоимости (%):
    CF : integer;   // полный перебор (100%)
    CQ : extended;  // быстрый
    CG : extended;  // жадный
    CR : extended;  // случайный

    // Времена:
    TF : integer; // полный перебор
    TQ : integer; // быстрый
    TG : integer; // жадный
    TR : integer; // случайный

    Arr : array [1..CArray] of integer; // массив для гистограммы
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вывод статистики
procedure ResultExpo(const aFile: String);
var n: integer;
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('-----------------------------------------------------------');
  Writeln('  Nodes=', Nodes:4, '  Retry=', Retry:4);
  Writeln('            %       ms');
  Writeln('Main =   100.0',           TF/Retry:7:0);
  Writeln('Quick= ',100*CQ/Retry:7:1, TQ/Retry:7:0);
  Writeln('Greed= ',100*CG/Retry:7:1, TG/Retry:7:0);
  Writeln('Rand = ',100*CR/Retry:7:1, TR/Retry:7:0);
  for n:= 1 to CArray do Writeln(n*CPc:3,#9, Arr[n]);;
  Writeln('-----------------------------------------------------------');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var    Na, Nb : TNode; // начальная и конечная вершины

begin
  repeat
    Write('Nodes= '); Readln(Nodes);
    if Nodes<5 then Break;
    repeat
      Write('Retry= '); Readln(Retry);
      if Retry=0 then Break;
      // Очистка времён и стоимостей
      CQ:=0; CG:=0; CR:=0;
      TF:=0; TQ:=0; TG:=0; TR:=0;
      // Массив гистограммы
      for k:= 1 to CArray do Arr[k]:=0;

      for i:= 1 to Retry do begin
        Write('.');
        Gr:= TGraphChars.GenFull(false, 0, 99, Nodes);
        Na:= (Gr as TGraphChars).GetNode('A');
        Nb:= (Gr as TGraphChars).GetNode('B');
        //-----------------------------------------
        // Полный перебор
        Start:= Now;
        Res:= Gr.GenHamPath_Full(Na, Nb, Cost);
```

```
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CF:= Cost; Inc(TF, Time);
        //-------------------------------------------
        // Ускоренный приближённый (штрафование вершин)
        Start:= Now;
        Res:= Gr.GenHamPath(Na, Nb, Cost);
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CQ:= CQ + Cost/CF; Inc(TQ, Time);
        // k - индекс в массиве диаграммы
        k:= 1 + Round(100*(Cost/CF-1)/CPc);
        if k in [1..CArray] then Inc(Arr[k]);

        //-------------------------------------------
        // Жадный
        Start:= Now;
        Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, true);
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CG:= CG + Cost/CF; Inc(TG, Time);
        //-------------------------------------------
        // Случайный
        Start:= Now;
        Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, false);
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CR:= CR + Cost/CF; Inc(TR, Time);
        //-------------------------------------------
        Gr.Free;
      end;
      Writeln;
      ResultExpo('');
      ResultExpo('Test_Time.txt');
    until false;
  until false;
end.
```

32.10. Результаты испытаний

В ходе испытаний обработаны по сотне случайных графов с количеством вершин от 10 до 17, результаты даны на следующих рисунках и в таблицах. В табл. 32-1 сопоставляется точность разных методов в сравнении с идеалом, получаемым полным перебором всех путей. Здесь быстрым алгоритмом назван метод штрафования вершин, он находит либо точные, либо очень близкие к идеалу пути, средняя цена которых лишь на **0,5%** превышает идеальную. В этом отношении жадный алгоритм значительно хуже метода штрафования, но существенно лучше случайного (см. рис. 32-13). Отметим, также, что качество этих двух алгоритмов — жадного и случайного — ухудшается с ростом размера графа.

Табл. 32-1 — Отношение стоимостей путей к идеалу, %

Количество вершин	Алгоритм		
	Быстрый	Жадный	Случайный
10	100,5	134	256
11	101,0	133	280
12	100,5	140	305
13	100,7	140	328
14	100,3	147	356
15	100,6	150	396
16	100,3	155	392
17	100,3	153	425
СРЕДНЕЕ ЗНАЧЕНИЕ	*100,5*	*144,0*	*342,3*

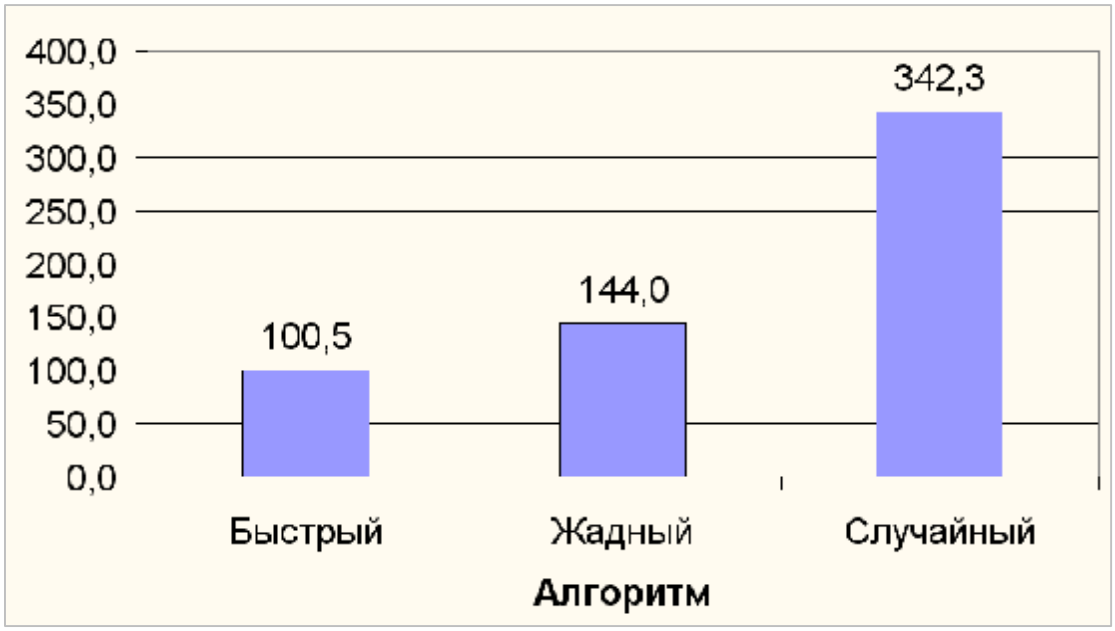


Рис. 32-13 — Отношение стоимостей путей, полученных разными алгоритмами, к стоимости идеального пути (%)

На рис. 32-14 дана гистограмма распределения стоимостей результатов быстрого алгоритма в сравнении с идеалом, — таблица получена по результатам 800 испытаний. Здесь видно, что в **87%** случаев результаты быстрого алгоритма уступают идеалу не более **2%**, а в **97%** случаев — не более **6%**, причём большинство из этих результатов точные.

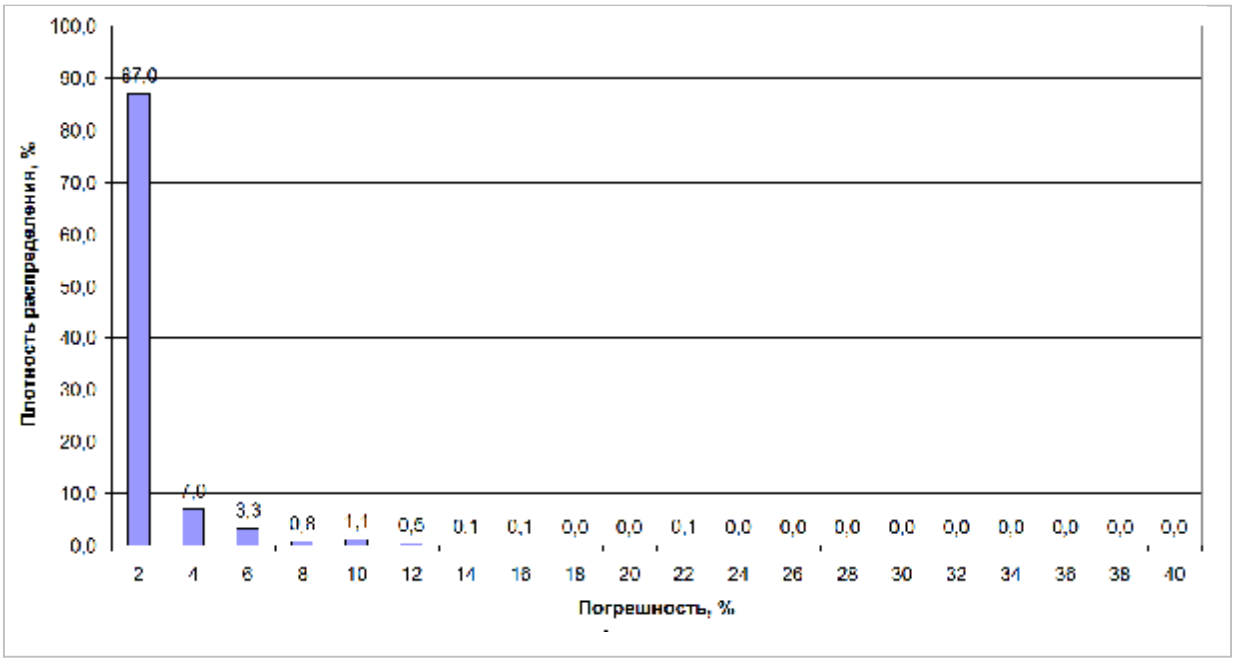


Рис. 32-14 — Плотность распределения результатов для быстрого алгоритма

Точность метода штрафования вершин вполне приемлема, а что со временем? В табл. 32-2 представлены средние времена обработки графов точным, но медленным алгоритмом полного перебора, и быстрым методом штрафования вершин. Даны соотношения этих времён. То же показано на рис. 32-15 и рис. 32-16.

Табл. 32-2 — Среднее время (ms) обработки одного графа полным перебором и быстрым методом, а также отношения времён

Количество вершин	Алгоритмы и отношение		
	Полный перебор T1, ms	Быстрый T2, ms	Отношение T1/T2
10	1,0	2	0,5
11	6,0	2	3,0
12	20,0	3	6,7
13	68,0	5	13,6
14	227,0	7	32,4
15	756,0	9	84,0
16	3002,0	13	230,9
17	6810,0	17	400,6

Эти данные демонстрируют ценность быстрого алгоритма штрафования вершин, преимущество которого с ростом размера графа растёт (рис. 32-16). При обработке отдельных графов полным перебором наблюдается, также, большой разброс времени для графов с одинаковым количеством вершин (в разы). Время обработки быстрым алгоритмом для всех таких графов стабильно.

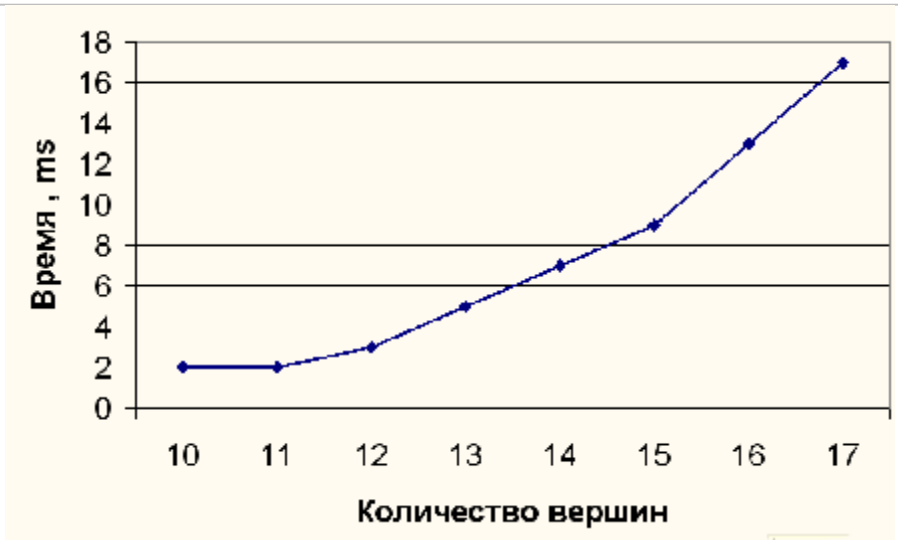


Рис. 32-15 — Среднее время обработки графа
быстрым методом штрафования вершин

Рис. 32-16 — Отношение скорости быстрого алгоритма
к скорости точного алгоритма (T1/T2)

32.11. Итоги

- В задачах коммивояжёра и Гамильтона ищутся кратчайшие (или длиннейшие) пути посещения *всех* вершин графа.
- В задачах коммивояжёра *разрешены* повторные посещения вершин, а в задачах Гамильтона каждая вершина посещается *единожды*.
- Задачи посещения делят на *замкнутые* и *разомкнутые*.
- В замкнутых задачах путь — цикл — заканчивается в исходной вершине, тем самым исходной может быть любая вершина.
- В разомкнутых задачах путь начинается и заканчивается в разных вершинах. Иногда интерес представляет пара вершин, путь между которыми минимален (или максимален).
- Задачи коммивояжёра всегда имеют решения на связных графах и сильно связных орграфах, для задач Гамильтона этих условий недостаточно.
- На *полных* графах всегда решаются и задачи коммивояжёра, и задачи Гамильтона.
- Задачи коммивояжёра решают через задачи Гамильтона созданием вспомогательного полного графа, с последующим возвратом к исходному графу.
- *Точное* решение задачи Гамильтона может быть получено перебором всех возможных путей, однако ввиду экспоненциально растущей трудоёмкости этот способ применим лишь для небольших графов.
- *Быстрое*, но грубое решение даёт жадный алгоритм, выбирающий на каждом шаге кратчайшее ребро (дугу), ведущее в ещё не посещённую вершину.
- Весьма *точное* и относительно *быстрое* решение для *неориентированных* графов даёт метод штрафования вершин. В его основе лежит многократное применение алгоритма Прима, строящего минимальный остов графа. Штрафуя вершины (взаимно отдаляя их друг от друга) алгоритм деформирует граф так, что кратчайший его остов становится одновременно либо Гамильтоновым путём, либо значительно приближается к нему по стоимости.

32.12. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 265
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 33
Замкнутая задача Гамильтона

Продолжим вникать в проблемы Гамильтона и обратимся к *замкнутому* варианту этой задачи. Здесь начальной и конечной точкой пути является любая вершина графа, стало быть, решением будет *кратчайший* цикл. Отличие от разомкнутого варианта задачи, казалось бы, пустяшное — всего одно лишнее ребро или дуга, однако эта мелочь в корне меняет подход к решению.

33.1. Опорные алгоритмы

Как и для разомкнутого варианта задачи, для оценки эффективности создаваемого алгоритма применим упрощённые процедуры: либо *быстрые*, либо *точные*. Эти процедуры аналогичны процедурам для разомкнутого варианта задачи, и потому здесь приведено лишь их перечисление, полные тексты содержатся в листинге модуля Graph, приложение G:

`GenHamilton_Greed` — с параметром **`FALSE`** этот метод случайно выбирает очередную дугу, ведущую в неокрашенную вершину. Алгоритм быстрый, но не точный.

`GenHamilton_Greed` — с параметром **`TRUE`** выбирает кратчайшую дугу, ведущую в неокрашенную вершину (жадный алгоритм). По скорости мало уступает случайному, но даёт существенно лучший результат, хотя и далёкий от идеального.

`GenHamilton_Full` — полный перебор всех возможных путей, даёт идеально точный результат, но крайне трудоёмок, и потому применим лишь для небольших графов.

Отметим, также, что случайный и жадный алгоритмы гарантируют решение лишь на *полных* графах.

33.2. Цикл, поток и паросочетание

Основные идеи предстоящего решения демонстрирует рис. 33-1, где показан полный орграф с уже обнаруженным гамильтоновым циклом минимальной стоимости. Этот цикл выделен жирным, а все прочие дуги графа обозначены серым.

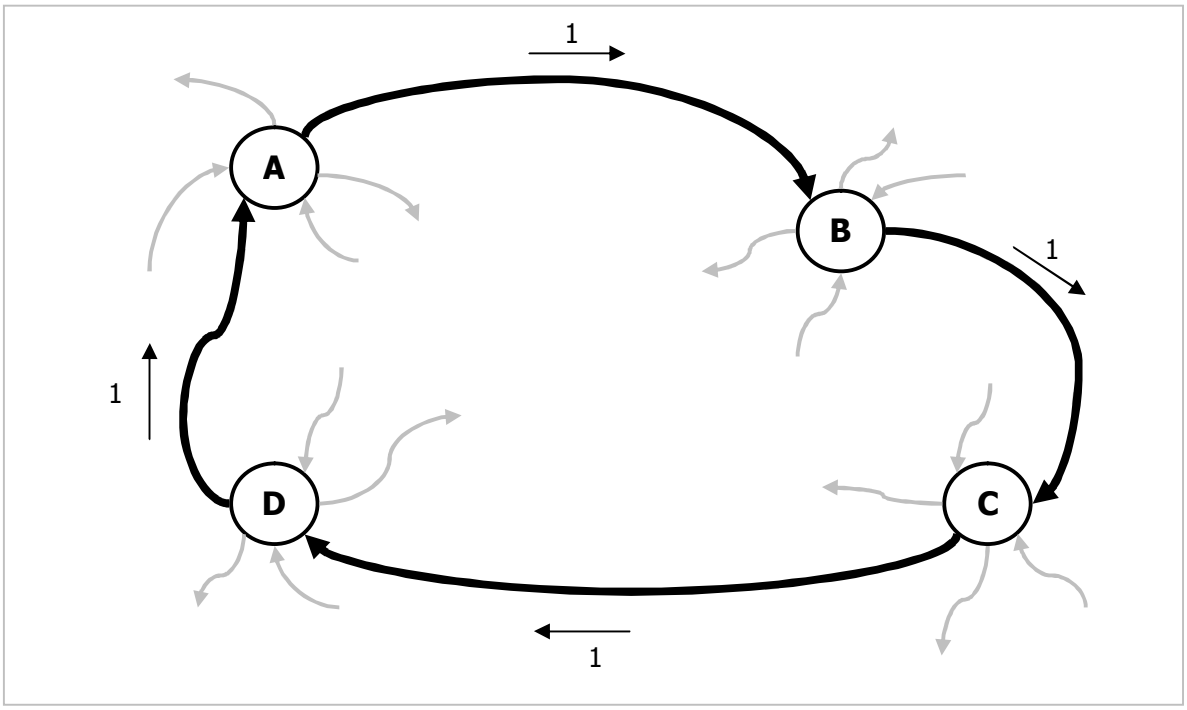


Рис. 33-1 — Единица потока, циркулирующая по замкнутому пути

Пустим по замкнутому пути единичный поток. Что можно сказать о его стоимости? Очевидно, что эта стоимость будет минимальной в сравнении с любым другим замкнутым потоком, проходящим через все вершины графа по иным дугам. Стало быть, задача поиска минимального Гамильтонова цикла родственна задаче поиска минимального потока в сети. Вопрос лишь в том, как найти этот минимальный поток в *замкнутом цикле*. Сейчас силой волшебства сведём эту задачу к поиску минимального потока в двудольном графе.

Разделим каждую вершину исходного графа на две половины, одна из которых (**S**) будет содержать только исходящие дуги и служить источником потока, а другая (**T**) — только входящие дуги, и будет принимать поток (рис. 33-2). Вновь полученный граф окажется двудольным, и количество вершин в нём будет вдвое больше, чем в исходном (рис. 33-3). В 27-й главе показаны два способа поиска потока минимальной стоимости в таком графе. Один из них заключается в добавлении ещё двух вершин: общем истоке *S* и общем стоке *T*, которые соединяются со всеми прочими вершинами графа так, как показано на рис. 33-3. Там же показано, что эта задача равнозначна поиску минимального паросочетания, и быстрее решается *венгерским* алгоритмом. Итак, найдя полное паросочетание минимальной стоимости на построенном двудольном графе, тем самым найдём и цикл минимальной стоимости в исходном графе.

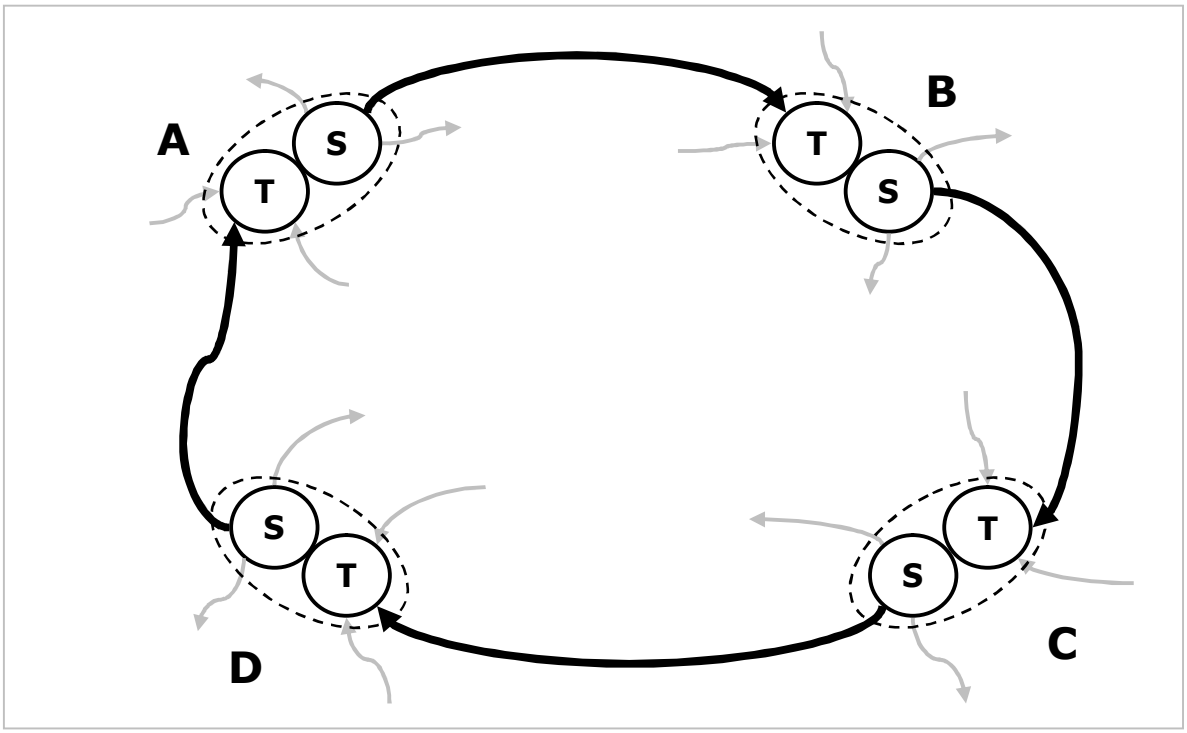


Рис. 33-2 — Раздвоение вершин исходного графа

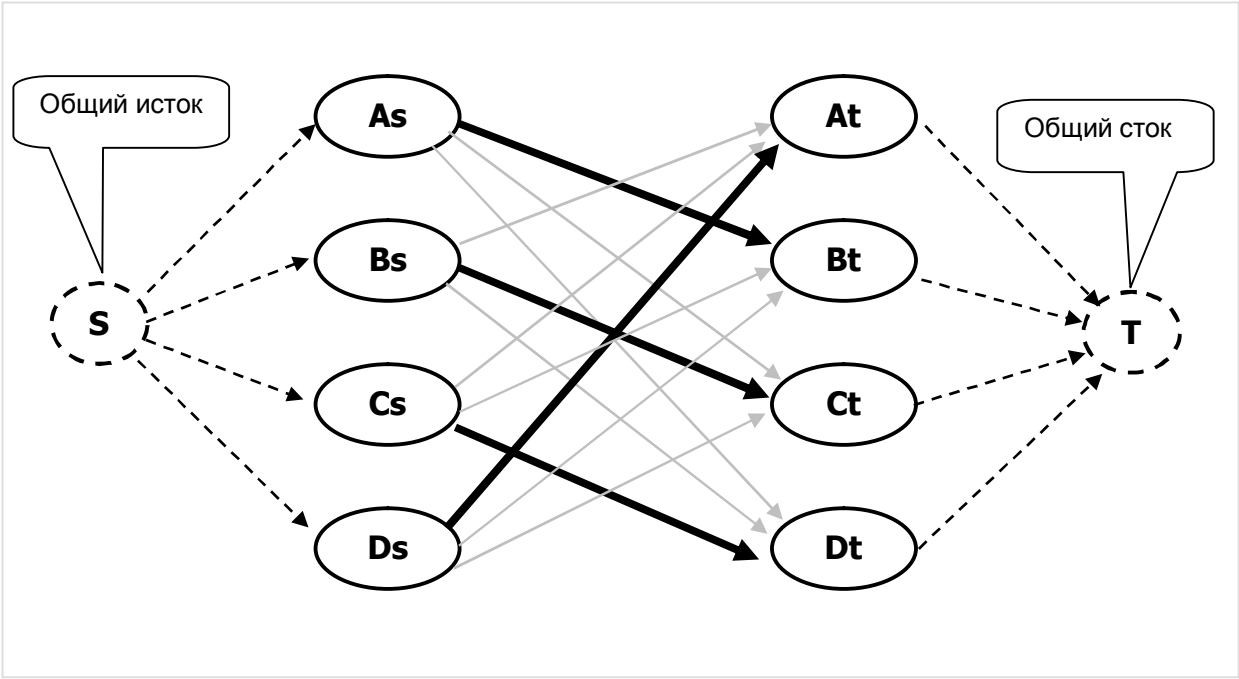


Рис. 33-3 — Эквивалентный двудольный граф

33.3. Не цикл, а циклы

Впрочем, с минимальным потоком всё не так просто. Да, венгерский алгоритм даёт поток минимальной стоимости в двудольном графе. Но, отобразив его на исходный граф, можно получить не один, а несколько замкнутых потоков (см. рис. 33-4, где граф содержит 8 вершин). Эти не связанные циклы в совокупности удовлетворяют только одному из требований к циклу Гамильтона — дают множество дуг или рёбер *минимальной* стоимости. Но не отвечают второму

требованию: цикл должен быть *единственным*. И для достижения этой второй цели волшебства уже не достаточно.

Рассмотрим рис. 33-4, где поиск минимального потока дал два несвязанных цикла (эти циклы называют *факторами*). Можно ли объединить эти циклы в один? Разорвём (удалим) дугу одного из циклов, и тогда после повторного поиска минимального потока, инцидентные ему вершины поневоле соединятся в циклы как-то иначе, тем самым повышая вероятность получения *единственного* фактора. Эта вероятность повысится ещё больше, если у какой либо вершины разорвать *все* дуги, ведущие в вершины *её* текущего цикла, — тем самым при повторном поиске гарантируется её присоединение к другому циклу. Так, например, для вершины *B* можно разорвать дуги *B-A*, *B-C* и *B-D*, и тогда вершина *B* перекочует в другой цикл (рис. 33-4). Разумеется, что применение этой процедуры только к одной вершине не гарантирует получения единственного цикла, и тогда придётся поочерёдно применить её к нескольким вершинам. Но, даже получив единственный цикл, нельзя быть уверенным в том, что он минимален, — подобных циклов может быть несколько, и надо перепробовать отключения всех вершин графа. К чему это ведёт?

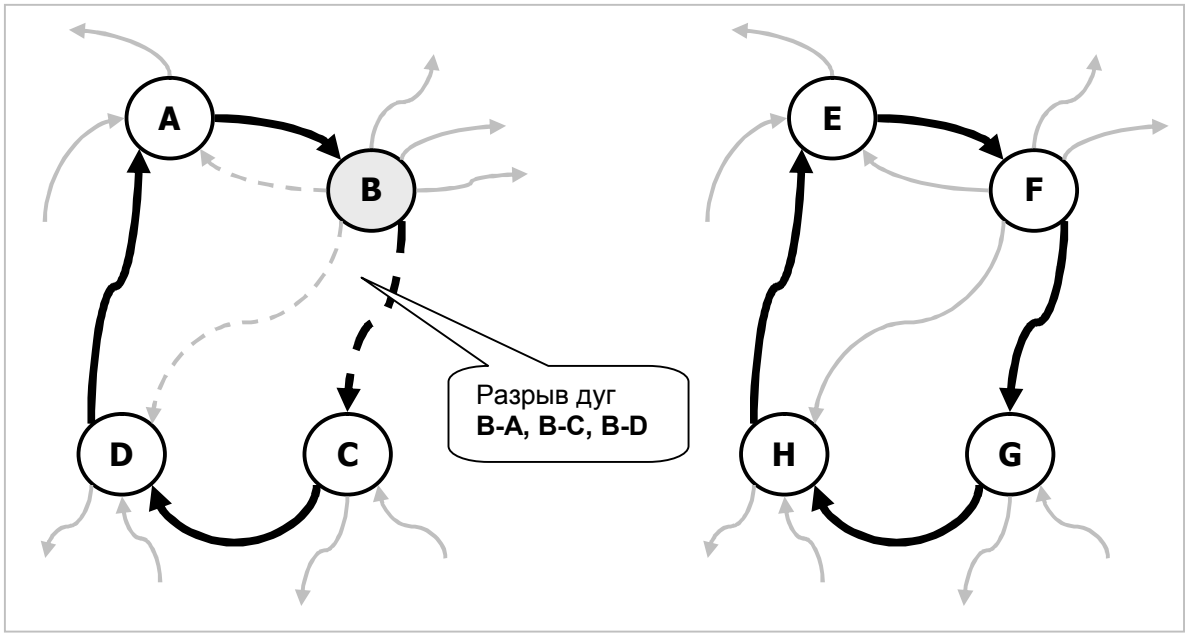


Рис. 33-4 — Минимальный поток, разбитый на два цикла (фактора), пунктиром показаны разрываемые дуги для вершины *B*

Пусть первый поиск минимального потока породил несколько не связанных факторов. Будем поочерёдно применять выше описанную процедуру разрыва дуг ко всем вершинам графа. Разорвав дуги одной из вершин, найдём соответствующую этому состоянию дуг новую конфигурацию *факторов*. Если фактор (цикл) оказался единственным, значит, получен кандидат на Гамильтонов цикл минимальной стоимости, — запомним его и его стоимость. Если же в полученной конфигурации оказалось несколько факторов, но стоимость соответствующего ей потока оказалась меньше текущего минимума, то с этой конфигурацией можно работать дальше и её следует как-то запомнить. Этим заканчивается обработка выбранной вершины, после чего восстанавливается

состояние её дуг, и берётся следующая вершина графа. Таким образом, обработка каждой вершины даёт один из двух результатов:

- поток с единственным фактором (циклом) — это *очередное* решение; если его стоимость оказалась ниже текущего минимума, то это решение и его стоимость запоминаются в качестве лучшего;
- поток с несколькими факторами (циклами) — это *промежуточное* решение; если его стоимость оказалась ниже текущего минимума, то сохраняется шанс на улучшение итогового решения, и тогда эта конфигурация вместе с состоянием дуг сохраняется в приоритетной очереди для дальнейшей обработки.

Когда решение с единственным фактором находится не сразу, или когда его начальная стоимость далека от лучшей, тогда порождается и ставится в приоритетную очередь много промежуточных решений. Эта очередь кандидатов растёт геометрически. В конечном счёте, погоня за точным решением может привести к неприемлемым затратам памяти и времени. И потому далее рассмотрим меры, сокращающие объём вычислений.

33.4. Минимальный фактор

Первая мера состоит в том, чтобы обрабатывать не все вершины, а только те, что образуют кратчайший фактор (или один из кратчайших, если таких факторов несколько). Под кратчайшим понимаем фактор, содержащий наименьшее количество вершин. Практика показала, что тем самым стоимость окончательного решения ухудшается незначительно, а затраты времени и памяти сокращаются кардинально. Технически выделить минимальный фактор не сложно в ходе подсчёта факторов.

33.5. Уточнённый прогноз стоимости

Вторая мера состоит в более точной оценке прогнозируемой стоимости, достигаемой для промежуточного решения. В качестве этой нижней границы проще всего взять сумму стоимостей потока в несвязанных циклах. Однако эта оценка слишком оптимистична, — реальная стоимость единственного цикла будет несколько выше, поскольку объединение факторов сопряжено с повышением стоимости потока (и уж точно не с её понижением). Более точная оценка нижней границы позволяет отбросить многих кандидатов на последующую обработку, и установить более точную очерёдность обработки этих кандидатов (более перспективные обрабатываются раньше).

Базой уточнённого прогноза будет всё та же суммарная стоимость потока в несвязных циклах, обозначим её как $C0$, — она находится после отключения части дуг некоторой вершины. Поскольку в последующем эти факторы будут соединены другими дугами, нижняя граница приращения стоимости этого соединения ($C1$) может быть найдена поиском потока минимальной стоимости на вспомогательном полном графе, полученном *стягиванием факторов* в псевдо-вершины. Эти

псевдо-вершины соединяются дугами, длины которых выбираются так, чтобы соединить псевдо-вершины кратчайшими путями. Для этого берутся остаточные (редуцированные) стоимости дуг, полученные после решения задачи о минимальном паросочетании венгерским методом, — их подсчёт уже предусмотрен венгерским алгоритмом (глава 27). Итак, стянув факторы в псевдо-вершины и соединив их дугами минимальной (остаточной) стоимости, получим полный граф. Перед вычислением в этом вспомогательном графе потока минимальной стоимости необходима ещё одна операция, — *компрессия дуг*. Она заменяет прямые дуги кратчайшими путями между вершинами, если стоимость этих цепочек будет ниже стоимости прямых дуг (правило треугольника не работает).

На этом полном графе с компрессированными дугами вновь находим минимальный поток всё тем же венгерским методом, его цена даст приращение прогноза *C1*, а в целом прогноз составит сумму *C0 + C1*. Если во вспомогательном графе поток будет образован одним фактором, то вычисление прогноза завершится. Если же факторов опять окажется больше одного, то все предыдущее действия рекурсивно повторяются на вновь полученном графе: стягивание факторов в псевдо-вершины, построение полного графа, компрессия дуг, вычисление стоимости потока *C2* и добавление её к оценке. В худшем случае этот процесс закончится за время, пропорциональное логарифму от количества вершин, а реально гораздо быстрее. На первый взгляд алгоритм кажется громоздким и затратным, но на деле он рекурсивен и не сложен в реализации. В дальнейшем вернёмся к нему, рассмотривая числовой пример в пункте 33.9. Отметим, что эффект от этой меры весьма заметен на неориентированных графах и симметричных орграфах.

33.6. Общее описание основного алгоритма

Рассмотрев основные идеи будущего метода поиска минимального Гамильтонова цикла (назовём этот метод *основным*), сформулируем его в целом. Для упрощения формулировок введём объект, хранящий промежуточные решения. В таком объекте будет храниться следующая информация:

- прогнозируемая стоимость решения;
- текущее множество отключенных дуг;
- множество вершин кратчайшего фактора.

Учредим также *приоритетную* очередь, в которой выше названные объекты с промежуточными решениями будут храниться в порядке не убывания их прогнозируемой стоимости. Тогда алгоритм поиска минимального Гамильтонова цикла представится так (все действия выполняются на двудольном графе):

1. Назначить заведомо большую стоимость текущего лучшего решения.
2. Создать объект, хранящий фиктивное промежуточное решение, прогнозируемая стоимость которого равна нулю, множество отключенных дуг пусто, а множество вершин кратчайшего фактора содержит одну (любую) вершину. Поместить этот объект в очередь.
3. Пока приоритетная очередь не пуста:
   3.1. Выбрать первый объект, хранящий наименее дорогое промежуточное решение.
   3.2. Если его прогнозируемая стоимость меньше текущего решения, то обработать этот объект следующим образом:
      3.2.1. Отключить дуги, указанные в множестве отключенных дуг.
      3.2.2. Для каждой вершины кратчайшего фактора:
         3.2.2.1. Отключить дуги, ведущие из этой вершины внутрь фактора.
         3.2.2.2. Найти минимальный поток и его стоимость (окончательную или прогнозируемую), а также количество факторов и кратчайший фактор.
         3.2.2.3. Если полученная стоимость оказалась меньше текущего минимума, и если количество факторов равно единице, то запомнить это решение в качестве лучшего, а иначе сохранить прогноз стоимости, множество отключенных дуг и кратчайший фактор в объекте хранения и поместить этот объект хранения в приоритетную очередь.
         3.2.2.4. Подключить дуги, ведущие из текущей вершины внутрь цикла (вернуть состояние ранее отключенных дуг).
      3.2.3. Подключить дуги, указанные в множестве отключенных дуг (вернуть состояние).
   3.3. Уничтожить объект хранения, выбранный из очереди.

Итак, обработку графа начинаем в состоянии, когда все дуги подключены, активны. Если в этом состоянии двудольного графа циркулирующий поток минимальной стоимости будет образован одним фактором, то решение найдено. Иначе для кратчайшего фактора создаём несколько объектов хранения промежуточных решений, и ставим их в очередь. Отметим, что с момента, когда стоимость лучшего решения окажется ниже прогнозируемой стоимости первого объекта в очереди, все оставшиеся объекты в очереди будут ликвидированы.

Приступим к деталям этого алгоритма, и для начала рассмотрим вспомогательные механизмы и объекты, принимающие на себя часть его функций.

33.7. Вспомогательные объекты

33.7.1. Механизм отключения дуг

Отметим, что отключать и вновь включать те или иные дуги графа придётся очень часто, и потому желательно избежать при этом перестройки графа. С этой целью будет использовано поле **`TLink.mHigh`**: ноль соответствует включенной

дуге, единица — отключенной. Венгерский алгоритм будто бы не замечает отключенных дуг, игнорирует, — такое поведение уже предусмотрено в реализации венгерского метода (см. **TGraph.MarkMinPairsDicoty**).

33.7.2. Объект для хранения промежуточного решения

Этот объект хранит состояние графа на момент получения очередного промежуточного решения, когда количество факторов отлично от единицы. Поскольку это состояние характеризуется главным образом совокупностью закрытых и закрываемых дуг, объекту дано название **TClosed**. Назначение полей объекта таково:

Поле **mCost** хранит прогнозируемую стоимость потенциального решения, оно определяет положение элемента в приоритетной очереди так, что наименее дорогое и перспективное решение оказывается в начале очереди.

Уникальное поле **mId** необходимо для различения объектов с одинаковой прогнозируемой стоимостью, иначе дубликаты отбрасывались бы.

Поле **mClosed** содержит множество дуг, которые были закрыты на момент получения решения, — они останутся закрытыми и в последующих модификациях.

Поле **mFactors** по идее могло бы хранить множество вершин кратчайшего фактора (как было сказано выше), но технически удобней хранить здесь множество множеств дуг, которые будут закрываться в продолжение решения. Таким образом, каждой вершине кратчайшего фактора здесь соответствует одно множество закрываемых дуг, а количество этих множеств равно количеству вершин в этом факторе.

Листинг 33-1 — Объект, хранящий промежуточное решение

```
type  TClosed = class (TItem)
        mCost: integer;  // стоимость потока
        mId: integer;    // уникальный идентификатор
        mClosed: TSet;   // множество закрытых дуг (клапанов)
        mFactors: TSet;  // множества отключаемых дуг кратчайшего фактора
        constructor Create(aCost: Integer; aClosed: TSet);
        destructor Destroy;  override;
        function Compare(arg: TItem): TCompare; override;
        procedure InsertLinksSet(aSet: TSet);
        procedure GatesOn;
        procedure GatesOff;
        procedure Print(var aFile: TextFile); override;
      end;

var ClosedId : integer; // уникальный идентификатор элемента очереди
                        // наращивается в конструкторе объекта

constructor TClosed.Create(aCost: Integer; aClosed: TSet);
begin
  inherited Create;
  Inc(ClosedId);     // наращиваем уникальный идентификатор
  mId:= ClosedId;    // и заносим в поле объекта
```

```
  mCost:= aCost;     // стоимость потока на момент создания объекта
  // Создаём и копируем множество закрытых дуг из параметра
  mClosed:= CreateSet;
  if Assigned(aClosed) then mClosed.CopyItems(aClosed);
  // Заготавливаем множество множеств закрываемых дуг
  // (по одному множеству для каждой вершины фактора).
  // В это множество будут вставляться множества закрываемых дуг
  // процедурой TClosed.InsertLinksSet
  mFactors:= CreateSet;
end;

destructor TClosed.Destroy;
begin
  mFactors.ClrAndDestroy;  // уничтожаются множества дуг
  mFactors.Free;
  mClosed.Free;
  inherited;
end;

// Добавляет очередное множество отключаемых дуг aSet

procedure TClosed.InsertLinksSet(aSet: TSet);
begin
  mFactors.Insert(aSet)
end;

// Метод сравнения выстраивает объекты в приоритетной очереди
// в порядке не убывания стоимости,
// а при равной стоимости - в порядке вставки
// путём сравнения уникальных идентификаторов

function TClosed.Compare(arg: TItem): TCompare;
begin
  if mCost < (arg as TClosed).mCost
    then Result:= cmpLess
  else if mCost > (arg as TClosed).mCost
    then Result:= cmpGreate
  else { при равенстве стоимостей сравниваем уникальные идентификаторы }
    if mId < (arg as TClosed).mId
      then Result:= cmpLess
    else if mId > (arg as TClosed).mId
      then Result:= cmpGreate
    else Result:= cmpEq
end;

// Отключает дуги, содержащиеся в mClosed

procedure TClosed.GatesOff;
var Link: TLink;
begin
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // дуга отключается единицей
      Link:= GetNext as TLink;
    end;
  end;
end;

// Включает дуги, содержащиеся в mClosed

procedure TClosed.GatesOn;
```

```
var Link: TLink;
begin
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 0; // дуга подключается нулём
      Link:= GetNext as TLink;
    end;
  end;
end;
```

33.7.3. Вспомогательный двудольный граф и его вершины

Этот вспомогательный граф, как сказано ранее, используется для поиска замкнутого потока минимальной стоимости венгерским методом. Граф несёт основную алгоритмическую нагрузку создаваемого метода. Ознакомимся с типом его вершин **THamNode**.

Этот тип (листинг 33-2) построен на основе **TNode**, стало быть, обладает всеми его полями и методами, но вдобавок содержит и дополнительные, а именно:

mLeft — признак принадлежности вершины к левой или правой доле графа;

mNode — ссылка на родовую вершину исходного графа; две вершины двудольного графа — левая и правая — ссылаются на общую родовую вершину;

mBack — поле для взаимных ссылок пары вершин: левая ссылается на правую, а правая на левую; используется для быстрого обхода циклов.

Унаследованные от **TNode** поля используются следующим образом:

mValue — содержит числовую метку фактора (цикла); все вершины одного фактора метятся одним числом;

mLink — содержит линк текущего паросочетания; эти линки указывают направление потока, полученного венгерским методом (применяется только в левых вершинах двудольного графа).

Листинг 33-2 — Описание типа вершин двудольного графа

```
type  THamNode = class (TNode)
        mLeft : boolean;     // признак левой вершины (TRUE)
        mNode : TNode;       // родовая вершина
        mBack : THamNode;    // ссылка справа налево и слева направо
        constructor Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
        function  Compare(arg: TItem): TCompare; override;
        function  GetName: string;  override;
        function  NextCircNode: THamNode;
        function  GenFactorsLinks: TSet;
      end;

constructor THamNode.Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
begin
  inherited Create(0, aOwner);
  mNode:= aNode;     // родовая вершина
  mLeft:= aLeft;     // сторона: TRUE - Left, FALSE - Right
end;

function  THamNode.GetName: string;
begin
  // Сторона + имя родовой вершины
  if mLeft then Result:= 'L.' else Result:= 'R.';
  Result:= Result + mNode.GetName;
end;

// Генерирует множество не отключенных дуг,
// ведущих внутрь текущего фактора (цикла)

function THamNode.GenFactorsLinks: TSet;
var Link: TLink;
begin
  Result:= CreateSet;
  Link:= OutLinkFirst;
  while Assigned(Link) do begin
    // Если дуга ведёт внутрь текущего цикла (фактора)
    // и она не отключена (mHigh = 0), то сохраняем в результате
    with Link do
    if ((mDest as THamNode).mBack.mValue = Self.mValue) and (mHigh = 0)
      then Result.Insert(Link);  // запомнить в множестве отключаемых
    Link:= OutLinkNext;
  end;
end;

// Возвращает следующую левую вершину текущего цикла
// в соответствии с текущим потоком

function THamNode.NextCircNode: THamNode;
begin
  Result:= (mLink.mDest as THamNode).mBack;
end;

// При вставке вершины сортируются по алфавиту

function  THamNode.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  if GetName < (arg as THamNode).GetName
    then Result:= cmpLess
```

```
  else if GetName > (arg as THamNode).GetName
    then Result:= cmpGreate
end;
```

Теперь рассмотрим основной объект — двудольный граф **THamGraph**, он основан на типе **TGraph** и конструируется на основе родового графа, указанного через параметр конструктора. Двудольный граф ориентирован слева направо.

Основной метод **CalcCostAndFactors** находит: a) минимальный поток (с учётом текущего состояния дуг), b) количество факторов, c) одну из вершин кратчайшего фактора, а также d) прогнозируемую или конечную стоимость потока (конечная стоимость соответствует единственному фактору).

Листинг 33-3 — Двудольный граф для поиска цикла Гамильтона

```
type  THamGraph = class (TGraph)
        mGraph: TGraph;   // исходный (родовой) граф
        constructor Create(aGraph : TGraph);
      protected
        function Check: boolean;
        function CalcFactors(var aStart: THamNode): integer;
        procedure SaveBest(aBuf: TBuffer);
        function CalcCostAndFactors(
                       var aFactors: integer; //кол-во факторов
                       var aStart: THamNode  // стартовая вершина
                       ): integer;           // стоимость потока
        function Shrink(aFact: integer): THugeGraph;
      end;

constructor THamGraph.Create(aGraph : TGraph);
var Node: TNode;
    HamL, HamR: THamNode;
    Link : TLink;
begin
  inherited Create('Hamilton',  // имя
                   true,        // дуги ориентированы
                   false,       // вершины не нагружены
                   true);       // дуги нагружены
  mGraph:= aGraph; // ссылка на исходный граф

  // На базе вершин исходного графа создаём пары специальных вершин:
  Node:= aGraph.NodeFirst;
  while Assigned(Node) do begin
    // Создаём пару вершин
    HamL:= THamNode.Create(Self, Node, True);  // исток
    HamR:= THamNode.Create(Self, Node, False); // сток
    // Формируем взаимные ссылки для ускорения доступа:
    HamL.mBack:= HamR;
    HamR.mBack:= HamL;
    // и вставляем вершины в этот граф
    mNodes.Insert(HamL);
    mNodes.Insert(HamR);
    Node:= aGraph.NodeNext;
  end;

  // Формируем дуги, ведущие из левой доли в правую:
  HamL:= NodeFirst as THamNode;
  while Assigned(HamL) do begin
    // Это левая вершина?
```

```
    if HamL.mLeft then  begin
      // Да, обрабатываем её
      PosPush;  // сохраним позицию перебора вершин
      // Перебираем исходящие дуги в вершине исходного графа
      Link:= HamL.mNode.OutLinkFirst;
      while Assigned(Link) do begin
        Node:= Link.mDest;  // целевая вершина исходного графа
        // Ищем соответствующую ей вершину в правой доле:
        HamR:= NodeFirst as THamNode;
        while Assigned(HamR) do begin
          with HamR do if not mLeft and (mNode = Node) then Break;
          HamR:= NodeNext  as THamNode;
        end;
        // Здесь HamR соответствует целевой вершине исходного графа
        // Создаём дугу из левой доли в правую (той же стоимости)
        HamL.MakeLink(HamR, Link.mValue);
        // Переход к следующей дуге исходного графа
        Link:= HamL.mNode.OutLinkNext;
      end; // while
      PosPop;  // восстановим позицию перебора вершин
    end; // if
    HamL:= NodeNext as THamNode;
  end; // while
end;

// Поиск и пометка в поле mValue всех факторов (циклов)
// Возвращает количество факторов.
// Принадлежность вершины к фактору определяется полем mValue

function THamGraph.CalcFactors(var aStart: THamNode): integer;
var NodesCnt: integer;  // счётчик оставшихся необработанных вершин
    Start, Left : THamNode; // стартовая и текущая вершины
    CircLen: integer;   // длина очередного фактора
    MinLen : integer;   // длина кратчайшего фактора
    Link : TLink;
begin
  Result:= 0;          // количество факторов
  MinLen:= MaxInt;     // длина кратчайшего фактора
  // количество вершин исходного графа:
  NodesCnt:= mGraph.mNodes.GetCount;
  // очистка полей mValue двудольного графа:
  Start:= NodeFirst as THamNode;
  while Assigned(Start) do begin
    Start.mValue:= 0;
    Start:= NodeNext as THamNode;
  end;
  // Пока не обработаны все вершины графа (исходного)
  while NodesCnt>0 do begin
    Inc(Result);  // Result= 1, 2, .. - цвет очередного цикла
    CircLen:=0;   // длина текущего фактора (цикла)
    // Ищем первую неокрашенную вершину в левой доле:
    Start:= NodeFirst as THamNode;
    while Assigned(Start) do begin
      with Start do if mLeft and (mValue = 0) then Break;
      Start:= NodeNext as THamNode;
    end;
    // Начиная с вершины Start обходим двудольный граф
    // отмечая цветом Result очередной замкнутый цикл:
    // => вправо - по направлению текущего потока;
    // <= влево  - по обратным ссылкам mBack
    // пока не вернёмся к исходной вершине
    Left:= Start;
```

```
    repeat
      Dec(NodesCnt);         // счётчик оставшихся вершин
      Inc(CircLen);          // длина этого фактора (цикла)
      Left.mValue:= Result; // метим цикл текущим цветом
      // Left.mLink -- дуга из вершины Left в вершину Right
      // через которую течёт ненулевой поток (чёрный линк паросочетания)
      Link:= Left.mLink;
      // переход в левую долю вдоль потока:
      Left:= (Link.mDest as THamNode).mBack;
    until Left = Start;
    // После обхода цикла запоминаем кратчайший:
    if CircLen < MinLen then begin
      MinLen:= CircLen;    // длина фактора (цикла)
      aStart:= Start;      // стартовая вершина фактора (любая из)
    end;
  end; // while NodesCnt >0
end;

// Проверка возможности пропуска через двудольный граф
// насыщенного (максимального) потока
// Возвращает:
// FALSE -- если хотя бы одна из вершин не пропускает ни одной единицы потока
// TRUE  -- если через все вершины проходит хоть одна единица потока
// (Link.mHigh <> 0) -- признак закрытой дуги (заблокированной)

function THamGraph.Check: boolean;
var Node: THamNode;
    Link: TLink;
begin
  Result:= false;
  // Перебор вершин
  Node:= NodeFirst as THamNode;
  while Assigned(Node) do begin
    if Node.mLeft then begin
      // В левой доле исследуем исходящие линки
      Link:= Node.OutLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.OutLinkNext;
      if not Assigned(Link) then Exit;
    end else begin
      // В правой доле исследуем входящие линки
      Link:= Node.InLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.InLinkNext;
      if not Assigned(Link) then Exit;
    end;
    Node:= NodeNext as THamNode;
  end;
  Result:= true;  // насыщенный поток возможен
end;

// Сохранение текущего потока в качестве наилучшего
// Вызывается при обнаружении очередного лучшего фактора (цикла)

procedure THamGraph.SaveBest(aBuf: TBuffer);
var Start, Node : THamNode;
begin
  aBuf.Clear; // очистка буфера результата
  // Начинаем с первой вершины в левой доле
  Start:= NodeFirst as THamNode;
  while not Start.mLeft do Start:= NodeNext as THamNode;
  // Обходим цикл от Start до Start
```

```
  Node:= Start;
  repeat
    aBuf.Put(Node.mNode);
    Node:= Node.NextCircNode;
  until Node = Start;
end;

// Стягивание факторов в новый граф для вычисления прогноза стоимости
// aFact - количество факторов
// Вызывается после построения потока и факторов в последовательности:
// MarkMinPairsDicoty -> CalcFactors -> Shrink

function THamGraph.Shrink(aFact: integer): THugeGraph;
var i: integer;
    Link: TLink;
    N1, N2: TNodeInt;   // вершины стянутого графа помечены числами
    NH1, NH2: THamNode; // вершины данного двудольного графа
    Cost: integer;      // остаточная стоимость дуги
begin
  // Создаём пустой граф:
  Result:= THugeGraph.Create('',  // имя
                         true,    // граф направленный
                         false,   // вершины не нагружены
                         true     // дуги нагружены
                        );
  // Создаём и вставляем вершины, помеченные числами от 1 до aFact
  for i:= 1 to aFact do Result.InsertNode(TNodeInt.Create(i, 0, Result));

  // Соединяем дугами бесконечной длины все вершины созданного графа
  with Result do begin
    N1:= NodeFirst as TNodeInt;
    while Assigned(N1) do begin
      PosPush;
      N2:= NodeFirst as TNodeInt;
      while Assigned(N2) do begin
        if N1 <> N2 then SetLink(N1, N2, MaxInt);
        N2:= NodeNext as TNodeInt;
      end;
      PosPop;
      N1:= NodeNext as TNodeInt;
    end;
  end;

  // Находим минимальные относительные стоимости дуг нового графа
  // Остаточные стоимости дуг в полях TLink.mLow
  // определены при поиске минимального паросочетания (потока)
  // процедурой TGraph.MarkMinPairsDicoty
  NH1:= NodeFirst as THamNode;
  while Assigned(NH1) do begin
    // Обработка вершин левой доли:
    if NH1.mLeft then begin
      // Перебираем дуги вершины:
      Link:= NH1.OutLinkFirst;
      while Assigned(Link) do begin
        NH2:= (Link.mDest as THamNode).mBack;
        // Если линк ведёт в другой фактор (цикл), то обрабатываем
        if NH1.mValue <> NH2.mValue then begin
          // Остаточная стоимость дуги после редуцирования
          // (определена процедурой TGraph.MarkMinPairsDicoty)
          Cost:= Link.mLow;
          // Находим линк между вершинами (факторами) в стянутом графе
          // Здесь поля NH1.mValue, NH2.mValue соответствуют факторам
```

```
          // и вершинам стянутого графа
          Link:= Result.GetLinkByNumber(NH1.mValue, NH2.mValue);
          // Назначаем этому линку пониженую (редуцированную) цену
          if Link.mValue > Cost then Link.mValue:= Cost;
        end;
        Link:= NH1.OutLinkNext;
      end;
    end;
    NH1:= NodeNext as THamNode;
  end;
  // Выполняем компрессию дуг, заменяя их длину кратчайшими расстояниями
  Result.Compress;
end;

// Оценка прогнозируемой стоимости и подсчёт количества факторов
// при текущем состоянии клапанов (открытых и закрытых дуг)
// Возвращает:
// Result   -- нижний предел стоимости для данного состояния клапанов mHigh
// aFactors -- количество факторов
// aStart   -- стартовую вершину кратчайшего фактора

function THamGraph.CalcCostAndFactors(
                           var aFactors: integer; //кол-во факторов
                           var aStart: THamNode   // стартовая вершина
                           ): integer;            // стоимость потока

var Level: integer; // уровень рекурсии

  // Рекурсивная процедура стягивания факторов

  procedure Local(aGraph: THamGraph);
  var Shrinked: TGraph; // очередной граф, образованный стягиванием факторов
      HamGr: THamGraph; // и соответствующий ему двудольный граф
      Cost: integer;    // стоимость потока
      Factors: integer; // количество факторов
      Dummy: integer;   // количество обнаруженных пар (не используем)
      Node: THamNode;   // любая вершина кратчайшего фактора
  begin
    Inc(Level); // уровень рекурсии +1

    if Level=1 then begin
      // На первом уровне проверим возможность достижения насыщенного потока
      if not aGraph.Check then begin
        // Насыщенный поток невозможен
        Result:= -1;   Exit;
      end;
    end;

    // Вычисляем стоимость полного потока на двудольном графе
    Cost:= aGraph.MarkMinPairsDicoty(Dummy);

    // Накапливаем результат (прогнозируемую стоимость)
    Inc(Result, Cost);

    // Находим факторы (циклы)
    Factors:= aGraph.CalcFactors(Node);

    // На первом уровне рекурсии запоминаем количество факторов
    // и любую вершину кратчайшего фактора

    if Level=1 then begin aFactors:= Factors; aStart:= Node; end;
```

```
    // Если фактор единственный, то выход с возвращением цены
    if Factors = 1 then Exit;

    // При наличии нескольких факторов (не связанных циклов)
    // стягиваем факторы в вершины, создаём новый полный граф
    // и компрессируем его дуги
    Shrinked:= aGraph.Shrink(Factors);

    // Создаём двудольный граф типа THamGraph
    // и рекурсивно повторяем предыдущие шаги

    HamGr:= THamGraph.Create(Shrinked);
    Local(HamGr);

    // Удаляем вспомогательные графы
    HamGr.Free;
    Shrinked.Free;
    Dec(Level); // уровень рекурсии -1
  end;

begin   { THamGraph.CalcCostAndFactors }
  Result:=0;   aFactors:= 0; aStart:= nil;  Level:= 0;
  Local(Self);
end;
```

Метод компрессии дуг реализован в базовом классе **TGraph** и представлен ниже.

Листинг 33-4 — Метод компрессии дуг

```
// Компрессия дуг: замена длин дуг кратчайшими расстояниями
// Используется для поиска гамильтонова цикла

procedure TGraph.Compress;
var  N: TNode;
     L: TLink;
    FL: TFarLink;
begin
  // Строим карту с дальними указателями:
  InitMap_Floyd;
  // Заменяем длины дуг кратчайшими путями:
  N:= NodeFirst;
  while Assigned(N) do begin
    // Перебор дальних указателей:
    FL:= N.mFarLinks.GetFirst as TFarLink;
    while Assigned(FL) do begin
      if (FL.mNodeFar <> N) then begin
        // Ищем линк из данной вершины N --> mNodeFar
        L:= N.GetLink(FL.mNodeFar);
        // Ели он существует, меняем длину на кратчайшее расстояние
        if Assigned(L) then L.mValue:= FL.mDist;
      end;
      FL:= N.mFarLinks.GetNext as TFarLink;
    end;
    N:= NodeNext;
  end;
  // Удаляем карту:
  DoneMap;
end;
```

Ниже представлен текст основного метода поиска гамильтонова цикла, который использует описанные выше объекты. Метод возвращает стоимость цикла и соответствующую последовательность вершин, начиная с первой.

Листинг 33-5 — Основной метод поиска гамильтонова цикла

```
function TGraph.GenHamilton(var aCost: integer): TBuffer;

var   HamGraph: THamGraph; // вспомогательный двудольный граф
      Que : TSet;          // Приоритетная очередь клапанов (по неубыванию)
      BestCost: integer;   // лучшая цена на текущий момент
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Отключение дуг, переданных в множестве aLinks

  procedure LinksOff(aLinks: TSet);
  var Link: TLink;
  begin
    Link:= aLinks.GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // отключить дугу
      Link:= aLinks.GetNext as TLink;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Включение дуг, переданных в множестве aLinks

  procedure LinksOn(aLinks: TSet);
  var Link: TLink;
  begin
    Link:= aLinks.GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 0;  // включить дугу
      Link:= aLinks.GetNext as TLink;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка объекта с клапанами aClosed
  // (множеством отключенных и множествами отключаемых дуг)

  procedure Handle(aClosed: TClosed);
  var Cost: integer;     // прогноз или окончательная стоимость решения
      Factors: integer; // количество факторов в очередном решении
      Start: THamNode;  // стартовая вершина кратчайшего фактора
      Closed: TClosed;  // новый объект с клапанами
      Links: TSet;      // множество отключенных дуг в текущий момент
      Node: THamNode;
  begin
    // Отключить блокированные дуги объекта
    aClosed.GatesOff;

    // Цикл: поочерёдно отключаем дуги, ведущие внутрь фактора,
    // оцениваем стоимость соответствующих потоков,
    // и ставим в приоритетную очередь элементы TClosed
    // для частичных решений

    // Перебор множеств отключаемых дуг:
    Links:= aClosed.mFactors.GetFirst as TSet;
    while Assigned(Links) do begin
      // Отключаем дуги одной из вершин фактора, ведущие внутрь этого фактора
      LinksOff(Links);
      // При текущем состоянии клапанов определить стоимость потока
```

```
      Cost:= HamGraph.CalcCostAndFactors(Factors, Start);
      // Вновь подключаем дуги вершины, ведущие внутрь фактора
      LinksOn(Links);
      // Если поток существует, и он лучше текущего...
      if (Cost>=0) and (Cost< BestCost) then begin
        // Здесь поток существует и его стоимость ниже текущего минимума.
        // Если количество факторов=1, то запомнить лучший результат.
        if Factors=1 then begin
          BestCost:= Cost;            // лучшая цена
          aCost:= Cost;               // она же как результат
          // сохраняем последовательность вершин цикла в буфере результата
          HamGraph.SaveBest(Result);
        end else begin
          // Здесь количество факторов Factors > 1 (частичное решение),
          // то создаём и ставим в очередь множество закрытых клапанов
          Links.Add(aClosed.mClosed); // объединяем Links и mClosed
          Closed:= TClosed.Create(Cost, Links);
          // Обходим кратчайший фактор
          // и сохраняем в объекте Closed отключаемые дуги
          Node:= Start; // Начальная (любая) вершина фактора
          repeat
            // Функция THamNode.GenFactorsLinks создаёт множество дуг
            // ведущих внутрь текущего фактора
            Closed.InsertLinksSet(Node.GenFactorsLinks);
            Node:= Node.NextCircNode; // следующая вершина фактора
          until Node = Start;
          Que.Insert(Closed); // Вставляем объект в приоритетную очередь
        end;
      end;
      // Следующее множество отключаемых дуг:
      Links:= aClosed.mFactors.GetNext as TSet
    end;
    // Вернуть исходное состояние клапанов
    aClosed.GatesOn;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -

var  Closed: TClosed; // очередное множество закрытых клапанов

begin  { TGraph.GenHamilton }

  Result:= TBuffer.Create;  // создаём пустой буфер для хранения решения
  aCost:= -1;               // стоимость на случай отсутствия решения

  // Создаём вспомогательный двудольный граф HamGraph (Dicotyledonous)
  // с целью определения минимальной стоимости полного потока,
  // а также очередь клапанов Que (элементов типа TGate)

  HamGraph:= THamGraph.Create(Self);
  Que:= CreateSet;  // Приоритетная очередь клапанов по неубыванию стоимости
  BestCost:= MaxInt;// Минимальная стоимость на текущий момент

  // Создаём и вставляем в приоритетную очередь фиктивный элемент,
  // в котором множества закрытых и закрываемых линков пусты
  Closed:= TClosed.Create(0, nil);
  Closed.InsertLinksSet(CreateSet);
  Que.Insert(Closed);

  // Цикл обработки приоритетной очереди:
  while Que.GetCount > 0 do begin
    // Выбрать из очереди объект-множество клапанов
    Closed:= Que.GetFirst as TClosed;
```

```
    // Если его прогнозируемая цена меньше лучшей на данный момент,
    // то обработать это состояние клапанов
    if Closed.mCost < BestCost then Handle(Closed);
    // Удалить объект-множество клапанов
    Que.Delete(Closed);
    Closed.Free;
  end;
  // Очистка памяти:
  Que.Free;        // приоритетная очередь клапанов по неубыванию стоимости
  HamGraph.Free;   // вспомогательный двудольный граф (Dicotyledonous)
end;
```

33.8. Испытания

В ходе испытаний основного метода оценим его точность (в сравнении с полным перебором), а также время работы в зависимости от размера графа. Объектом расчётов будут случайные графы с дугами, длины которых распределены равномерно в интервале от 1 до 99. Ниже дана программа для сравнения четырёх методов поиска на отдельных случайно генерируемых графах.

Листинг 33-6 — Программа для сравнения методов поиска циклов Гамильтона

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

// Преобразование буфера в строку

function HamToStr(aBuf: TBuffer): string;
var N: TNode;
    i: integer;
begin
  Result:='';
  for i:= 1 to aBuf.GetCount do begin
    N:= aBuf.Get as TNode;
    aBuf.Put(N);
    Result:= Result+ N.GetName;
    if i<aBuf.GetCount then Result:=Result +'-';
  end;
end;

var Gr : TGraph;
    Res : TBuffer;
    Cost: integer;
    Nodes: integer;
    Start: TDateTime;
    Time: integer;
    N : integer;

begin
    repeat
```

```
      Write('Nodes= '); Readln(Nodes);
      if Abs(Nodes)<3 then Break;
      {
        Nodes > 0 -- орграф,  Nodes < 0 -- граф
      }
      Gr:= TGraphChars.GenFull((Nodes>0), 0, 99, Abs(Nodes));
      if Gr.Nodes < 18 then begin
        Writeln('-------------- Full -------------------');
        Start:= Now;
        Res:= Gr.GenHamilton_Full(Cost);
        Time:= MilliSecondsBetween(Start, Now);
        Writeln(HamToStr(Res));
        Res.Free;
        Write('Cost    = ', Cost);
        Writeln('  Time (ms) =', Time:5, #10);
        N:= Cost; // Стоимость образца
      end;
      Writeln('-------------- Main -------------------');
      Start:= Now;
      Res:= Gr.GenHamilton(Cost);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Res.Free;
      Write('Cost(T)= ', Cost);
      Writeln('  Time (ms) =', Time:5, #10);
      if Gr.Nodes >= 18 then N:= Cost; // Стоимость образца

      if Cost>0 then begin
      // Жадный и случайный выбор
      Writeln('-------------- Greed true -------------------');
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, true);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Res.Free;
      Write('Cost= ', Cost);
      Writeln('  Time (ms) =', Time:5, #10);
      Writeln('R2= ', 100*Cost/N:6:1);
      // Случайный выбор
      Writeln('-------------- Greed false -------------------');
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, false);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Res.Free;
      Write('Cost= ', Cost);
      Writeln('  Time (ms) =', Time:5, #10);
      Writeln('R3= ', 100*Cost/N:6:1);
      Writeln('-----------------------------------------------'#10);
      end;
      Gr.Free;
    until false;
end.
```

Более достоверные результаты даст обработка большого количества графов с накоплением статистики. Следующая программа генерирует и обрабатывает заданное пользователем количество полных графов и вычисляет среднее время их обработки, процент точных результатов, а также среднее и максимальное отклонения стоимости от идеала. В качестве идеала берётся стоимость, полученная либо полным перебором (для небольших графов), либо основным методом. При

указании положительного количества вершин генерируются ориентированные несимметричные графы, при указании отрицательного — неориентированные графы (симметричные орграфы).

Примечание: вызываемый в данной программе метод **GenHamilton_Quick** будет рассмотрен в следующей главе.

Листинг 33-7 — Накопление статистики
по пяти методам поиска циклов Гамильтона

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
{-------------------------------------------------------}

procedure Test(aRetry, aNodes: integer);
var Gr : TGraph;              // исследуемый граф
    Res : TBuffer;            // результат - гамильтонов цикл
    Cost, CostFull: integer;           // стоимости
    Sum2, Sum3, Sum4, Sum5: Extended; // суммы относительной стоимости в %
    Max2, Max3, Max4, Max5: Extended; // максимальные отклонения в процентах
    OK2, OK3, OK4, OK5: integer;      // количество совпадений с идеалом

    T1, T2, T3, T4, T5 : integer; // время накопленное
    Start : TDateTime;            // для засечки времени

    Cnt : integer;      // счётчик решений
    Delta : integer;    // абсолютное отклонение
    pc :  extended;     // отношение в %

   // Вывод статистики
  procedure Expo(const aFile: String);
  const CLine='-------------------------------------------------------';
  var  x1, x2, x3, x4, x5 : Extended;
  begin
    if aFile <> '' then begin
      Assign(Output, aFile);
      if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
    end;
    Writeln(CLine);
    Writeln('Retry=', aRetry:4,  '  Nodes=', aNodes:4);
    Writeln(CLine);
    Writeln('                  Full      Ham      Greed    Random     Quick');
    // Относительная стоимость, %:
    x2:= Sum2/aRetry;
    x3:= Sum3/aRetry;
    x4:= Sum4/aRetry;
    x5:= Sum5/aRetry;
    Write('Ratio(%) = ', '':10, x2:10:1, x3:10:1, x4:10:1);
    if aNodes > 0 then Writeln else Writeln(x5:10:1);

    // максимальные отклонения в процентах
```

```
    Write('MaxR (%) = ', '':10, Max2:10:1, Max3:10:1, Max4:10:1);
    if aNodes > 0 then Writeln else Writeln(Max5:10:1);
    // количество совпадений с идеалом в процентах
    Write('OK   (%) = ', '':10, 100*OK2/aRetry:10:1,
                 100*OK3/aRetry:10:1, 100*OK4/aRetry:10:1);
    if aNodes > 0 then Writeln else Writeln(100*OK5/aRetry:10:1);
    // Среднее время, мс
    x1:= T1/aRetry; x2:= T2/aRetry; x3:= T3/aRetry; x4:= T4/aRetry;
    x5:= T5/aRetry;
    Write('Time (ms)= ', x1:10:1, x2:10:1, x3:10:1, x4:10:1);
    if aNodes > 0 then Writeln else Writeln(x5:10:1);

    Writeln(CLine);
    if aFile <> '' then begin
      Close(Output);  Assign(Output, '');    Rewrite(Output);
    end;
  end;

begin  { Test }

    // суммы относительной стоимости в %
    Sum2:= 0; Sum3:= 0; Sum4:= 0; Sum5:=0;
    // максимальные отношение в процентах
    Max2:= 0; Max3:=0; Max4:=0; Max5:=0;
    // количество совпадений с идеалом
    OK2:=0; OK3:=0; OK4:=0; OK5:=0;
    // время накопленное
    T1:=0; T2:=0; T3:=0; T4:=0; T5:=0;

    for Cnt:= 1 to aRetry do begin
      // aNodes>0 - орграф,  aNodes<0 - симметричный граф
      Gr:= THugeGraph.GenFull((aNodes>0), 0, 99, Abs(aNodes));

      if Abs(aNodes) > 15 then CostFull:=-1
      else begin
        {-------- Full -------}
        Start:= Now;
        Res:= Gr.GenHamilton_Full(CostFull);
        Inc(T1, MilliSecondsBetween(Start, Now));
        Res.Free;
      end;

      {-------- Main -------}
      Start:= Now;
      Res:= Gr.GenHamilton(Cost);
      Inc(T2, MilliSecondsBetween(Start, Now));
      Res.Free;
      if CostFull < 0 then CostFull:= Cost;
      Delta:= Cost - CostFull;
      if Delta=0 then Write('.') else Write('*');
      if Delta = 0 then Inc(OK2);
      pc:= 100 * Cost/CostFull;
      Sum2:= Sum2 + pc;
      if Max2 < pc then Max2:= pc;

      {-------- Greed -------}
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, true);
      Inc(T3, MilliSecondsBetween(Start, Now));
      Res.Free;
      Delta:= Cost - CostFull;
      if Delta = 0 then Inc(OK3);
```

```
      pc:= 100 * Cost/CostFull;
      Sum3:= Sum3 + pc;
      if Max3 < pc then Max3:= pc;

      {-------- Random -------}
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, false);
      Inc(T4, MilliSecondsBetween(Start, Now));
      Res.Free;
      Delta:= Cost - CostFull;
      if Delta = 0 then Inc(OK4);
      pc:= 100 * Cost/CostFull;
      Sum4:= Sum4 + pc;
      if Max4 < pc then Max4:= pc;

      {-------- Quick -------}
      Start:= Now;
      Res:= Gr.GenHamilton_Quick(Cost);
      Inc(T5, MilliSecondsBetween(Start, Now));
      Res.Free;
      Delta:= Cost - CostFull;
      if Delta = 0 then Inc(OK5);
      pc:= 100 * Cost/CostFull;
      Sum5:= Sum5 + pc;
      if Max5 < pc then Max5:= pc;

      Gr.Free;
    end; // for
    Writeln(#7);
    // Вывод статистики
    Expo('');
    Expo('Out.txt');
end;
{-------------------------------------------------------}
var Nodes: integer;
    Retry : integer;

begin
    Write('Retry= '); Readln(Retry);
    if Retry <= 0 then Exit;
    repeat
      Write('Nodes= '); Readln(Nodes);
      if Abs(Nodes) < 4 then Break;
      Test(Retry, Nodes);
    until false;
end.
```

Ниже даны результаты обработки для 200 *графов* с количеством вершин от 11 до 14, и 500 *орграфов* с тем же количеством вершин. Отклонение от идеала (погрешность) в сравнении с полным перебором дана в табл. 33-1.

Табл. 33-1 — Погрешность вычисления стоимости цикла в сравнении с полным перебором, %

Алгоритм	Средняя погрешность,%		Максимальная погрешность,%	
	Орграфы	Графы	Орграфы	Графы
Основной алгоритм	менее 0,1	0,1	5,6	9,2
Жадный алгоритм	64,6	36,2	233,0	132,0
Случайный выбор	294,6	224,2	913,0	517,0

Отметим высокую точность основного алгоритма, а также существенную разницу результатов для орграфов и ненаправленных графов (симметричных орграфов).

Среднее время обработки одного графа дано в табл. 33-2. Здесь для жадного и случайного алгоритмов времена не показаны ввиду их ничтожности в сравнении с другими алгоритмами.

Табл. 33-2 — Среднее время обработки одного графа, ms

Количество вершин	Полный перебор		Основной алгоритм	
	Орграфы	Графы	Орграфы	Графы
11	9,7	21,0	1,3	3,7
12	31,8	74,0	1,6	6,1
13	95,2	265,0	2,2	8,9
14	297,0	920,0	3,1	12,9

Зависимость среднего времени вычислений от количества вершин показана на следующих рисунках.

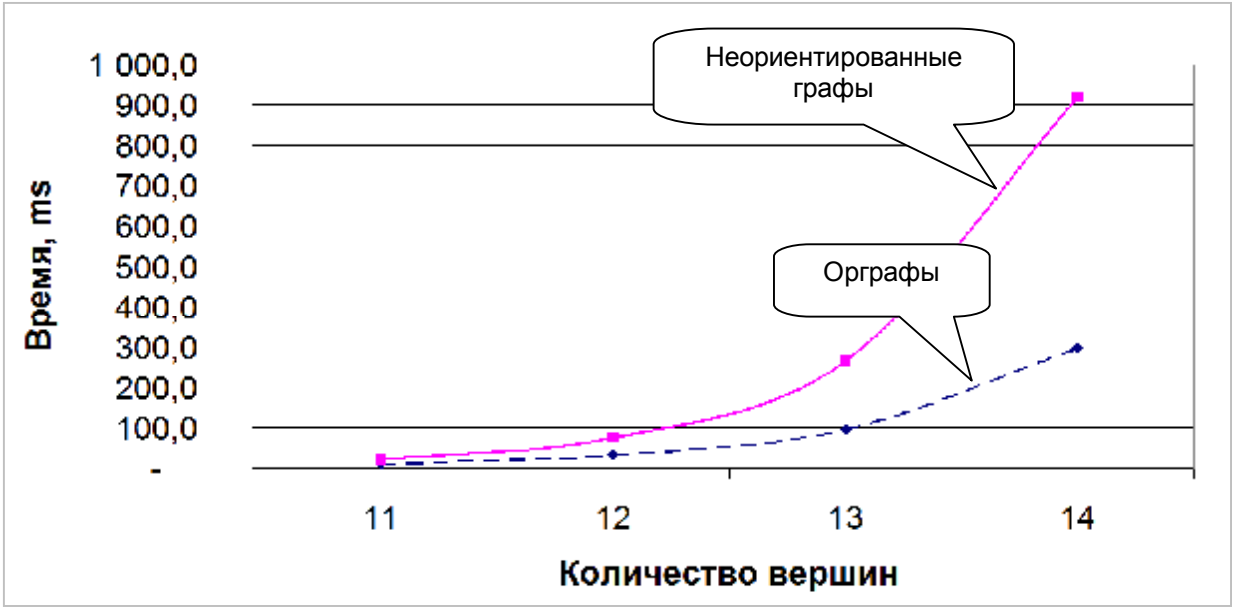


Рис. 33-5 — Зависимость среднего времени *полного перебора* от количества вершин

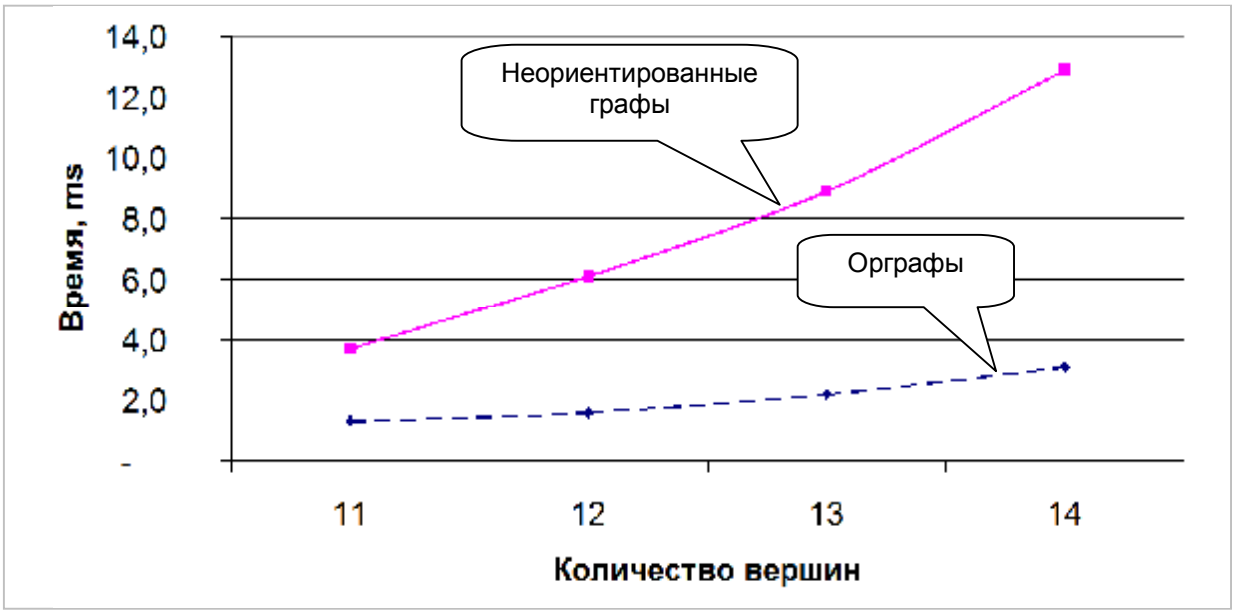


Рис. 33-6 — Зависимость среднего времени работы *основного алгоритма* от количества вершин

Из приведенных выше результатов следует, что:

- Основной алгоритм обладает в целом и высокой точностью, и высокой скоростью работы.
- Точность и скорость основного алгоритма заметно ниже при обработке неориентированных графов (симметричных орграфов).
- Жадный алгоритм сильно уступает в точности основному, но ввиду непревзойдённой скорости его применение порою оправдано.

33.9. Разбор уточнённого прогноза

Напоследок рассмотрим числовой пример, подтверждающий полезность уточнённого прогноза. Воспользуемся для этого симметричным графом, представленным в табл. 33-3.

Табл. 33-3 — Исходный симметричный граф

Источники дуг	Приёмники дуг									
	A	B	C	D	E	F	G	H	I	J
A	*	32	41	22	20	57	54	32	22	45
B	32	*	22	50	42	51	61	20	54	51
C	41	22	*	63	41	30	45	10	60	36
D	22	50	63	*	36	78	72	54	20	64
E	20	42	41	36	*	45	36	32	22	28
F	57	51	30	78	45	*	22	32	67	20
G	54	61	45	72	36	22	*	41	57	10
H	32	20	10	54	32	32	41	*	50	32
I	22	54	60	20	22	67	57	50	*	50
J	45	51	36	64	28	20	10	32	50	*

Описанные далее действия относятся главным образом к методу **`THamGraph.CalcCostAndFactors`**.

В исходном состоянии все дуги графа открыты. Венгерский алгоритм даёт здесь полный замкнутый поток минимальной стоимости, представленный в табл. 33-4 и на рис. 33-7. Этот поток образован четырьмя факторами, его стоимость составляет 184 единицы.

Табл. 33-4 — Факторы (4), образующие
полный поток минимальной стоимости в исходном графе

№	Фактор (цикл)	Обозначение стянутой вершины
1	A → E → A	a1
2	B → H → C → B	b1
3	D → I → D	c1
4	F → J → G → F	d1

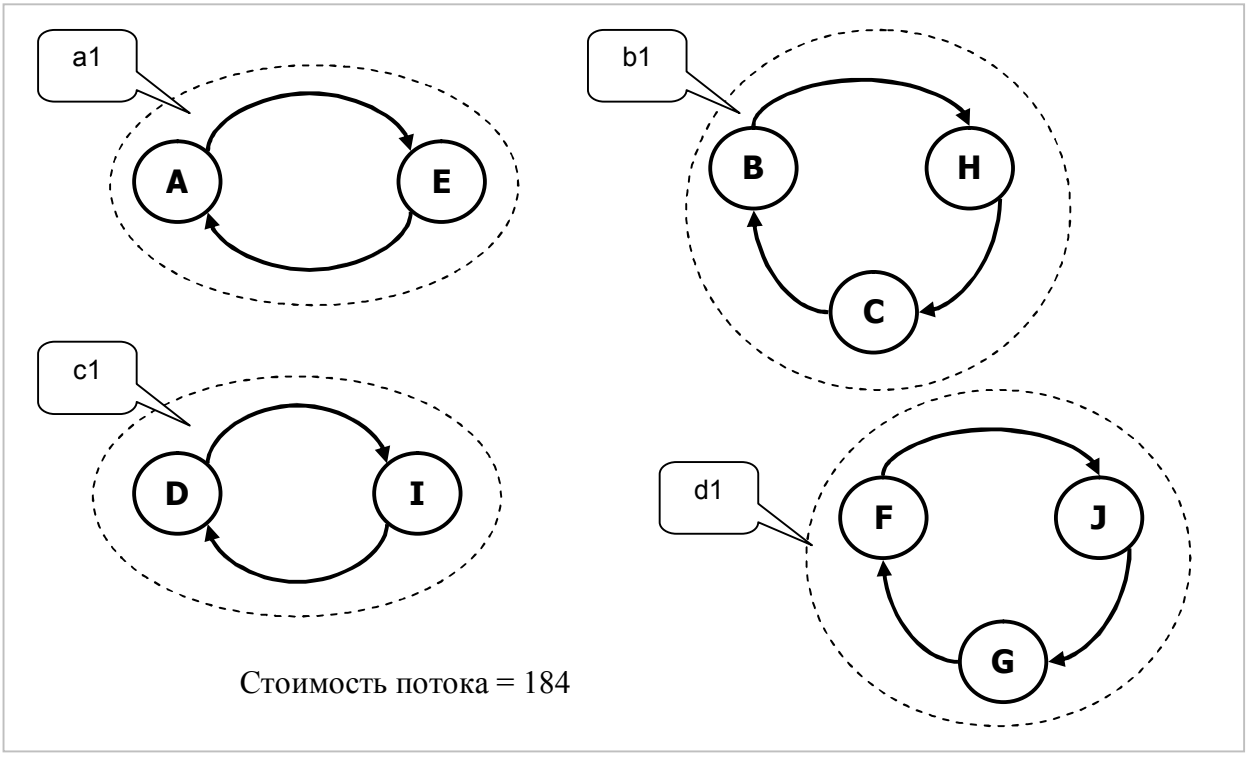


Рис. 33-7 — Факторы (4), образующие полный поток минимальной стоимости

Остаточная (редуцированная) стоимость дуг исходного графа сохранена в полях **`TLink.mLow`** и дана в табл. 1-5. Из этих данных строим полный стянутый граф, где четырём факторам будут сопоставлены псевдо-вершины **`a1`**, **`b1`**, **`c1`**, **`d1`** (см. рис. 33-8).

Табл. 33-5 — Остаточная (редуцированная стоимость дуг исходного графа

Источники дуг	Приёмники дуг									
	A	B	C	D	E	F	G	H	I	J
A	*	12	31	2	0	37	44	24	2	37
B	0	*	0	18	10	19	39	0	22	31
C	19	0	*	41	19	8	33	0	38	26
D	2	30	53	*	16	58	62	46	0	56
E	0	22	31	16	*	25	26	24	2	20
F	25	19	8	46	13	*	0	12	35	0
G	32	39	33	50	14	0	*	31	35	0
H	12	0	0	34	12	12	31	*	30	24
I	2	34	50	0	2	47	47	42	*	42
J	25	31	26	44	8	0	0	24	30	*

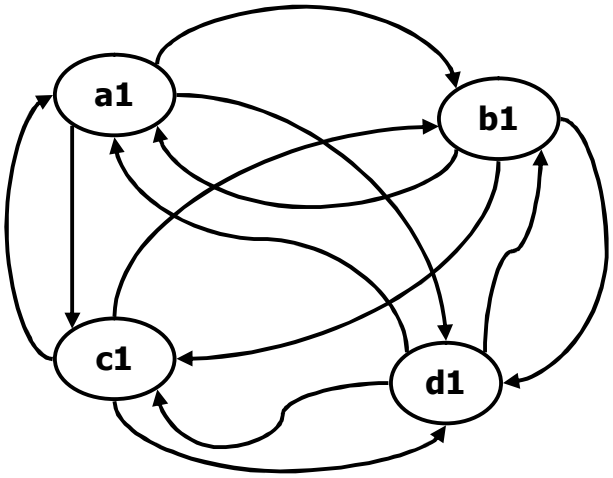


Рис. 33-8 — Полный стянутый граф

Для назначения стоимости дугам стянутого графа (в полях **TLink.mValue**), надо рассмотреть все дуги, ведущие из одного подмножества вершин в другое. Так, например, для дуги **a1**→ **b1** необходимо перебрать дуги, показанные в табл. 33-6, и выбрать среди них кратчайшую.

Табл. 33-6 — Перебор дуг, направленных из фактора a1 в фактор b1
(из подмножества A-E в подмножество B-C-H)

Дуга	Остаточная стоимость	Минимум
A → B	12	12
A → C	31	
A → H	24	
E → B	22	
E → C	31	
E → H	24	

Перебрав, таким образом, все сочетания стянутых вершин, получим полный стянутый граф, показанный в табл. 33-7.

Табл. 33-7 — Стянутый граф 1-го уровня до компрессии

Источники дуг	Приёмники дуг			
	a1	b1	c1	d1
a1	*	12	2	20
b1	0	*	18	8
c1	2	30	*	42
d1	8	8	30	*

Далее обеспечим правило треугольника (компрессию), то есть, проследим за тем, чтобы длины всех дуг в графе оказались не длиннее «окольных путей» между двумя вершинами. Здесь правило нарушено для четырёх дуг (они выделены серым). Например, длина прямой дуги **b1** → **c1** составляет 18 единиц, а длина обходного пути **b1** → **a1** → **c1** равна всего 2 единицам, поэтому меняем здесь 18 на 2. Так же поступаем с остальными дугами (табл. 33-8), после чего получаем сжатый стянутый граф (табл. 33-9).

Табл. 33-8 — Компрессия дуг
(замена длины дуги длинной кратчайшего пути между вершинами)

Дуга	Длина дуги	Кратчайший путь	Длина пути
b1 → c1	18	b1 → a1 → c1	2
c1 → b1	30	c1 → a1 → b1	14
c1 → d1	42	c1 → a1 → d1	22
d1 → c1	30	d1 → a1 → c1	10

Табл. 33-9 — Стянутый граф 1-го уровня после компрессии

Источники дуг	Приёмники дуг			
	a1	b1	c1	d1
a1	*	12	2	20
b1	0	*	2	8
c1	2	14	*	22
d1	8	8	10	*

Венгерский алгоритм найдёт в этом графе поток стоимостью 20 единиц, — эта стоимость добавится к прогнозу (184+20). Поток представится двумя факторами, показанными в табл. 33-10 и на рис. 33-9.

Табл. 33-10 — Факторы (2), образующие полный поток
минимальной стоимости в стянутом графе 1-го уровня

№	Фактор (цикл)	Обозначение стянутой вершины
1	a1 → c1 → a1	a2
2	b1 → d1 → b1	b2

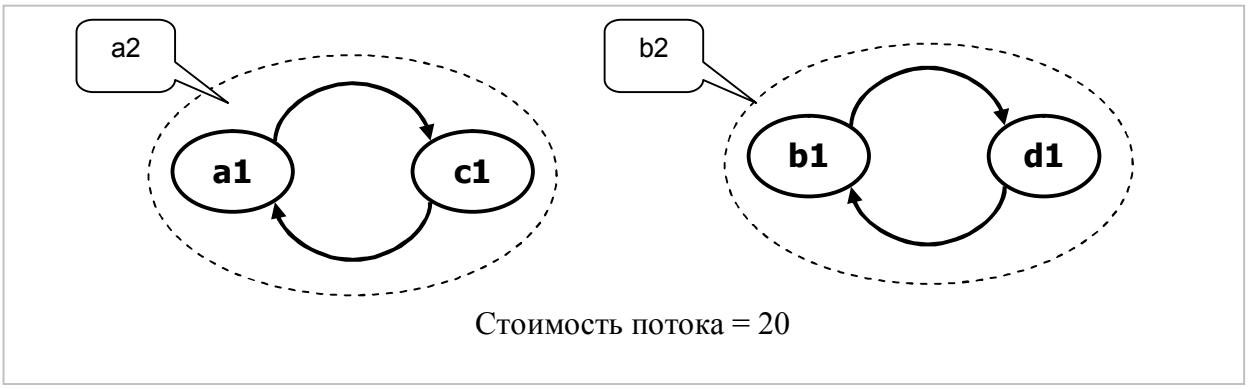


Рис. 33-9 — Факторы (2), полученные в результате поиска полного потока минимальной стоимости в стянутом графе 1-го уровня

По завершении венгерского алгоритма остаточная стоимость дуг представится матрицей в табл. 33-11.

Табл. 33-11 — Остаточная (редуцированная) стоимость дуг в стянутом графе 1-го уровня

Источники дуг	Приёмники дуг			
	a1	b1	c1	d1
a1	*	4	0	12
b1	0	*	0	0
c1	0	4	*	12
d1	8	0	8	*

Поскольку найденный поток вновь образован не одним фактором, продолжим уточнение прогноза, повторив (рекурсивно) выше описанные шаги: стянем факторы в вершины, найдём кратчайшие длины дуг и выполним их компрессию. Результат показан в табл. 33-12, здесь компрессия дуг ничего не изменила, поэтому показана лишь одна таблица.

Табл. 33-12 — Стянутый граф второго уровня (до и после компрессии)

Источники дуг	Приёмники дуг	
	a2	b2
a2	*	4
b2	0	*

Венгерский метод найдёт здесь единственный фактор стоимостью 4 единицы, в итоге прогноз стоимости составит **184+20+4=208** единиц. Поскольку конечный прогноз получен, далее рекурсивно выходим из процедуры с уничтожением вспомогательных объектов.

Табл. 33-13 — Фактор, полученный в результате поиска полного потока минимальной стоимости в стянутом графе 2-го уровня

№	Фактор (цикл)	Обозначение стянутой вершины
1	a2 → b2 → a2	a3

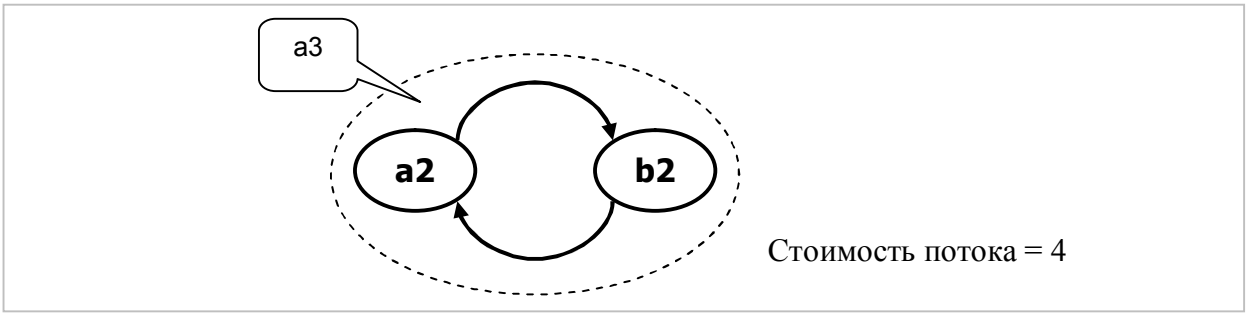


Рис. 33-10 — Единственный фактор, полученный в результате поиска полного потока минимальной стоимости в стянутом графе 2-го уровня

Итак, найден уточнённый прогноз стоимости промежуточного решения в ситуации, когда все вершины графа подключены. Затем это делается при каждом последующем отключении некоторых дуг. Проследим за процессом, воспользовавшись табл. 33-14. Здесь в первой колонке (шаг) указан номер промежуточного решения, то есть, идентификатор объекта **TClosed**. Начинаем с фиктивного объекта, в котором нет отключенных дуг.

Табл. 33-14 — Последовательность решений с уточнённым прогнозом

Шаг	Исх. прогноз	Отключенные дуги	Отключа-емые дуги	Состав прогноза	Конечный прогноз	Текущее решение
1	0	нет	нет	184+20+4	208	нет
2	208	нет	A-E	188+24	212	нет
		нет	E-A	188+24	212	нет
3	212	A-E	B-C, B-H	206+7	213	нет
		A-E	C-B, C-H	204+8	212	нет
		A-E	H-B, H-C	208+32	240	нет
4	212	E-A	B-C, B-H	206+12+0	218	нет
		E-A	C-B, C-H	204+12	216	нет
		E-A	H-B, H-C	208+12	220	нет
5	212	A-E, C-B, C-H	B-H	208+16	224	нет
	212	A-E, C-B, C-H	H-B	212+0	212	нет
6	212	A-E, C-B, C-H, H-B	D-I	216	216	216
		A-E, C-B, C-H, H-B	I-D	215+10	225	216
7	213	A-E, B-C, B-H	C-H	212+7	219	216
		A-E, B-C, B-H	H-C	215+1	216	216
8	216	E-A, C-B, C-H	*	*	*	216
9	218	E-A, B-C, B-H	*	*	*	216
10	220	E-A, H-B, H-C	*	*	*	216
11	224	A-E, C-B, C-H, B-H	*	*	*	216
12	240	A-E, H-B, H-C	*	*	*	216

При вычислении прогноза для стартового объекта **TClosed** обнаружены (см. выше) четыре фактора, и кратчайший из них: **A** → **E** → **A**. Поэтому при обработке следующего промежуточного решения №2 с исходным прогнозом **208** поочерёдно отключаем дуги **A** → **E** и **E** → **A**, получая при этом ещё два промежуточных решения с прогнозами, равными **212** единиц. Оба они не дают конечного решения, поскольку содержат несколько факторов, и потому ставятся в очередь. Далее (№3 и №4) они выбираются из очереди, и к уже отключенным дугам поочерёдно добавляются дуги кратчайшего фактора **B** → **C** → **H** → **B**. Эти решения вновь не дают единственного фактора, и потому ставятся в приоритетную очередь. Наконец, при обработке объекта №6 и отключении соответствующих дуг, получаем единственный фактор стоимостью **216**, — эту конфигурацию потока и его цену запоминаем в качестве текущего решения. При обработке объектов №6 и №7 получаются ещё три промежуточных решения с прогнозами **225**, **219** и **216** единиц, но они не ставятся в очередь, поскольку эти прогнозы не меньше цены текущего решения (**216**). Начиная с объекта №8, уже не обрабатываем оставшиеся в очереди промежуточные решения, поскольку их прогнозы не меньше текущего решения, — эти объекты очереди просто уничтожаются (отмечены звёздочками).

Итак, решение найдено при обработке 6-го объекта, а всего в очередь попало **12** объектов. Теперь сравним этот результат с упрощённым вариантом прогноза, когда он ограничивается лишь первым уровнем рекурсии, — для этого в процедуре **THamGraph.CalcCostAndFactors** организуем досрочный выход, отключив комментарием проверку условия:

```
    // Если фактор единственный, то выход с возвращением цены
    { if Factors = 1 then } Exit;
```

Результат дан в табл. 33-15, где показаны прогноз и конечное решение. Теперь оно получено только на **15**-м шаге, а количество объектов в очереди достигло **33**. Отметим, также, что и после обнаружения решения на шагах с **16** по **26** обрабатываются ещё **11** объектов из очереди, поскольку их грубые прогнозы оказались ниже цены текущего решения. И только, начиная с **27**-го, все последующие объекты игнорируются (отмечены звёздочками).

Табл. 33-15— Последовательность обработки промежуточных решений (объектов TClosed) с упрощённым (грубым) прогнозом

Шаг	Исх. прогноз	Текущее решение	Шаг	Исх. прогноз	Текущее решение	Шаг	Исх. прогноз	Текущее решение
1	0	нет	12	208	нет	23	215	216
2	184	нет	13	208	нет	24	215	216
3	188	нет	14	208	нет	25	215	216
4	188	нет	15	212	216	26	215	216
5	204	нет	16	212	216	27	219 *	216
6	204	нет	17	212	216	28	220 *	216
7	206	нет	18	212	216	29	220 *	216
8	206	нет	19	212	216	30	220 *	216
9	208	нет	20	212	216	31	222 *	216
10	208	нет	21	212	216	32	226 *	216
11	208	нет	22	212	216	33	226 *	216

Испытания на большом наборе случайных графов показало, что на несимметричных орграфах уточнённый прогноз может слегка замедлять алгоритм. Зато на ненаправленных графах и симметричных орграфах (как в рассмотренном выше примере) он даёт ощутимый выигрыш.

33.10. Итоги

- Замкнутая задача Гамильтона состоит в поиске кратчайшего (или длиннейшего) пути посещения *всех* вершин графа, который начинается и заканчивается в одной (любой) вершине графа.
- Так же, как и разомкнутая, замкнутая задача может быть решена полным перебором возможных путей (экспоненциально сложным по времени).
- Основной быстрый алгоритм решения замкнутой задачи основан на построении эквивалентного двудольного графа, и применении к нему венгерского алгоритма с целью поиска минимального замкнутого потока.
- Поскольку венгерский алгоритм на полном графе может дать несколько замкнутых потоков (факторов), для получения единственного цикла необходимо отключать часть дуг графа.
- Порядок выбора отключаемых дуг определяет точность и скорость алгоритма. Для сокращения вычислений поочерёдно отключают дуги только тех вершин, которые образуют один из кратчайших факторов очередного промежуточного решения.
- Время вычислений может быть снижено более точным прогнозом конечной стоимости, соответствующей некоторому промежуточному решению. Этот прогноз вычисляется рекурсивно и основан на стягивании факторов в вершины.

33.11. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 265
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 34
Гамильтон: комбинации методов

В главах 32 и 33 получены решения *разомкнутой (путь)* и *замкнутой (цикл)* задач Гамильтона. Здесь сопоставим эффективность упомянутых алгоритмов применительно к двум типам графов: ориентированным и неориентированным. Эффективный метод штрафования вершин (*разомкнутая* задача) пригоден лишь для неориентированных графов. С другой стороны, венгерскому алгоритму (для *замкнутой* задачи) трудней даются неориентированные графы.

Не даст ли перекрёстное применение методов для замкнутой и разомкнутой задач дополнительный эффект? В следующей таблице показана применимость их к различным сочетаниям Гамильтоновых задач. Указаны сочетания, соответствующие уже решённым задачам. Далее сосредоточимся на двух комбинациях, отмеченных в таблице знаками вопроса. Попытаемся приспособить венгерский метод к разомкнутой задаче, и наоборот: метод штрафования вершин к замкнутой.

Табл. 34-1 — Применимость двух методов
к вариантам гамильтоновых задач для разных типов графов

Метод решения	Разомкнутая задача		Замкнутая задача	
	Орграф	Граф	Орграф	Граф
Метод штрафования вершин	—	*Эффективен*	—	???
Поиск факторов венгерским методом	???	—	*Эффективен*	*Не вполне эффективен*

34.1. Разомкнутая задача на орграфе

Вспомним формулировку разомкнутой задачи: дан сильно связанный орграф и две его (разные) вершины X и Y. Необходимо найти кратчайший путь из вершины X в вершину Y, проходящий через все прочие вершины графа.

Эту задачу сведём к уже решённой замкнутой задаче на орграфе. Добавим в граф ещё одну вершину Z, и соединим её двумя «бесплатными» дугами с вершинами X и Y так, как показано на рис. 33-1. Здесь стоимость добавленных дуг принята равной нулю.

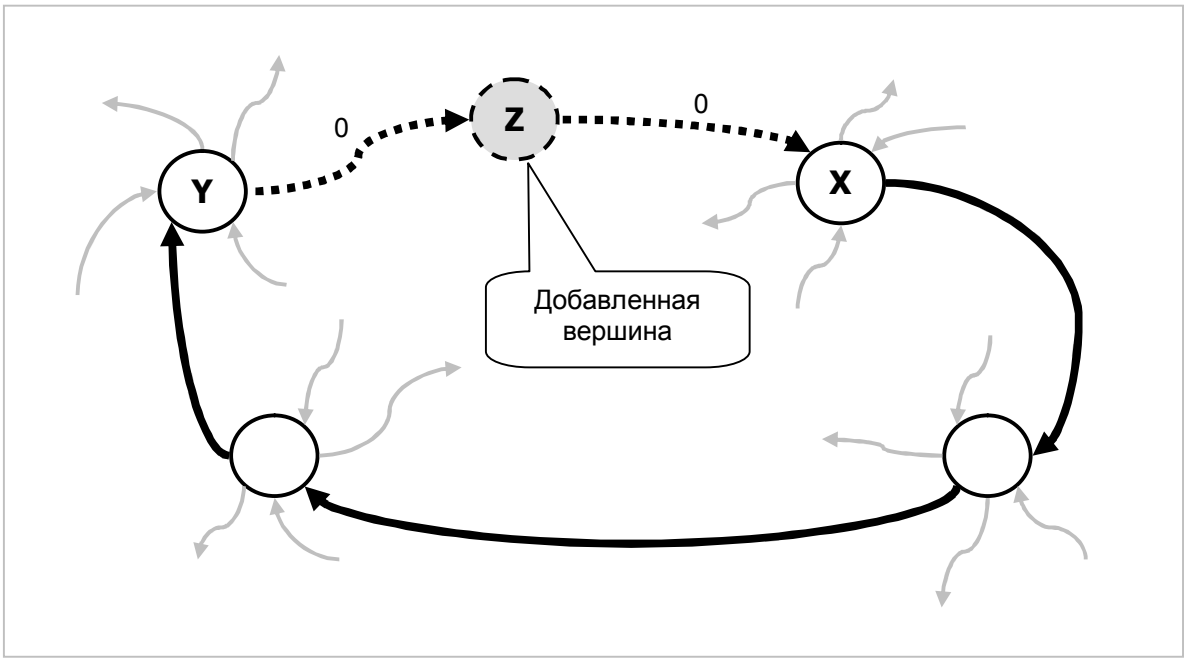


Рис. 34-1 — Вставка вспомогательной вершины Z

Найдём в этом орграфе цикл Гамильтона уже известным методом на основе венгерского алгоритма. Этот цикл неминуемо пройдёт через вспомогательные дуги. Если теперь из решения выбросить вспомогательную вершину Z и её дуги, то останется искомый путь из X в Y (см. листинг 34-1).

Листинг 34-1 — Метод поиска гамильтонова пути в ориентированном графе

```
function TGraph.GenHamPathDir(aStart, aFin: TNode;
                              var aCost: integer): TBuffer;
var Closer,       // Замыкающая вершина
    Node: TNode;
begin
  Closer:= TNode.Create(0, Self);  // создаём вспомогат. замыкающую вершину
  InsertNode(Closer);              // вставляем её в граф
  Closer.MakeLink(aStart, 0);      // создаём связь Closer -> aStart
  aFin.MakeLink(Closer, 0);        // создаём связь aFin -> Closer
  Result:= GenHamilton(aCost);     // находим цикл Гамильтона
  // Удаляем из буфера результата замыкающую вершину
  repeat
    Node:= Result.Get as TNode;
    if Node = Closer then Break;
    Result.Put(Node);
  until false;
  aFin.RemoveLink(Closer);     // удаляем связь aFin -> Closer
  Closer.RemoveLink(aStart);   // удаляем связь Closer -> aStart
  RemoveNode(Closer);          // удаляем замыкающую вершину из графа
  Closer.Free;                 // уничтожаем вершину
end;
```

В отношении точности и скорости этот алгоритм обладает теми же характеристиками, что и основной алгоритм для замкнутого варианта Гамильтоновой задачи на основе венгерского алгоритма. В следующем листинге представлен универсальный метод решения разомкнутой Гамильтоновой задачи, учитывающий род графа (орграф или неориентированный).

Листинг 34-2 — Универсальный метод решения разомкнутой Гамильтоновой задачи

```
function TGraph.GenHamPath(aStart,        // начальная вершины
                           aFin: TNode;   // конечная вершины
                           var aCost: integer  // длина (стоимость) цепи
                           ): TBuffer;
begin
  if mDirect
    // Для орграфа используется основной метод поиска гамильтонова цикла
    then Result:= GenHamPathDir(aStart, aFin, aCost)
    // Для неориентированного графа - метод штрафования верши
    else Result:= GenHamPathStrip(aStart, aFin, nil, aCost);
end;
```

34.2. Замкнутая задача на неориентированном графе

Метод решения замкнутой задачи на основе венгерского алгоритма в целом справляется с графами обоих типов: и ориентированными, и неориентированными (симметричными орграфами). Однако симметричные графы даются ему существенно труднее, что видно на рис. 34-2, где с ростом размерности графа соотношение времён существенно растёт.

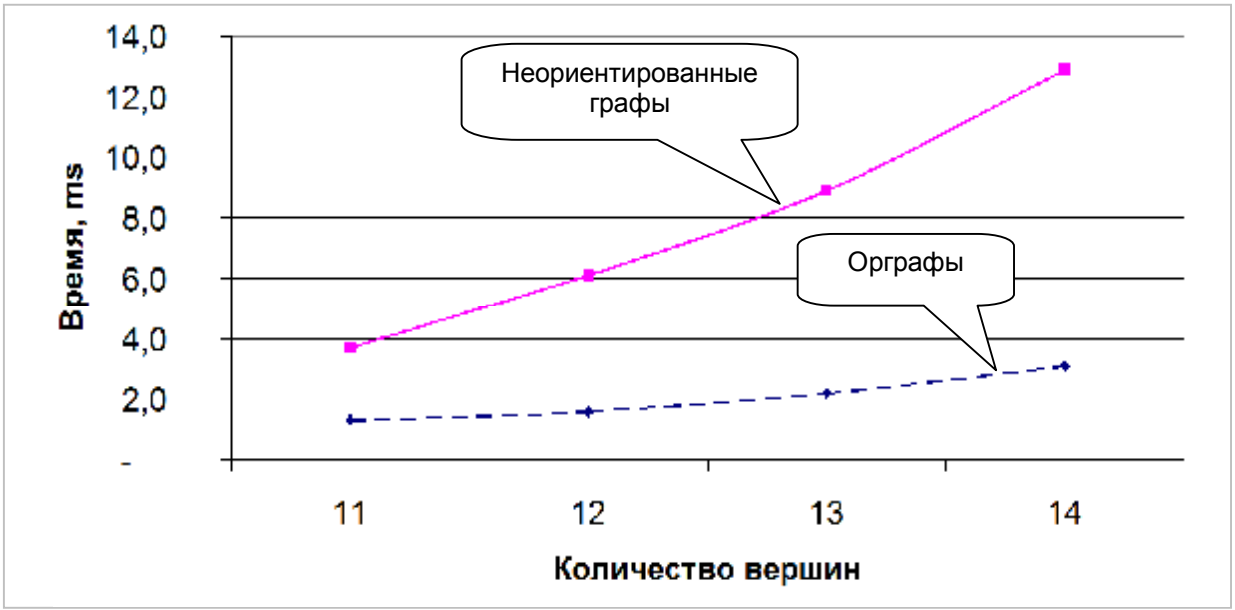


Рис. 34-2— Зависимость среднего времени решения замкнутой задачи от количества вершин (на основе венгерского алгоритма)

Причина в том, что при обработке симметричных графов порождается много коротких факторов (циклов), и соответственно много промежуточных решений.

Привлечём сюда быстрый метод штрафования вершин. Заметим, что замкнутый и разомкнутый варианты задачи отличаются лишь одним ребром. Найдём в графе кратчайшее ребро *X-Y*, мысленно удалим его, а затем найдём кратчайший путь из вершины *X* в вершину *Y*. Замкнув затем этот путь кратчайшим ребром, получим цикл, не слишком уступающий идеальному решению, а порой и совпадающий с ним. Исследуйте этот вариант самостоятельно.

Другой подход, дающий несколько лучший результат, похож на первый, и состоит в том, чтобы мысленно удалить из исходного графа одну вершину и рёбра, ей инцидентные. Как выбрать эту вершину и рёбра? Среди вершин графа найдём ту из них, для которой сумма двух кратчайших инцидентных ей рёбер окажется минимальной. Такую вершину *X* и двух её ближайших соседей *Y* и *Z* назовём *дружной* тройкой. Предположим далее, что уже найден кратчайший цикл Гамильтона, и случилось так (а вероятность этого велика), что *дружная* тройка выстроилась в этом цикле в том же порядке: *Y-X-Z* или *Z-X-Y*. Тогда можно ожидать, что тот же результат будет получен иным способом, а именно: нахождением кратчайшего гамильтонова пути из *Y* в *Z* и замыканием его временно исключённой вершиной *X* с её рёбрами. Разумеется, что упомянутый случай не гарантирован, но даже тогда, когда *дружная* тройка не содержится в идеальном Гамильтоновом цикле, полученный таким образом результат будет хорошим приближением к идеалу. Ниже представлен листинг, реализующий эту идею.

Листинг 34-3 — Ускоренный метод поиска Гамильтонова цикла
с привлечением метода штрафования вершин

```
function TGraph.GenHamilton_Quick(var aCost: integer): TBuffer;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка очередной вершины:
  // возвращаются две ближайшие вершины (aY, aZ)
  // и сумма расстояний

  function FindX(aX : TNode; var aY, aZ : TNode): integer;
  var L: TLink;
      YVal, ZVal : integer; // расстояния к соседям
  begin
    aY:= nil; aZ:= nil;
    YVal:= MaxInt; ZVal:= MaxInt;
    // Перебор исходящих линков:
    L:= aX.OutLinkFirst;
    // Всегда YVal <= ZVal
    while Assigned(L) do begin
      if L.mValue <= YVal then begin
        ZVal:= YVal;
        YVal:= L.mValue;
        aZ:= aY; aY:= L.mDest;
      end else if L.mValue < ZVal then begin
        ZVal:= L.mValue;
        aZ:= L.mDest;
      end;
      L:= aX.OutLinkNext;
    end;
    // Возвращаем сумму расстояний к ближайшим соседям
    Result:= YVal + ZVal;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск вершины с минимальной суммой расстояний
  // к двум ближайшим соседям

  function FindNode(var aX, aY, aZ : TNode): integer;
  var X, Y, Z : TNode;
      Cost, BestCost: integer;
  begin
    aX:= nil; aY:= nil; aZ:= nil;
    BestCost:= MaxInt;
```

```
    X:= NodeFirst;
    while Assigned(X) do begin
      Cost:= FindX(X, Y, Z);
      if Cost < BestCost then begin
        BestCost:= Cost;
        aX:= X; aY:= Y; aZ:= Z;
      end;
      X:= NodeNext;
    end;
    Result:= BestCost;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - -

var NodeX, NodeY, NodeZ : TNode;
    Cost, CostX : integer;

begin  { GenHamilton_Quick }

  // Для орграфа используем основной метод:
  if mDirect then begin
    Result:= GenHamilton(aCost);
    Exit;
  end;

  // Находим вершину X и двух её ближайших соседей Y, Z таких,
  // что сумма расстояний от X к двум этим соседям Y и Z минимальна

  CostX:= FindNode(NodeX, NodeY, NodeZ);

  // Строим кратчайшую цепь от вершины Y к вершине Z
  // с исключением вершины X

  Result:= GenHamPathStrip(NodeY, NodeZ, NodeX, Cost);
  if Cost < 0 then Exit;

  aCost:= Cost + CostX;  // общая стоимость цикла

  // Вставляем вершину в разрез между Z и Y, замыкая тем самым цикл
  Result.Push(NodeX);

  // Вращаем буфер, пока первая вершина графа не сместится в его начало
  NodeX:= NodeFirst;
  repeat
    NodeY:= Result.Get as TNode;
    if NodeY <> NodeX then Result.Put(NodeY);
  until NodeY = NodeX;
  Result.Push(NodeY);
end;
```

Для проверки идеи служит следующая программа (листинг 34-4). Здесь в качестве идеального решения для небольших графов принят результат полного перебора, а для крупных графов — то из двух решений, что даёт меньшую цену. Тестируются полные неориентированные графы (симметричные орграфы), длина рёбер которых равномерно распределена на интервале от 1 до 99.

Листинг 34-4 — Программа для сравнения основного и ускоренного алгоритмов поиска Гамильтонова цикла в неориентированных графах

```
{$APPTYPE CONSOLE}
uses
  SysUtils,  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

procedure Test(aRetry, aNodes: integer);

// Размер массивов для накопления погрешностей
const CArrSize = 50; // 50 квантов по 0,5% (всего 25 %)

var Gr : TGraph;           // исследуемый граф
    Res : TBuffer;         // результат - гамильтонов цикл
    Cost, CostF: integer; // стоимости

    TF, TH, TQ : integer;    // время накопленное
    MaxH, MaxQ : extended;   // максимальные погрешности
    SumH, SumQ : extended;   // суммы погрешностей
    Start : TDateTime;       // для засечки времени

    Cnt : integer;       // счётчик решений
    Delta : integer;     // абсолютное отклонение
    pc :  extended;      // отношение в %
    Index: integer;      // индекс в статистике

    // массивы для накопления распределения погрешностей (25% с шагом 0,5%)
    ArrH, ArrQ : array [0..CArrSize] of integer;

   // Вывод статистики
  procedure Expo(const aFileName: String);
  const CLine = '-------------------------------------------------------';
  var i : integer;
      exist: boolean;
  begin
    exist:= true;
    if aFileName <> '' then begin
      Assign(Output, aFileName);
      exist:= FileExists(aFileName);
      if exist then Append(Output) else Rewrite(Output)
    end;
    if not exist then Writeln(aFileName);
    Writeln(CLine);
    Writeln('Retry=', aRetry:4,  '  Nodes=', aNodes:4);
    Writeln(CLine);
    Writeln('                   F         H         Q');
    // Среднее время, мс
    Writeln('Time (ms)= ', TF/aRetry:10:1, TH/aRetry:10:1, TQ/aRetry:10:1);
    // максимальное отклонение
    Writeln('Max  (%%)= ', 0.0:10:1, (MaxH-100):10:1, (MaxQ-100):10:1);
    // средне квадратичное отклонение
    Writeln('Aver (%%)= ', 0.0:10:1,
             Sqrt(SumH/aRetry):10:1,
             Sqrt(SumQ/aRetry):10:1);
```

```
    Writeln(CLine);
    // Таблицы распределения
    if Abs(aNodes) < 15 then begin
      for i:= 0 to CArrSize-1 do Writeln(i:2, #9, 100*ArrH[i]/aRetry:6:1);
      Writeln(CLine);
    end;
    if not Gr.mDirect then begin
      for i:= 0 to CArrSize-1 do Writeln(i:2, #9, 100*ArrQ[i]/aRetry:6:1);
      Writeln(CLine);
    end;
    if aFileName <> '' then begin
      Close(Output);  Assign(Output, '');    Rewrite(Output);
    end;
  end;

begin  { Test }
    FillChar(ArrH, SizeOf(ArrH), 0);
    FillChar(ArrQ, SizeOf(ArrH), 0);
    // время накопленное
    TF:=0; TH:=0; TQ:=0;
    // максимальные погрешности
    MaxH:= 0; MaxQ:= 0;
    // суммы погрешностей
    SumH:= 0; SumQ:= 0;
    for Cnt:= 1 to aRetry do begin
      Gr:= TGraphChars.GenFull((aNodes>0), 0, 99, Abs(aNodes));
      {-------- Full -------}
      CostF:= 0;
      if Abs(aNodes) < 15 then begin
        Start:= Now;
        Res:= Gr.GenHamilton_Full(CostF);
        Inc(TF, MilliSecondsBetween(Start, Now));
        Res.Free;
      end;
      {-------- Main -------}
      Start:= Now;
      Res:= Gr.GenHamilton(Cost);
      Inc(TH, MilliSecondsBetween(Start, Now));
      Res.Free;
      if CostF=0 then CostF:= Cost; // если не вычислялся перебор
      Delta:= Cost - CostF;         // абсолютное отклонение
      if Delta=0 then Write('.') else Write('*');
      pc:= 100 * Cost / CostF;      // относительное отклонени pc >= 100 %
      if MaxH < pc then MaxH:= pc;  // максимальное отклонение
      SumH:= SumH + (pc-100)*(pc-100);
      // Формируем таблицу распределения с шагом 0,5 %:
      Index:= Round(2*(pc-100));
      if Index < CArrSize then Inc(ArrH[Index]);
      {-------- Quick -------}
      if not Gr.mDirect then begin
        Start:= Now;
        Res:= Gr.GenHamilton_Quick(Cost);
        Inc(TQ, MilliSecondsBetween(Start, Now));
        Res.Free;
        Delta:= Cost - CostF;     // абсолютное отклонение
        // Из двух решений (Cost и CostF) выбираем лучшее
        if Delta<0 then begin
          Delta:= 0;
          CostF:= Cost;
        end;
        if Delta=0 then Write('.') else Write('@');
        pc:= 100 * Cost / CostF;      // относительное отклонени pc >= 100 %
```

```
        if MaxQ < pc then MaxQ:= pc;  // максимальное отклонение
        SumQ:= SumQ + (pc-100)*(pc-100);
        // Формируем таблицу распределения с шагом 0,5 %:
        Index:= Round(2*(pc-100));
        if Index < CArrSize then Inc(ArrQ[Index]);
      end;
      Gr.Free;
    end; // for
    Writeln(#7);
    // Вывод статистики
    Expo('');   Expo('Out.txt');
end;
{-------------------------------------------------------}
var Nodes: integer;
    Retry : integer;
begin
    Write('Retry= '); Readln(Retry);
    if Retry = 0 then Exit;
    repeat
      Write('Nodes= '); Readln(Nodes);
      if Abs(Nodes) < 4 then Break;
      Test(Retry, Nodes);
    until false;
end.
```

Рассмотрим результаты испытаний. Сначала оценим точность основного и ускоренного методов в сравнении с идеалом, полученным полным перебором. В табл. 34-2 дан усреднённый результат, полученный для основного метода на небольших графах с количеством вершин от 11 до 14. Всего испытано 4 тысячи полных симметричных графов (по тысяче для каждого размера), длина рёбер распределена равномерно в интервале от 1 до 99.

Табл. 34-2 — Отклонение стоимости от идеала для *ОСНОВНОГО* алгоритма (на малых графах)

Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %
0,0	95,75	5,0	0,23
0,5	0,88	5,5	0,08
1,0	0,63	6,0	0,05
1,5	0,55	6,5	0,05
2,0	0,30	7,0	0,05
2,5	0,35	7,5	0,05
3,0	0,38	8,0	0,08
3,5	0,25	8,5	0,05
4,0	0,10	9,0	0,00
4,5	0,13	9,5	0,03

Результаты, полученные для ускоренного метода, показаны в табл. 34-3.

Табл. 34-3 — Отклонение стоимости от идеала
для *ускоренного* алгоритма (на малых графах)

Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %
0,0	41,00	5,0	2,45	10,0	1,43
0,5	3,13	5,5	2,65	10,5	1,50
1,0	2,15	6,0	2,08	11,0	1,65
1,5	2,75	6,5	1,93	11,5	1,03
2,0	2,68	7,0	1,78	12,0	1,00
2,5	2,23	7,5	1,40	12,5	0,90
3,0	2,58	8,0	1,58	13,0	1,00
3,5	2,58	8,5	1,65	13,5	0,85
4,0	2,15	9,0	1,13	14,0	0,83
4,5	1,88	9,5	1,05	14,5	0,85

На рис. 34-3 и рис. 34-4 данные таблиц представлены в графической форме. Первый столбец, соответствующий точному совпадению, на диаграммах не показан.

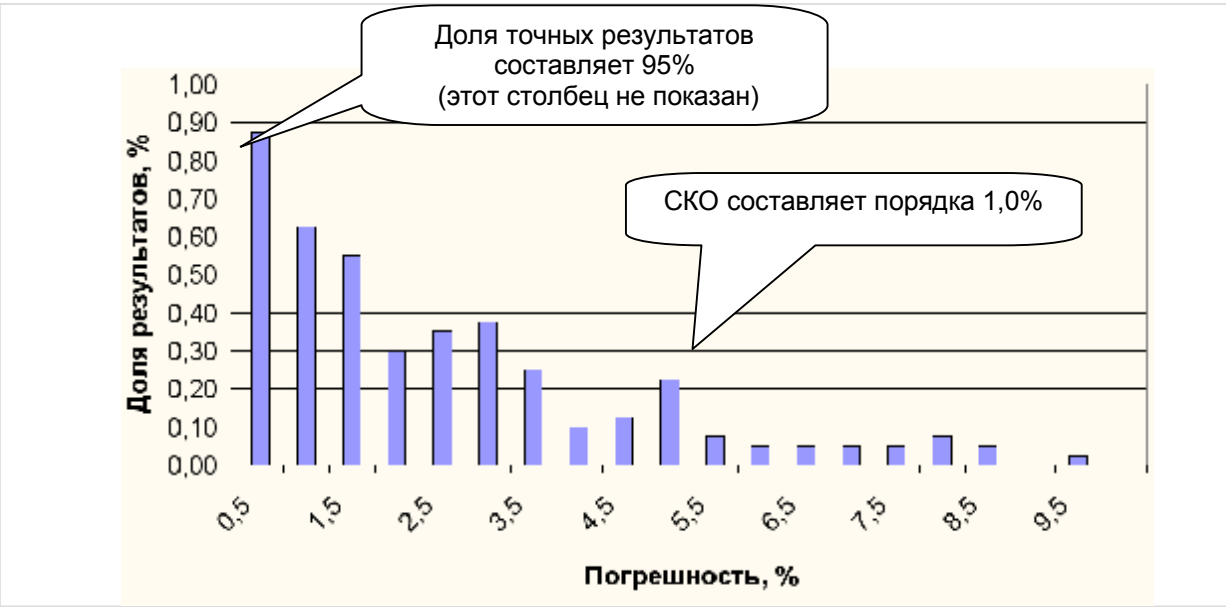


Рис. 34-3 — Распределение погрешностей для ***ОСНОВНОГО*** алгоритма
(количество вершин от 11 до 14)

Оценим точность алгоритмов по двум параметрам: a) по доле точных совпадений стоимости с идеалом, и b) по среднеквадратичному отклонению стоимости от идеала. Доля точных совпадений основного и ускоренного методов составила соответственно 95% и 41%, а среднеквадратичные отклонения — 1% и 8,5%. Это говорит о несколько сниженной точности ускоренного алгоритма (за всё надо платить!). Однако в сравнении с жадным алгоритмом эта точность весьма высока.

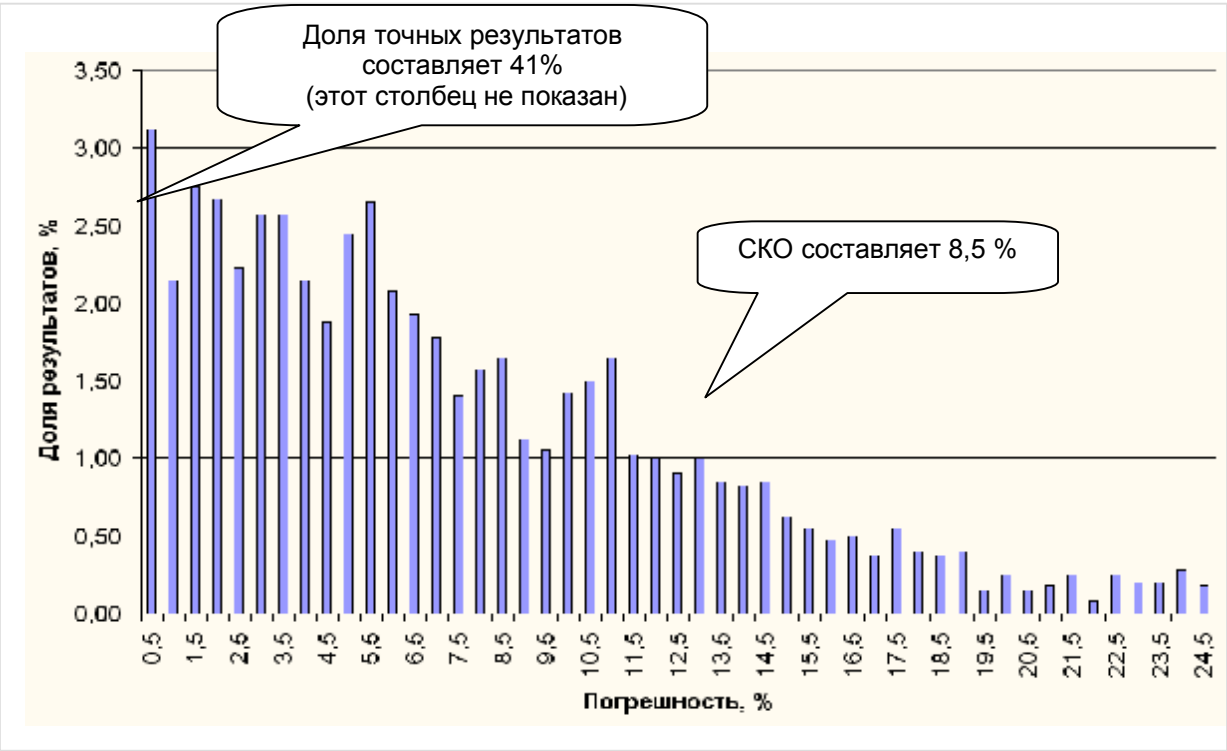


Рис. 34-4 — Распределение погрешностей для *ускоренного* алгоритма (количество вершин от 11 до 14)

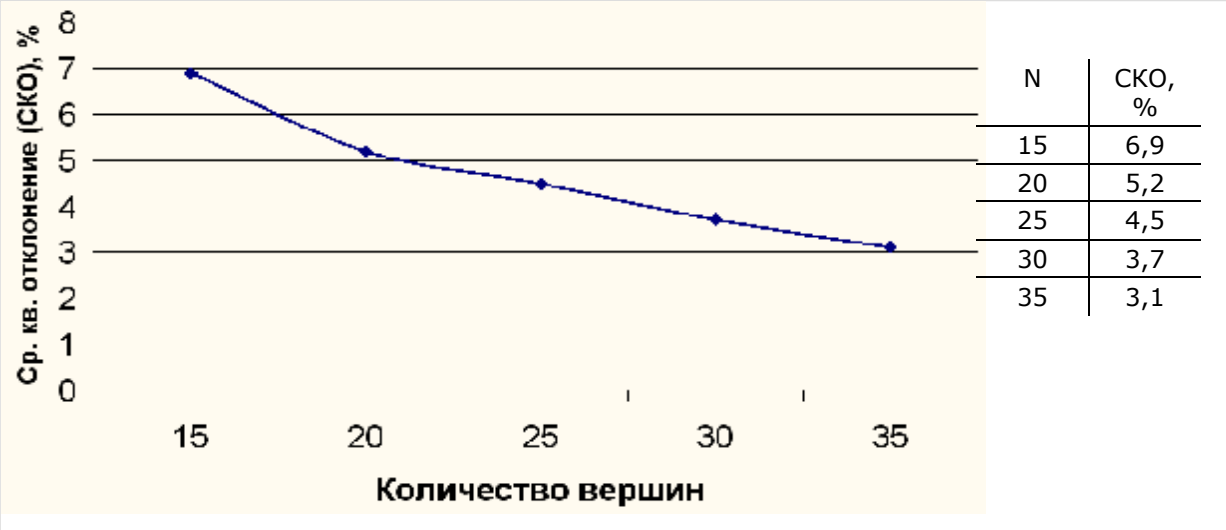
А какова точность при большем количестве вершин? Идеал, даваемый полным перебором, здесь не доступен, потому при сравнении основного и ускоренного методов условным идеалом будет результат с меньшей стоимостью из этих двух. Ниже даны усреднённые результаты обработки графов с числом вершин 15, 20, 25, 30 и 35; обработано по тысяче симметричных случайных полных графов каждого размера, длина их рёбер распределена равномерно в интервале от 1 до 99.

Табл. 34-4 — Отклонение от условного идеала ускоренного алгоритма (на случайных графах от 15 до 35 вершин)

Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %
0,0	39,36	5,0	2,68	10,0	0,70
0,5	5,90	5,5	2,10	10,5	0,80
1,0	5,16	6,0	1,84	11,0	0,78
1,5	5,38	6,5	1,96	11,5	0,52
2,0	4,28	7,0	1,00	12,0	0,38
2,5	4,66	7,5	1,44	12,5	0,50
3,0	3,96	8,0	1,32	13,0	0,42
3,5	3,10	8,5	1,16	13,5	0,14
4,0	2,98	9,0	0,76	14,0	0,22
4,5	3,46	9,5	0,74	14,5	0,38

Рис. 34-5 — Распределение погрешностей для ускоренного алгоритма при количестве вершин от 15 до 35 (в сравнении с основным алгоритмом)

Отметим, что с ростом числа вершин точность ускоренного метода, по крайней мере, не ухудшается, а среднеквадратичное отклонение даже уменьшается (рис. 34-6).



N	СКО, %
15	6,9
20	5,2
25	4,5
30	3,7
35	3,1

Рис. 34-6 — Зависимость среднеквадратичного отклонения от количества вершин для ускоренного метода

Как и ожидалось, точность ускоренного метода (штрафованием вершин), в сравнении с основным (венгерским), оказалась несколько ниже. Оправдано ли это снижение точности повышением скорости? В табл. 34-5 и на рис. 34-7 дано усреднённое время обработки выше упомянутых графов.

Табл. 34-5 — Среднее время обработки полных симметричных графов разных размеров, ms

Количество вершин	Среднее время поиска, ms		Отношение времён
	Основной	Ускоренный	
15	16,7	7,6	2,2
20	67,4	29,4	2,3
25	196,6	76,8	2,6
30	602,0	161,0	3,7
35	1454,0	303,0	4,8

Очевидно, что ускоренный алгоритм подтверждает своё название: при поиске Гамильтоновых циклов в *симметричных* графах метод штрафования вершин даёт ощутимый выигрыш во времени. И он растёт с ростом размера графа. Так, если на **15** вершинах наблюдается двукратный выигрыш, то при **35** — уже пятикратный.

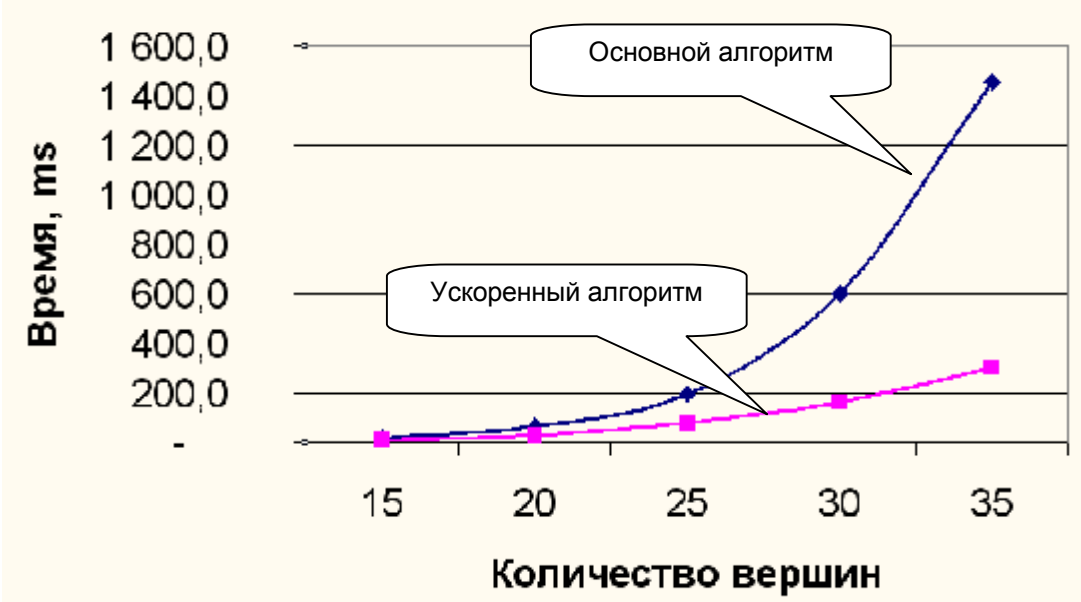


Рис. 34-7 — Среднее время обработки симметричных полных графов разных размеров

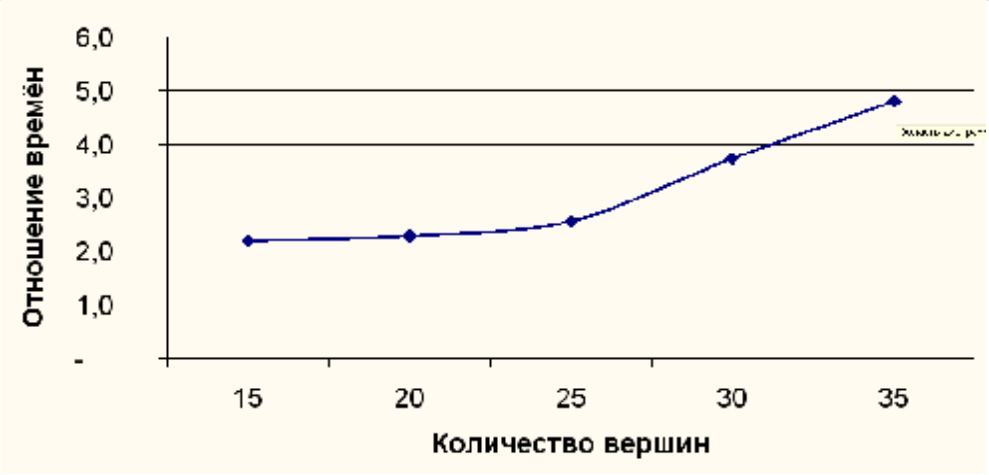


Рис. 34-8 — Отношение времени обработки симметричных графов основным и ускоренным алгоритмами

34.3. Итоги

- Для разомкнутой и замкнутой задач Гамильтона применяют два принципиально разных алгоритма: алгоритм штрафования вершин (для неориентированных графов) и алгоритм поиска циклов (на основе венгерского метода). Однако в чистом виде эти основные алгоритмы не решают всего круга гамильтоновых задач.
- *Разомкнутую* задачу для *орграфа* можно решить слегка изменённым венгерским методом, разработанным для замкнутой задачи. Модификация состоит в добавлении к исходному графу одной вспомогательной вершины с двумя дугами нулевой длины.
- *Замкнутая* задача для *неориентированного графа* быстрее решается методом штрафования вершин, разработанным для разомкнутой задачи. Для этого выбирают подходящую вершину и временно отключают её, превращая цикл в гамильтонов путь. При этом несколько снижается точность решения.

34.4. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 265
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Глава 35
Задачи коммивояжёра

К задачам Гамильтона мы пришли от обсуждения родственных им задач коммивояжёра (глава 32). Напомню, что в *разомкнутой* и *замкнутой* задачах *коммивояжёра* разыскивается кратчайший путь, проходящий через все вершины графа, при этом *допускается* неоднократное посещение некоторых вершин (в отличие от задач Гамильтона, где это *не* допускается). Очевидно, что в сильно связанных графах задачи коммивояжёра всегда имеют решения. Причём получить эти решения можно через решения задач Гамильтона. Хотя последние гарантировано решаются лишь на полных графах, где каждая вершина прямо соединена рёбрами либо дугами со всеми остальными вершинами графа.

Такое решение возможно с привлечением алгоритма поиска кратчайших путей методом Флойда, рассмотренного в главе 15. Пусть дан сильно связанный граф, где любая вершина достижима из любой другой вершины, либо прямо, либо посредством цепочки, проходящей через другие вершины. Дополним этот граф рёбрами (дугами) так, чтобы он стал насыщенным, причём дополнительным рёбрам или дугам предварительно назначим достаточно большой «бесконечный» вес. Далее методом Флойда построим в этом графе распределённую по вершинам карту дальних связей и выполним компрессию рёбер (дуг) так, чтобы вес любого прямого ребра (дуги) не превышал веса кратчайшего пути, проходящего между вершинами альтернативным путём.

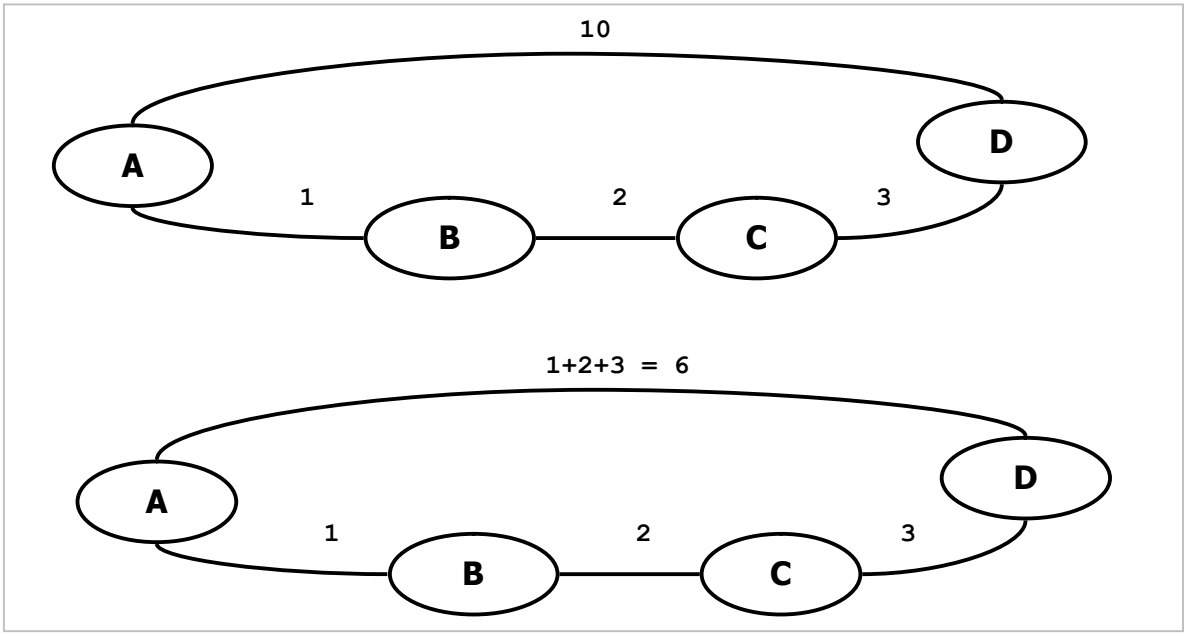


Рис. 35-1 — Компрессия ребра A-D

За примером обратимся к Рис. 35-1 где прямое ребро *A-D* изначально имело вес 10. Кратчайший путь между этими вершинами проходит через вершины *B* и *C* и составляет 6 единиц, поэтому после компрессии ребро *A-D* обретает вес равный 6 единицам.

В созданном таким путём насыщенном графе решается замкнутая либо разомкнутая задача Гамильтона, после чего в полученном решении некоторые предварительно сжатые рёбра (дуги) заменяются соответствующими кратчайшими путями (цепочками вершин).

35.1. Замкнутая задача коммивояжёра

Ниже представлен основанный на этих рассуждениях метод решения замкнутой задачи коммивояжёра.

```
function TGraph.GenTravelCycle(var aCost: integer): TBuffer;
var FullGraph : TGraph;
begin
  // Создаём копию данного графа
  FullGraph:= Self.Copy as TGraph;
  // Дополняем его до насыщенного,
  // причём добавочным линкам назначаем бесконечный вес
  FullGraph.MakeFull(MaxInt div 2);
  // Создаём карту маршрутов и сжимаем линки
  // заменяя длины дуг кратчайшими расстояниями
  FullGraph.Compress;
  // В насыщенном графе находим Гамильтонов цикл
  Result:= FullGraph.GenHamilton(aCost);
  // Заменяем в этом цикле длинные дуги кратчайшими путями
  Decompress(Result);
  // Заменяем копии вершин их оригиналами из Self
  ReplaceNodes(Self, Result);
  // Удаляем копию графа
  FullGraph.Free;
end;
```

Далее рассмотрим вызываемые здесь процедуры и функции. Прежде всего, во избежание «деформации» исходного графа, решение выполняется на удаляемой впоследствии его копии **FullGraph.** Метод **MakeFull** насыщает копию графа рёбрами либо дугами «бесконечного» веса.

```
procedure TGraph.MakeFull(aValue: integer);
var Node: TNode;
    N : TNode;
    L: TLink;
begin
  // Дополняем граф линками до насыщенного
  Node:= NodeFirst as TNode;
  // Перебор вершин
  while Assigned(Node) do begin
    Self.PosPush; // Сохр. позицию перебора
    N:= NodeFirst as TNode;
    while Assigned(N) do begin
      if N <> Node then begin
        // Получить линк на вершину Node --> N
        L:= Node.GetLink(N);
        // Если линк отсутствует, то создать его
        if not Assigned(L) then SetLink(Node, N, aValue);
      end;
      N:= NodeNext as TNode;
    end;
    Self.PosPop;  // Восст. позицию перебора
```

```
    Node:= NodeNext as TNode;
  end;
end;
```

Далее метод **Compress** строит распределённую по вершинам карту дальних связей (**InitMap_Floyd**), после чего заменят вес некоторых прямых дуг, если этот вес превышает вес «окольного» пути к соответствующей вершине.

```
procedure TGraph.Compress;
var  N: TNode;
     L: TLink;
    FL: TFarLink;
begin
  // Строим карту с дальними указателями:
  InitMap_Floyd;
  // Заменяем длины дуг кратчайшими путями:
  N:= NodeFirst;
  while Assigned(N) do begin
    // Перебор дальних указателей из вершины N:
    FL:= N.mFarLinks.GetFirst as TFarLink;
    while Assigned(FL) do begin
      if (FL.mNodeFar <> N) then begin
        // Ищем линк из данной вершины N --> mNodeFar
        L:= N.GetLink(FL.mNodeFar);
        // Ели он существует, меняем длину на кратчайшее расстояние
        if Assigned(L) and (L.mValue > FL.mDist)
            // заменяем кратчайшим расстоянием
            then L.mValue:= FL.mDist;
      end;
      FL:= N.mFarLinks.GetNext as TFarLink;
    end;
    N:= NodeNext;
  end;
end;
```

В полученном таким образом графе находится Гамильтонов цикл, вершины которого помещаются в буфер **Result**.

```
  // В насыщенном графе находим Гамильтонов цикл
  Result:= FullGraph.GenHamilton(aCost);
```

Далее выполняется декомпрессия цикла с использованием всё той же распределённой карты дальних связей. Здесь происходит извлечение из начала очереди пары вершин со вставкой между ними промежуточных вершин, если таковые существуют, и вся эта цепочка возвращается по кругу в конец очереди.

```
procedure Decompress(aBuf: TBuffer);
var i: integer;
    Node, Next : TNode;
    Link: TFarLink;
begin
  Node:= aBuf.Get as TNode;
  for i:=1 to aBuf.GetCount do begin
    Next:= aBuf.Get as TNode;
    aBuf.Put(Node);
    repeat
      Link:= Node.GetFarLink(Next);
```

```
      if Link.mNodeNear = Next then Break;
      Node:= Link.mNodeNear;
      aBuf.Put(Node);
    until false;
    Node:= Next;
  end;
  aBuf.Put(Next);
end;
```

В завершение копии вершин, содержащиеся в буфере результата, заменяются оригиналами из исходного графа.

```
procedure ReplaceNodes(aGr: TGraph; aBuf: TBuffer);
var Node, N: TNode;
    i: integer;
begin
  for i:=1 to aBuf.GetCount do begin
    // Выборка исходной вершины Node
    Node:= aBuf.Get as TNode;
    // Получение копии N
    N:= aGr.GetNode(Node);
    // и засылка в тот же буфер
    aBuf.Put(N);
  end;
end;
```

35.2. Разомкнутая задача коммивояжёра

Аналогично решается разомкнутая задача коммивояжёра. С тем дополнением, что перед вызовом метода **GenHamPath** находятся копии параметров **aStart, aFin** во вспомогательном графе (копии начальной и конечной вершин).

```
function TGraph.GenTravelPath(aStart, aFin: TNode; var aCost: integer):
TBuffer;
var FullGraph : TGraph;
    Start, Fin : TNode;
begin
  // Создаём копию данного графа
  FullGraph:= Self.Copy as TGraph;
  // Дополняем его до насыщенного,
  // причём добавочным линкам назначаем бесконечный вес
  FullGraph.MakeFull(MaxInt div 2);
  // Создаём карту маршрутов и сжимаем линки
  // заменяя длины дуг кратчайшими расстояниями
  FullGraph.Compress;
  // В копии графа находим вершины начала и конца пути
  Start:= FullGraph.GetNode(aStart);
  Fin:= FullGraph.GetNode(aFin);
  // В насыщенном графе находим Гамильтонов путь
  Result:= FullGraph.GenHamPath(Start, Fin, aCost);
  // Заменяем в этом цикле длинные дуги кратчайшими путями
  Decompress(Result);
  // Заменяем копии вершин их оригиналами из Self
  ReplaceNodes(Self, Result);
  // Удаляем копию графа
  FullGraph.Free;
end;
```

35.3. Испытание методов

Испытание методов выполнялось следующей программой.

```
program Graph_travel;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  SetUtils in '..\Common\SetUtils.pas',
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas';

var Gr : TGraph;
    Res: TBuffer;
    Cost: integer;
    Na, Nb : TNode;

begin
  try
    Gr:= nil;
    Gr:= TGraphChars.Load('Test-8.txt'); // Неориентированный граф
    // Gr:= TGraphChars.Load('Test-8d.txt'); // Неориентированный граф
    Gr.Expo;

    Writeln('- - - - - - - - - - - - - -');
    Writeln('         Cycle');
    Res:= Gr.GenTravelCycle(Cost);
    Res.Expo;
    Writeln('Cost= ',Cost);
    Res.Free;

    Writeln('- - - - - - - - - - - - - -');
    Writeln('         Path');
    Na:= (Gr as TGraphChars).GetNode('A');
    Nb:= (Gr as TGraphChars).GetNode('B');
    Res:= Gr.GenTravelPath(Na, Nb, Cost);
    Res.Expo;
    Writeln('Cost= ',Cost);
    Res.Free;
    Writeln('- - - - - - - - - - - - - -');
    Gr.Free;
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
end.
```

Испытания на ненаправленном графе (Рис. 21-7), дало следующие результаты (в скобках дан вес маршрута):

Замкнутый путь коммивояжёра (52):

A-> F-> E-> H-> B-> G-> A-> D-> C-> A

Разомкнутый путь коммивояжёра из *A* в *B* (49):

A-> D-> C-> A-> F-> E-> H-> B-> G-> B

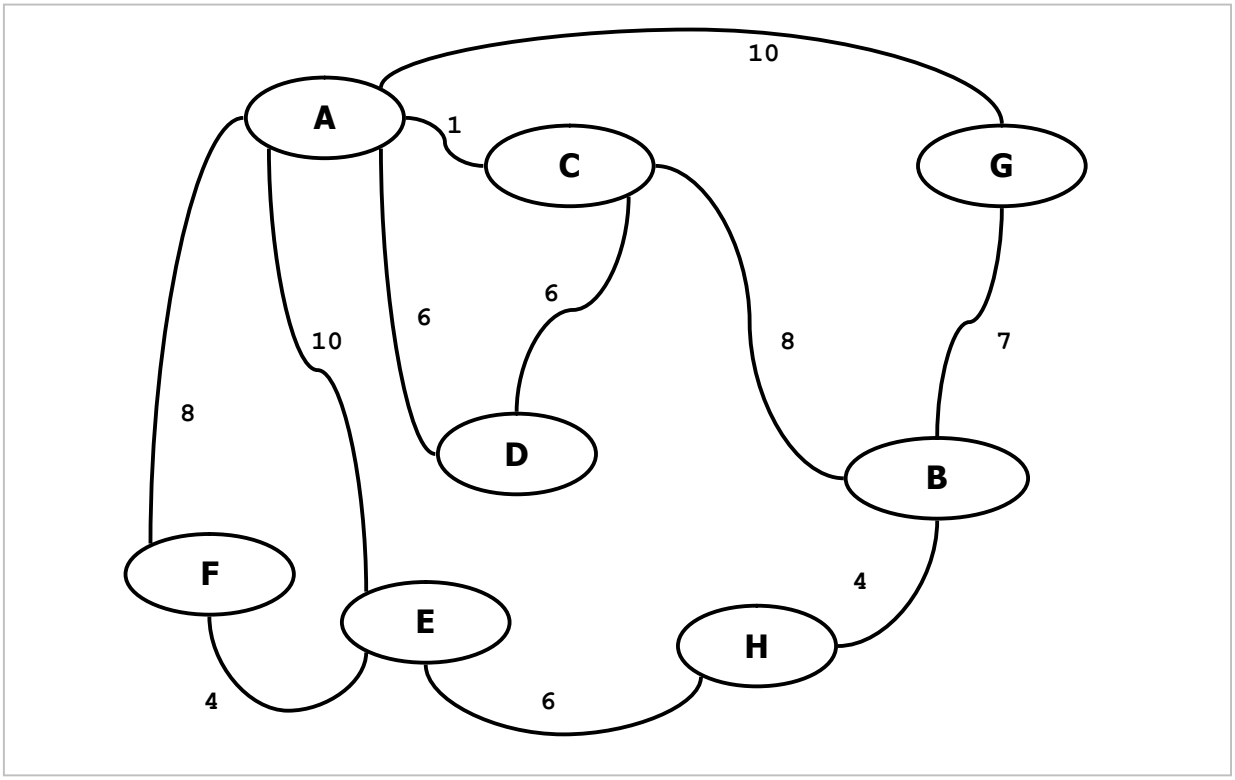


Рис. 35-2 — Взвешенный неориентированный граф

```
Взвешенный неориентированный граф
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

Ниже даны результаты для ориентированного графа (Рис. 35-3).

Замкнутый путь коммивояжёра (76):

A-> G-> B-> H-> E-> F-> A-> G-> B-> C-> D-> A

Разомкнутый путь коммивояжёра из *A* в *B* (93):

A-> G-> B-> H-> E-> F-> A-> G-> B-> C-> D-> A-> G-> B

Разомкнутый путь коммивояжёра из *B* в *A* (59):

B-> H-> E-> F-> A-> G-> B-> C-> D-> A

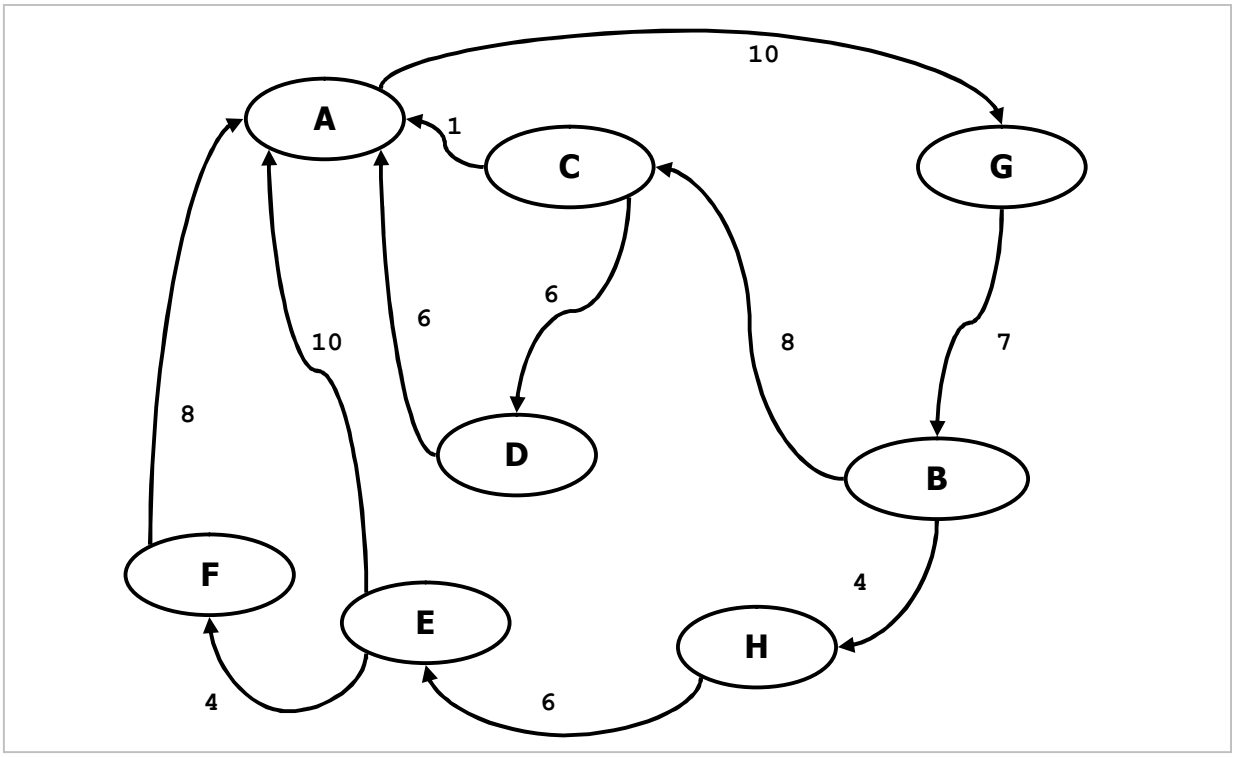


Рис. 35-3 — Взвешенный ориентированный граф

```
Взвешенный ориентированный граф
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

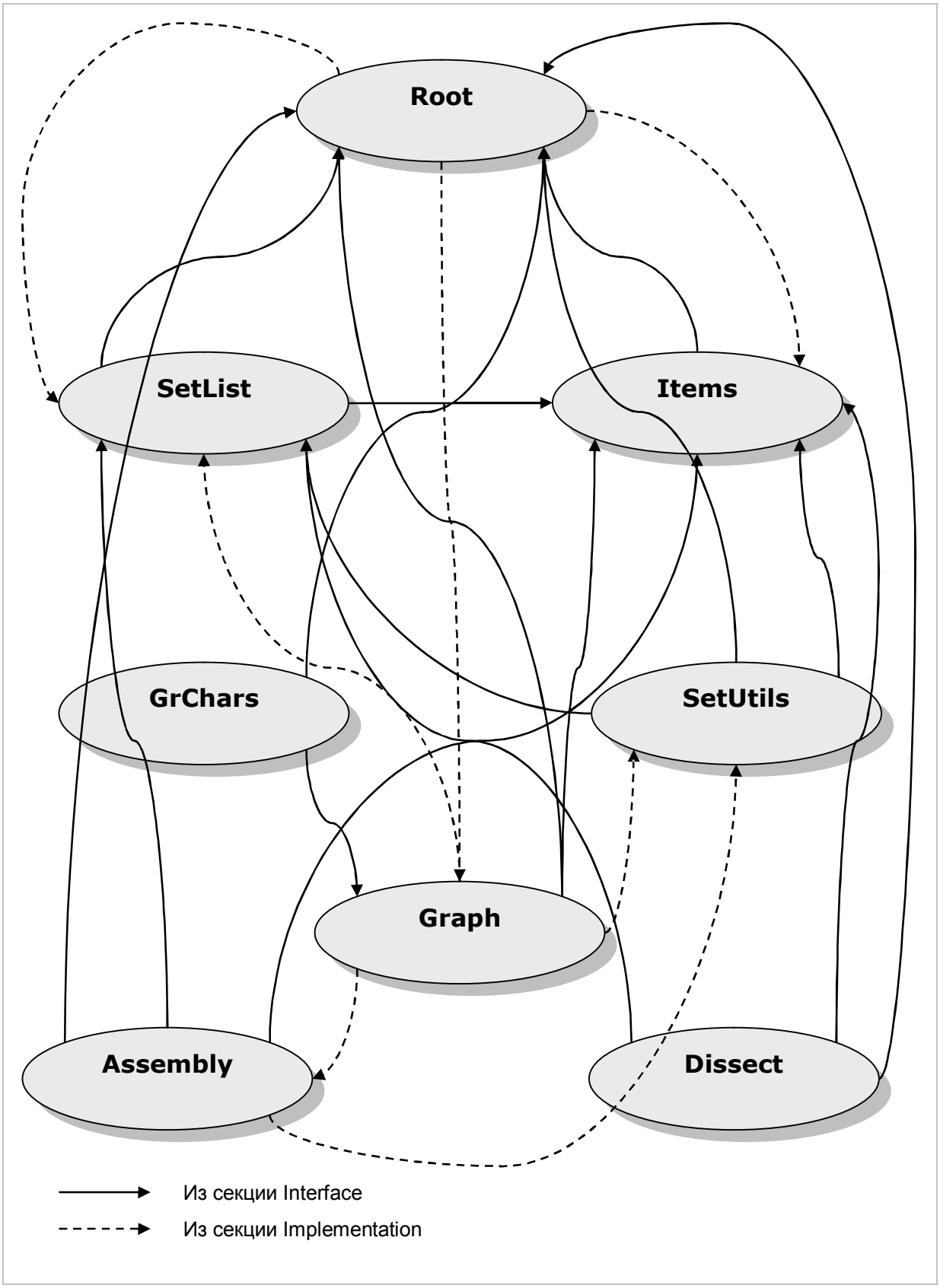
35.4. Итоги

- Замкнутая и разомкнутая задачи коммивояжёра всегда имеют решения на сильно связанном графе.
- Для использования методов задач Гамильтона к решению задач коммивояжёра необходимо дополнить граф до насыщенного и выполнить на нём процедуру компрессии рёбер (дуг).
- После отработки алгоритмов Гамильтона необходимо скорректировать решение, заменив в нём некоторые рёбра (дуги) кратчайшими путями через промежуточные вершины.

35.5. Что почитать

№		Автор(ы)	Название	Главы, страницы
	1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
	2	Басакер Р. Саати Т.	Конечные графы и сети	
	3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
	4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
	5	Деревенец О.В.	Песни о Паскале	
	6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√	7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 265
	8	Липский В.	Комбинаторика для программистов	
	9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
	10	Макконнелл Дж.	Основы современных алгоритмов	
	11	Турчин В.Ф.	Феномен Науки	
	12	Хаггарти Р.	Дискретная математика для программистов	

Приложение А
Взаимные ссылки модулей



Приложение B
Модуль Root

```
unit Root;
{$I Common.inc}
//*********************************************************************
//                        Корневой модуль                            *
//                                                                   *
// Содержит базовые объекты:                                         *
// TItem -- элемент                                                  *
// TSet  -- абстрактное множество                                    *
// TCostSet -- множество с оценкой для решения задач ЗНП и ЗНР       *
//*********************************************************************

interface

type

  // Перечисление результатов сравнения объектов

  TCompare = (cmpEq,     // совпадают или равны
              cmpLess,   // меньше
              cmpGreate, // больше
              cmpIncomp  // несравнимы (Incomparable)
             );

  // Базовый класс "элемент" -- прародитель всех классов

  TItem = class (TObject)
    public
      function Compare(arg: TItem): TCompare; virtual;
      function Copy: TItem;  virtual;{ abstract;}
      procedure Print(var aFile: TextFile); virtual; abstract;
      procedure Expo;
  end;

  // Класс "абстрактное множество"

  TSet = class (TItem)
    protected
      mCount: Longint;  // количество элементов множества
      procedure Clr(aDestroy: boolean); virtual; abstract;
    public
      // Базовые операции с множествами:
      procedure Add(arg : TSet); virtual; abstract;
      procedure Sub(arg : TSet); virtual; abstract;
      procedure Mul(arg : TSet); virtual; abstract;
      procedure ExOr(arg : TSet); virtual; abstract;
      function  TestIntersect(arg: TSet): boolean;
      function  Compare(arg: TItem): TCompare; override;
      function  Exist(arg : TItem): boolean; virtual; abstract;
      // Очистка
      procedure Clear;          // очистка без уничтожения элементов
      procedure ClrAndDestroy;  // очистка с уничтожением элементов
      // Вставка, удаление, копирование элементоа
      function  Insert(arg: TItem): boolean; virtual; abstract;
      procedure Delete(arg: TItem); virtual; abstract;
      procedure CopyItems(arg: TSet);
      // Последовательный перебор элементов
```

```
      function  GetFirst: TItem; virtual; abstract;
      function  GetNext: TItem; virtual; abstract;
      // Выбор по индексу и по элементу
      function  GetItem(index: integer): TItem;
      function  GetObject(aItem: TItem): TItem;
      // Сохранение-восстановление позиции перебора
      procedure PositionPush;  virtual; abstract;
      procedure PositionPop;  virtual; abstract;
      // Прочие операции
      procedure CoverToDissect; // Преобразование покрытия в разбиение
      function  GetCount: integer;
      function  Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      destructor Destroy;  override;
  end;

  // Класс "множество с оценкой", используется при поиске
  // наименьших разбиений (ЗНР) и покрытий (ЗНП)

  TCostSet = class (TItem)
    protected
      procedure Clr(aDestroy: boolean);
    public
      mCost: integer;       // оценка, стоимость элемента
      mSet : TSet;          // ссылка на множество
      mFlag  : boolean;     // флажок для пометок
      mDestroy : boolean;   // определяет способ уничтожения объекта
      constructor Create(aCost: integer; aSet: TSet; aDestroy : boolean);
      constructor CreateEmpty;
      destructor  Destroy;  override;
      function    Copy: TItem; override;
      procedure   Clear;
      procedure   ClrAndDestroy;
      function    Compare(arg: TItem): TCompare; override;
      procedure   Print(var aFile: TextFile); override;
      procedure   Append(arg : TCostSet);
      procedure   Insert(arg : TCostSet);
  end;

// Функция для создания множества

function CreateSet: TSet;

////////////////////////////////////////////////////////////////////////////
implementation
uses Items, SetList, Graph;
////////////////////////////////////////////////////////////////////////////

// Функция для создания множества

function CreateSet: TSet;
begin
  Result:= TSetList.Create;
end;

////////////////////////////////////////////////////////////////////////////
// TItem   -- Базовый класс "элемент" -- прародитель всех классов
////////////////////////////////////////////////////////////////////////////

function TItem.Compare(arg: TItem): TCompare;
begin
  if arg = Self
```

```
    then Result:= cmpEq
    else Result:= cmpIncomp;
end;


function TItem.Copy: TItem;
begin
  Result:= nil
end;

procedure TItem.Expo;       // Метод отображения на экране
begin
  if Assigned(Self) then Print(Output);
end;
////////////////////////////////////////////////////////////////////////////
// TSet  -- базовый класс "множество"
////////////////////////////////////////////////////////////////////////////

// Сравнение двух множеств

function TSet.Compare(arg: TItem): TCompare;
var p, q : TItem;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // один и тот же объект

  // Если аргумент не множество, то сравнивать нельзя
  if not (arg is TSet) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;

  // Сравнение элементов множеств

  PositionPush;             // сохр. позицию в данном множестве
  (arg as TSet).PositionPush;  // сохр. позицию в аргументе
  p:= GetFirst;             // взять первый элемент в данном множестве
  q:= (arg as TSet).GetFirst;  // взять первый элемент в аргументе
  while Assigned(p) and Assigned(q) do begin  // пока можно сравнивать
    Result:= p.Compare(q);                    // сравнение элементов
    if Result <> cmpEq then Break;            // выход, если не одинаковы
    p:= GetNext;  q:=(arg as TSet).GetNext;   // следующая пара
  end;
  if Result=cmpEq then begin                // если последние эл. совпали
    if GetCount < (arg as TSet).GetCount
      then Result:= cmpLess                 // данный меньше аргумента
    else if GetCount > TSet(arg).GetCount
      then Result:= cmpGreate               // данный больше аргумента
  end;
  (arg as TSet).PositionPop;    // восст. позицию в аргументе
  PositionPop;              // восст. позицию в данном множестве
end;

// Копирование множества целиком

function TSet.Copy: TItem;
var p, q : TItem;
begin
  Result:= CreateSet;
  PositionPush;
  p:= GetFirst;
  while Assigned(p) do begin
```

```
    if p is TSet
      then q:= p.Copy          // вложенное множество
      else q:= p;              // элементарный объект
    (Result as TSet).Insert(q);
    p:= GetNext;
  end;
  PositionPop;
end;

// Копирование элементов из аргумента в данное множество

procedure TSet.CopyItems(arg: TSet);
var t: TItem;
begin
  Clear;     // очистка множества без уничтожения элементов
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    Self.Insert(t);
    t:= arg.GetNext;
  end;
  arg.PositionPop;
end;

// очистка без уничтожения элементов

procedure TSet.Clear;
begin
  Clr(false)
end;

// очистка с уничтожением элементов

procedure TSet.ClrAndDestroy;
begin
  Clr(true)
end;

// Деструктор

destructor TSet.Destroy;
begin
  Clear;   // очистка без уничтожения элементов
  inherited;
end;

// Преобразование покрытия в разбиение.
// Элементами множества должны быть TCostSet!

procedure TSet.CoverToDissect;
var   i, j : integer;  // индексы
    Si, Sj : TCostSet; // текущие элементы
begin
  for i:= 1 to GetCount-1 do begin
    Si:= GetItem(i) as TCostSet;
    // Вычитаем из текущего подмножества все последующие
    for j:= i+1 to GetCount do begin
      Sj:= GetItem(j) as TCostSet;
      Si.mSet.Sub(Sj.mSet);
    end;
  end;
end;
```

```
// Вывод множества в текстовый файл.
// Вспомогательные переменные PrintLevel и OldFlag
// служит для формирования отступов
// при печати вложенных множеств

var PrintLevel : integer = 0;
    OldFlag: boolean;

procedure TSet.Print(var aFile: TextFile);
var t : TItem;       // очередной элемент
    S : string;      // отступ из пробелов
    i : integer;
    flag: boolean;
begin
  S:=''; for i:=1 to 2*PrintLevel do S:= S+#32;
  Write(aFile,S+'{');
  if PrintLevel=0 then OldFlag:= true;
  Inc(PrintLevel);
  flag:= false;
  PositionPush;
  t:= GetFirst;
  while Assigned(t) do begin
    flag:= (t is TSet) or
           (t is TCostSet) or
           (t is TGraph) or
           (t is TNode);
    if flag and OldFlag then Writeln(aFile);
    t.Print(aFile);
    t:= GetNext;
  end;
  PositionPop;
  if flag
    then Writeln(aFile,S+'} : ', mCount)
    else Writeln(aFile,' } : ', mCount);
  OldFlag:= flag;
  Dec(PrintLevel);
end;

// Проверка пересечения двух множеств

function TSet.TestIntersect(arg: TSet): boolean;
var  Temp: TSet;   // для проверки пересечения
begin
  Temp:= CreateSet;
  Temp.CopyItems(Self);
  Temp.Mul(arg);
  Result:= Temp.GetCount > 0;
  Temp.Free;
end;

function TSet.GetCount: integer;
begin
  Result:= mCount;
end;

// Доступ к элементу множества по его индексу
// index = 1..GetCount

function TSet.GetItem(index: integer): TItem;
begin
  Result:= nil;
```

```
  if (index<1) or (index>GetCount) then exit;
  PositionPush;              // сохр. позицию
  Result:= GetFirst;         // взять первый
  while index>1 do begin     // переход по цепочке
    Dec(Index);
    Result:= GetNext;
  end;
  PositionPop;               // восст. позицию
end;

// Доступ к элементу множества по содержимому

function TSet.GetObject(aItem: TItem): TItem;
begin
  PositionPush;             // сохр. позицию
  Result:= GetFirst;        // взять первый
  while Assigned(Result) and
        (Result.Compare(aItem) <> cmpEq)
    do Result:= GetNext;    // переход по цепочке
  PositionPop;              // восст. позицию
end;

////////////////////////////////////////////////////////////////////////////
// TCostSet -- множество с оценкой
////////////////////////////////////////////////////////////////////////////

constructor TCostSet.Create(aCost: integer; aSet: TSet; aDestroy : boolean);
begin
  inherited Create;
  mCost:= aCost;      // оценка
  mSet := aSet;       // ссылка на множество
  if not Assigned(mSet) then begin
    mDestroy:= true;
    mSet:= CreateSet;
  end;
  if aDestroy then mDestroy:= true;  // множество уничтожается с объектом
end;

constructor TCostSet.CreateEmpty;
begin
  inherited Create;
  mDestroy:= true;
  mSet:= CreateSet;
end;

function TCostSet.Copy: TItem;
begin
  Result:= TCostSet.Create(mCost, mSet.Copy as TSet, mDestroy);
  (Result as TCostSet).mDestroy:= true;
end;

destructor TCostSet.Destroy;
begin
  if mDestroy then mSet.Free;
  inherited;
end;


procedure TCostSet.Clr(aDestroy: boolean);
begin
  if aDestroy
    then mSet.ClrAndDestroy
```

```
    else mSet.Clear;
  mCost:=0;
end;

procedure TCostSet.Clear;
begin
  Clr(false)
end;

procedure TCostSet.ClrAndDestroy;
begin
  Clr(true)
end;

// Добавление подмножеств

procedure TCostSet.Append(arg: TCostSet);
var CS : TCostSet;
     i : integer;
begin
  if not Assigned(arg) then Exit;
  for i:= 1 to arg.mSet.GetCount do begin
    CS:= TCostSet(arg.mSet.GetItem(i));
    Insert(CS);
  end;
end;

// Вставка нового элемента (подмножества) с корректировкой цены

procedure TCostSet.Insert(arg: TCostSet);
begin
  if not Assigned(arg) then Exit;
  Inc(mCost, arg.mCost);
  mSet.Insert(arg);
end;

// Сравнение множеств с оценкой:

function TCostSet.Compare(arg: TItem): TCompare;
begin
  if Self = arg then begin
    Result:= cmpEq;
    Exit;
  end;
  Result:= cmpIncomp;  // начальное значение
  if not Assigned(arg) then Exit;
  if not (arg is TCostSet) then Exit;
  // Сравниваем цены:
  if mCost = (arg as TCostSet).mCost then begin
    // При равной цене
    // сравнением мощности множеств
    if mSet.GetCount > (arg as TCostSet).mSet.GetCount
      then Result:= cmpLess
    else if mSet.GetCount < (arg as TCostSet).mSet.GetCount
      then Result:= cmpGreate
      // Если мощности одинаковы сравниваем сами множества
      else Result:= mSet.Compare((arg as TCostSet).mSet);
  end else begin
    // Если цены не совпали,
    // сравнениваем по удельной цене (дешёвые - в начало)
    if mCost * (arg as TCostSet).mSet.GetCount <
      (arg as TCostSet).mCost * mSet.GetCount
```

```
      then  Result:= cmpLess
      else
        if mCost * (arg as TCostSet).mSet.GetCount >
            (arg as TCostSet).mCost * mSet.GetCount
          then   Result:= cmpGreate
          else  { сравнение мощностей элементов-множеств }
            if mSet.GetCount > (arg as TCostSet).mSet.GetCount
              then Result:= cmpLess
              else
                if mSet.GetCount < (arg as TCostSet).mSet.GetCount
                  then Result:= cmpGreate
  end;
end;


procedure TCostSet.Print(var aFile: TextFile);
var t: TItem;
begin
  t:= mSet.GetItem(1);
  if Assigned(t) and (t is ClassType) then begin
    Writeln(aFile, 'Cost= ', mCost);  // печатаем оценку и новую строку
  end else begin
    Write(aFile, ' Cost=', mCost:4, '  ');  // печатаем только оценку
  end;
  mSet.Print(aFile)
end;

end.
```

Приложение С
Модуль Items

```
unit Items;
{$I Common.inc}
//*********************************************************************
//                        Модуль с элементами                         *
//                                                                    *
// Содержит объекты:                                                  *
// TItemChar -- элемент-символ                                        *
// TItemStr  -- элемент-строка                                        *
// TItemNum  -- элемент-число                                         *
// TBuffer   -- универсальный буфер                                   *
//*********************************************************************

interface

uses Root;

type

  // TItemChar -- элемент-символ
  TItemChar = class (TItem)
    private
      mData: Char;
    public
      constructor Create(arg: Char);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: Char;
  end;

  // TItemStr  -- элемент-строка
  TItemStr = class(TItem)
    private
      mData: String;
    public
      constructor Create(arg: string);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: string;
  end;

  // TItemNum  -- элемент-число
  TItemNum = class (TItem)
    private
      mData: integer;
    public
      constructor Create(arg: integer);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: integer;
  end;

  // Вспомогательная структура для организации
  // односвязного списка
```

```
  PStackRec = ^TStackRec;
  TStackRec = record
    mItem : TItem;              // элемент списка
    mNext : PStackRec;          // ссылка на следующий
  end;

  // Класс для организации буферов: стеков и очередей

  TBuffer = class (TItem)
    private
      mCount: integer;          // счётчик элементов
      mHead, mQue: PStackRec; // указатели на начало и конец буфера
      mCurrent : PStackRec;   // указатель на текущий элемент при переборе
      procedure Clr(aDestroy: boolean);
    public
      destructor Destroy;  override;
      function Copy: TItem; override; // создание копии
      procedure Put(arg: TItem);      // занесение в очередь
      function Get: TItem;            // извлечение из очереди
      procedure Push(arg: TItem);     // занесение в стек
      function Pop: TItem;            // извлечение из стека
      function Top: TItem;            // Возвращает элемент на вершине стека
      procedure Reversion;            // реверс буфера
      function GetByIndex(aIndex: integer): TItem; // доступ по индексу
      function GetCount: integer;     // количество элементов в буфере
      procedure Clear;                // очистка без уничтожения объектов
      procedure ClrAndDestroy;        // очистка с уничтожением объектов
      procedure Print(var aFile: TextFile); override; // вывод в файл
      // Последовательный перебор элементов (сканирование)
      function GetFirst: TItem;
      function GetNext: TItem;
      // Поиск (проверка наличия) элемента в буфере
      function IsPresent(arg : TItem): boolean;
  end;

////////////////////////////////////////////////////////////////////////////
implementation
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// TItemChar -- элемент-символ
////////////////////////////////////////////////////////////////////////////

function TItemChar.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemChar) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
  if (arg as TItemChar).mData = mData
    then Result:=cmpEq
    else if (arg as TItemChar).mData > mData
            then Result:= cmpLess
            else Result:= cmpGreate;
end;

constructor TItemChar.Create(arg: Char);
begin
  inherited Create;
  mData:= arg;
```

```
end;

function TItemChar.Copy: TItem;
begin
  Result:= TItemChar.Create(mData);
end;


procedure TItemChar.Print(var aFile: TextFile);
begin
  Write(aFile, mData:2)
end;

function TItemChar.GetData: Char;
begin
  Result:= mData
end;

////////////////////////////////////////////////////////////////////////////
// TItemStr  -- элемент-строка
////////////////////////////////////////////////////////////////////////////

function TItemStr.Copy: TItem;
begin
  Result:= TItemStr.Create(mData);
end;

constructor TItemStr.Create(arg: string);
begin
  inherited Create;
  mData:= arg
end;

function TItemStr.GetData: string;
begin
  Result:= mData;
end;

function TItemStr.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemStr) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
  if (arg as TItemStr).mData <> mData
    then if (arg as TItemStr).mData > mData
            then Result:= cmpLess
            else Result:= cmpGreate
end;

procedure TItemStr.Print(var aFile: TextFile);
begin
  Write{ln}(aFile, ' ', mData);
end;

////////////////////////////////////////////////////////////////////////////
// TItemNum  -- элемент-число
////////////////////////////////////////////////////////////////////////////

function TItemNum.Compare(arg: TItem): TCompare;
```

```
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemNum) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
  if (arg as TItemNum).mData <> mData
    then if (arg as TItemNum).mData > mData
            then Result:= cmpLess
            else Result:= cmpGreate
end;


function TItemNum.Copy: TItem;
begin
  Result:= TItemNum.Create(mData);
end;

constructor TItemNum.Create(arg: integer);
begin
  inherited Create;
  mData:= arg
end;

function TItemNum.GetData: integer;
begin
  Result:= mData;
end;

procedure TItemNum.Print(var aFile: TextFile);
begin
  Write(aFile,' ', mData);
end;

////////////////////////////////////////////////////////////////////////////
// TBuffer -- класс для организации очередей и стеков
////////////////////////////////////////////////////////////////////////////

// Создание копии буфера из существующего

function TBuffer.Copy: TItem;
var t: TItem;
    i: integer;
begin
  Result:= TBuffer.Create;
  for i:= 1 to GetCount do begin
    t:= Get;
    Put(t);
    TBuffer(Result).Put(t);
  end;
end;

// Уничтожение буфера без уничтожения элементов

destructor TBuffer.Destroy;
begin
  Clear;
  inherited;
end;

// Очистка буфера
```

```
procedure TBuffer.Clr(aDestroy: boolean);
var i : integer;
    t : TItem;
begin
  for i:= 1 to mCount do begin
    t:= Get;
    if aDestroy then t.Free
  end;
end;

// Очистка с уничтожением элементов

procedure TBuffer.ClrAndDestroy;
begin
  Clr(true)
end;

// Очистка без уничтожения

procedure TBuffer.Clear;
begin
  Clr(false)
end;

// Получение счётчика элементов

function TBuffer.GetCount: integer;
begin
  Result:= mCount;
end;

// Последовательный перебор элементов

function TBuffer.GetFirst: TItem; // Выбор первого
begin
  Result:= nil;
  mCurrent:= mHead;
  if not Assigned(mHead) then Exit;
  Result:= mHead^.mItem;
end;

function TBuffer.GetNext: TItem; // Выбор следующего
begin
  Result:= nil;
  if Assigned(mCurrent) then mCurrent:= mCurrent^.mNext;
  if Assigned(mCurrent) then Result:= mCurrent^.mItem;
end;

// Поиск (проверка наличия) элемента в буфере

function TBuffer.IsPresent(arg: TItem): boolean;
var Item: TItem;
begin
  Result:= false;
  Item:= GetFirst;
  while Assigned(Item) do begin
    Result:= Item = arg;
    if Result then Break;
    Item:= GetNext;
  end;
end;
```

```
// Push - помещение в стек (в начало буфера)

procedure TBuffer.Push(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= mHead;
  mHead:= r;
  Inc(mCount);
  // Обновляем указатель на конец списка
  if not Assigned(mQue) then mQue:= r;
end;

// Put - помещение в очередь (в конец буфера)

procedure TBuffer.Put(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= nil;
  // Присоединяем к концу списка
  if Assigned(mQue) then mQue^.mNext:= r;
  mQue:= r;
  Inc(mCount);
  // Обновляем указатель на начало
  if not Assigned(mHead) then mHead:= r;
end;

// Извлечение из очереди и стека (из начала списка )

function TBuffer.Get: TItem;
var r: PStackRec;
begin
  Result:= nil;
  if not Assigned(mHead) then Exit;
  r:= mHead;
  Result:= mHead^.mItem;
  mHead:= mHead^.mNext;
  Dispose(r);
  Dec(mCount);
  // Обновляем указатель на хвост
  if mCount=0 then mQue:= nil;
end;

// Извлечение из стека (синоним Get)

function TBuffer.Pop: TItem;
begin
  Result:= Get;
end;

// Возвращает элемент на вершине стека без его извлечения

function TBuffer.Top: TItem;
begin
  Result:= nil;
  if not Assigned(mHead) then Exit;
  Result:= mHead^.mItem;
end;
```

```
// Доступ к очереди по индексу

function TBuffer.GetByIndex(aIndex: integer): TItem;
var  r: PStackRec;
begin
  Result:= nil;
  if (aIndex<1) or (aIndex>mCount) then Exit;
  r:= mHead;
  while Assigned(r) do begin
    Dec(aIndex);
    if aIndex=0 then Break;
    r:= r^.mNext;
  end;
  if Assigned(r) then Result:= r^.mItem;
end;

// Реверс буфера (перестановка в обратном порядке)

procedure TBuffer.Reversion;
var B : TBuffer;
    t : TItem;
begin
  B:= TBuffer.Create;
  t:= Get;
  while Assigned(t) do begin   B.Push(t);  t:= Get;  end;
  t:= B.Get;
  while Assigned(t) do begin   Put(t);   t:= B.Get;  end;
  B.Free;
end;

// Печать буфера

procedure TBuffer.Print(var aFile: TextFile);
var r: PStackRec;
begin
  Writeln(aFile, '(');
  r:= mHead;
  while Assigned(r) do begin
    r^.mItem.Print(aFile);
    r:= r^.mNext;
  end;
  Writeln(aFile, ') BufCount= ', mCount);
end;

//////////////////////////////////////////////////////////////////////////////

end.
```

Приложение D
Модуль SetList

```
unit SetList;
{$I Common.inc}
//*********************************************************************
//        Реализация множества на базе односвязного списка            *
//                                                                    *
// Содержит объекты:                                                  *
// TSetList -- Множество, реализованное списком                       *
//*********************************************************************

interface

uses Root, Items;

type

  // Вспомогательная структура для организации односвязного списка

  PListRec = ^TListRec;
  TListRec = record
    mItem : TItem;
    mNext : PListRec;
  end;

  // Множество, реализованное списком

  TSetList = class (TSet)
    protected
      mHead: PListRec; // голова списка
      procedure Clr(arg: boolean); override;
    private
      mCurrent : PListRec;  // указатель на текущий элемент
      mStack : TBuffer;   // стек для хранения позиций чтения mCurrent
      procedure Del(var p, q : PListRec);
      procedure Ins(p: PListRec; var q: PListRec; arg: TItem);
    public
      destructor Destroy;  override;
      function Exist(arg : TItem): boolean; override;
      function Insert(arg: TItem): boolean; override;
      procedure Delete(arg: TItem); override;
      function GetFirst: TItem; override;
      function GetNext: TItem; override;
      procedure PositionPush;  override;
      procedure PositionPop;   override;
      procedure Add(arg : TSet); override;
      procedure Sub(arg : TSet); override;
      procedure Mul(arg : TSet); override;
      procedure ExOr(arg : TSet); override;
  end;

//////////////////////////////////////////////////////////////////////////
implementation
//////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
// TSetList -- Множество, реализованное списком
////////////////////////////////////////////////////////////////////////////

// Извлечение первого элемента списка

function TSetList.GetFirst: TItem;
begin
  mCurrent:= mHead;
  if Assigned(mCurrent)
    then Result:= mCurrent^.mItem
    else Result:= nil;
end;

// Извлечение следующего элемента списка

function TSetList.GetNext: TItem;
begin
  if Assigned(mCurrent)
    then mCurrent:= mCurrent^.mNext;
  if Assigned(mCurrent)
    then Result:= mCurrent^.mItem
    else Result:= nil;
end;

// Сохранение в стеке текущей позиции перебора

procedure TSetList.PositionPop;
begin
  mCurrent:= PListRec(mStack.Pop);
end;

// Извлечение из стека текущей позиции перебора

procedure TSetList.PositionPush;
begin
  if not Assigned(mStack)
    then mStack:= TBuffer.Create;
  mStack.Push(TItem(mCurrent));
end;

// Очистка множества без уничтожения элементов

procedure TSetList.Clr(arg: boolean);
var p : PListRec;
begin
  while Assigned(mHead) do begin
    p:= mHead;
    mHead:= mHead^.mNext;
    if arg and Assigned(p^.mItem)
       then p^.mItem.Free;
    Dispose(p);
  end;
  mCount:= 0;
end;
```

```
// Деструктор множества

destructor TSetList.Destroy;
begin
  mStack.Free;
  inherited;
end;

// Проверка наличия элемента в множестве

function TSetList.Exist(arg: TItem): boolean;
var p : PListRec;
begin
  Result:= false;
  if not Assigned(arg) then Exit;
  p:= mHead;
  while Assigned(p) do begin
    if p.mItem = arg then begin Result:= true; Break end;
    case p^.mItem.Compare(arg) of
      cmpLess,
      cmpIncomp: p:=p^.mNext;  // меньше или несравнимо
      cmpEq    : begin Result:= true; Break end;
      cmpGreate: Break;  // больше
    end;
  end;
end;

// Вспомогательная процедура вставки

procedure TSetList.Ins(p: PListRec; var q: PListRec; arg: TItem);
var r : PListRec;
begin
  Inc(mCount);
  New(r);  r^.mItem:= arg; r^.mNext:= p;
  if Assigned(q)
    then q^.mNext:= r
    else mHead:= r;
  q:= r;
end;

// Вставка элемента в множество
// Возвращает FALSE,
// если элемент или его эквивалент уже содержится в множестве

function TSetList.Insert(arg: TItem): boolean;
var p, q : PListRec;
    cmp : TCompare;
begin
  Result:= false;
  if not Assigned(arg) then Exit;
  cmp:= cmpEq;
  p:= mHead; q:=nil;
  while Assigned(p) do begin
    cmp:= p^.mItem.Compare(arg);
    if cmp in [cmpEq, cmpGreate] then Break;
    q:= p;  p:= p^.mNext
  end;
  if not Assigned(p) or (cmp = cmpGreate) then begin
    Ins(p,q,arg);
    Result:= true;
  end;
end;
```

```
// Вспомогательная процедура удаления

procedure TSetList.Del(var p, q: PListRec);
begin
  if Assigned(q)
    then q^.mNext:= p^.mNext
    else mHead:= p^.mNext;
  Dispose(p);
  if Assigned(q)
    then p:= q^.mNext
    else p:= mHead;
  Dec(mCount);
end;

// Процедура удаления элемента из множества

procedure TSetList.Delete(arg: TItem);
var p, q : PListRec;
begin
  if not Assigned(arg) then Exit;
  p:= mHead; q:= nil;
  while Assigned(p) do begin
    case p^.mItem.Compare(arg) of
      cmpLess: begin q:= p;  p:= p^.mNext;  end;
      cmpEq:   begin Del(p,q); Break; end;
      else Break;
    end;
  end;
end;

// Сложение множеств: аргумент добавляется к данному множеству

procedure TSetList.Add(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  if not (arg is TSet) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    cmp:= cmpEq;
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      case cmp of
        cmpLess  : begin q:= p; p:= p^.mNext end;
        cmpEq    : begin t:= arg.GetNext; break end;
        else  Break;
      end;
    end;
    if not Assigned(p) or (cmp = cmpGreate) then begin
      Ins(p,q,t);
      if Assigned(t) then t:= arg.GetNext;
    end;
  end;
  arg.PositionPop;
end;
```

```
// Вычитание множеств: аргумент вычитается из данного множества

procedure TSetList.Sub(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  if not arg.InheritsFrom(TSet) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    cmp:= cmpLess;
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      if cmp in [cmpEq, cmpGreate] then Break;
      q:= p; p:= p^.mNext;
    end;
    if cmp = cmpEq then Del(p,q);
    if not Assigned(p) then Break;
    t:= arg.GetNext;
  end;
  arg.PositionPop;
end;

// Пересечение множеств: аргумент пересекается с данным множеством

procedure TSetList.Mul(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    cmp:= cmpEq;
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      if cmp <> cmpEq then Break;
      q:= p;
      p:= p^.mNext;
    end;
    if cmp = cmpLess then Del(p,q);
    if not Assigned(p) then Break;
    if cmp = cmpGreate then t:= arg.GetNext;
  end;
  while Assigned(p) do Del(p,q);
  arg.PositionPop;
end;

// Исключающее ИЛИ множеств: результат в данном множестве

procedure TSetList.ExOr(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  p:= mHead; q:= nil;
```

```
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      case cmp of
        cmpLess:
          begin  q:= p; p:= p^.mNext end;
        cmpEq:
          begin
            Del(p, q);
            t:= arg.GetNext;
            if not Assigned(t) then break;
          end;
        cmpGreate:
          begin
            Ins(p, q, t);
            t:= arg.GetNext;
            if not Assigned(t) then break;
          end;
      end;
    end;
    while Assigned(t) do begin
      Ins(p, q, t);
      t:= arg.GetNext
    end;
  end;
  arg.PositionPop;
end;

end.
```

Приложение E
Модуль Dissect

```
unit Dissect;
//*********************************************************************
//                Разбиения множества на подмножества                 *
// Функции:                                                           *
// GenAllDissections -- Генерирует все разбиения множества            *
// GenDissectRand -- Генерирует заданное количество разбиений         *
// GenSubsRand -- Генерирует заданное количество подмножеств          *
// GenSubsFromText -- Генерирует подмножества по текстовому файлу     *
//*********************************************************************

interface

uses Root, Items, SetList;

// Генерация всех разбиений множества aSet

function GenAllDissections(aSet: TSet): TSet;

// Генерирует заданное в aCnt количество разбиений
// исходного множества aSet.
// Параметры задают минимальную и максимальная мощность подмножеств,
// входящих в разбиения

function GenDissectRand(
                     aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TSet;

// Генерирует заданное в aCnt количество подмножеств
// исходного множества aSet.
// Параметры задают минимальную и максимальную (%) мощность подмножеств,
// входящих в разбиения
// Совокупность полученных подмножеств обладает двумя свойствами разбиения:
// - сумма всех подмножеств равна aSet
// -- произведение всех подмножеств пусто
// Но в отличие от разбиения некоторые подмножества совокупности
// могут пересекаться и даже совпадать.

function GenSubsRand(aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TBuffer;

// Генерация из текстового файла
// набора подмножеств TBuffer и УНИВЕРСУМА aSet
// Используются только буквы 'a'..'z'

function GenSubsFromText(const aName: String;  // имя файла
                         var aSet: TSet;       // результат-УНИВЕРСУМ
                         aCnt: integer         // количество подмножеств
                         ): TBuffer;
```

```
////////////////////////////////////////////////////////////////////////////
implementation
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Генерация всех разбиений множества ( Липский, стр 43 )
////////////////////////////////////////////////////////////////////////////

function GenAllDissections(aSet: TSet): TSet;
var Que    : TBuffer;  // буфер-очередь
    SSGet  : TSet;       // извлекаемое множество подмножеств
    SSPut : TSet;        // добавляемое множество подмножеств
    //-----------------------------------------
    // Процедура обработка всех подмножеств
    // из множества множеств SSGet

    procedure LocalHandle(aItem: TItem);
    var k: integer;
        S: TSet;     // текущее множество из множества подмножеств
    begin
      // Обработка очередного извлечённого множества подмножеств SSGet,
      // перебираем его элементы-множества S:
      for k:= 1 to SSGet.GetCount do begin
          SSPut:= SSGet.Copy as TSet;   // копия извлеч. множества
подмножеств
          S:= SSPut.GetItem(k) as TSet; // очередное множество из МП
          SSPut.Delete(S);   // удаляем из него очередное подмножество
          S.Insert(aItem);   // к очередному добавляем текущий элемент
          SSPut.Insert(S);   // полученное вставляем в множество подмножеств
          Que.Put(SSPut);    // новое множество подмножеств заносим в очередь
      end;
      // Добавление нового подмножества к SSGet
      SSPut:= SSGet.Copy as TSet;  // копия извлеч. множества подмножеств
      S:= CreateSet;      // создаём множество
      S.Insert(aItem);    // из единственнного элемента aItem
      SSPut.Insert(S);    // добавляем к множеству подмножеств
      Que.Put(SSPut);     // новое множество подмножеств заносим в очередь
    end;
    //-----------------------------------------
var   i, j  : integer;  // счётчики
      Item: TItem;      // текущий элемент исходного множества

begin
  Que:= TBuffer.Create; // создание буфера для очереди
  SSPut:= CreateSet;      // пустое множество подмножеств
  Que.Put(SSPut);         // заносим в очередь
  // Цикл по всем элементам исходного множества:
  for i:= 1 to aSet.GetCount do begin
    Item:= aSet.GetItem(i);
    // Цикл обработки очереди:
    for j:=1 to Que.GetCount do begin
      SSGet:= Que.Get as TSet;  // берём очередное множество из очереди
      LocalHandle(Item);        // создаём из него ряд других
      SSGet.ClrAndDestroy;      // очищаем от подмножеств
      SSGet.Free;               // и уничтожаем
    end;
  end;
  Result:= CreateSet; // создание пустого результата
  // Перенос из буфера в множество-результат
  while Que.GetCount>0 do Result.Insert(Que.Get);
  Que.Free;  // удаление буфера
end;
```

```
////////////////////////////////////////////////////////////////////////////
//  Генерация случайного разбиения исходного множества aSet
//  Внимание: aSet очищается!
////////////////////////////////////////////////////////////////////////////

function GenRandSet(aMin, aMax: integer; aSet: TSet): TSet;
var n, k, j: integer;
    t: TItem;
    S: TSet;
begin
  Result:= CreateSet;                // создаём множество-результат
  repeat
    n:= aMin + Random(aMax-aMin);    // случайная мощность подмножества
    S:= CreateSet;                   // создаём подмножество
    for j:= 1 to n do begin          // и заполняем его случайным выбором эл-
в
      k:= 1 + Random(aSet.GetCount); // случайный индекс в исходном множестве
      t:= aSet.GetItem(k);           // случайный элемент исходного множества
      S.Insert(t);                   // вставляем в множество
      aSet.Delete(t);                // и удаляем из исходного
      if aSet.GetCount=0 then Break; // выход из цикла, если исчерпано
    end;
    if S.GetCount>0          // если множество не пусто,
      then Result.Insert(S)  // то вставляем в результат
      else S.Free;           // а иначе удаляем
  until aSet.GetCount=0;     // пока не исчерпано исходное множество
end;

////////////////////////////////////////////////////////////////////////////
// Генерирует заданное в aCnt количество разбиений
// исходного множества aSet.
// Параметры задают минимальную и максимальную (%) мощность подмножеств,
// входящих в разбиения
////////////////////////////////////////////////////////////////////////////

function GenDissectRand(
                      aSet: TSet;  { исходное множество }
                      aCnt,  { количество генерируемых разбиений }
                      aMin,  { минимальная мощность подмножеств, %}
                      aMax   { максимальная мощность подмножеств, %}
                      : integer): TSet;
var
  nMin, nMax: Integer;
  Copy: TSet;    // рабочая копия исходного множества aSet
  SS: TSet;      // очередное множество подмножеств
  i: Integer;

begin
  // Перевод процентов в количество

  nMin:= (aSet.GetCount * aMin) div 100;   // aMin %
  if nMin<1 then nMin:=1;
  if nMin>=aSet.GetCount then nMin:=aSet.GetCount-1;

  nMax:= (aSet.GetCount * aMax) div 100;   // aMax %
  if nMax<nMin then nMax:=nMin;
  if nMax>aSet.GetCount then nMax:=aSet.GetCount;

  Result:= CreateSet;
  for i:= 1 to aCnt do begin
    Copy:= aSet.Copy as TSet;
```

```
    repeat
      SS:= GenRandSet(nMin, nMax, Copy);
      Result.Insert(SS);
    until Copy.GetCount=0;
    Copy.Free;
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Генерирует заданное в aCnt количество подмножеств
// исходного множества aSet.
// Параметры задают минимальную и максимальную (%) мощность подмножеств,
// входящих в разбиения
// В отличие от разбиения некоторые подмножества совокупности
// могут пересекаться и даже совпадать.
////////////////////////////////////////////////////////////////////////////

function GenSubsRand(aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TBuffer;
var
  nMin, nMax: Integer;
  Copy: TSet;  // рабочая копия исходного множества aSet
  SS: TSet;    // очередное разбиение
  S : TSet;    // очередное множество внутри разбиения

begin
  // Перевод процентов в количество

  nMin:= (aSet.GetCount * aMin) div 100;   // aMin %
  if nMin<1 then nMin:=1;
  if nMin>=aSet.GetCount then nMin:=aSet.GetCount-1;

  nMax:= (aSet.GetCount * aMax) div 100;   // aMax %
  if nMax<nMin then nMax:=nMin;
  if nMax>aSet.GetCount then nMax:=aSet.GetCount;

  Result:= TBuffer.Create;  // создаём буфер результата

  // Цикл создания заданного количества подмножеств

  while aCnt>0 do begin
    Copy:= aSet.Copy as TSet;   // создаём копию исходного множества
    repeat
      SS:= GenRandSet(nMin, nMax, Copy);  // создаём случайное разбиение
      // Перебор подмножеств случайного разбиения:
      S:= SS.GetFirst as TSet;   // первое подмножество из разбиения
      while Assigned(S) and (aCnt>0) do begin
        if Random(3)<>0 then begin
          // копия очередного "осколка"
          // с вероятностью 2/3 вставляется в результат
          Result.Put(S.Copy);
          Dec(aCnt);
        end;
        S:= SS.GetNext as TSet;  // следующее подмножество из разбиения
      end;
      SS.ClrAndDestroy;  // очищаем разбиение
      SS.Free;           // и удаляем его
    until (aCnt=0) or (Copy.GetCount=0);
    Copy.Free;   // удаляем копию исходного множества
```

```
  end;
end;

//////////////////////////////////////////////////////////////////////////
// Генерация из текстового файла
// набора подмножеств TBuffer и УНИВЕРСУМА aSet
// Используются только буквы 'a'..'z'
//////////////////////////////////////////////////////////////////////////

function GenSubsFromText(const aName: String;  // имя файла
                         var aSet: TSet;       // результат-УНИВЕРСУМ
                         aCnt: integer         // количество подмножеств
                         ): TBuffer;

type TChrSet = set of char;

    // - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Формирование объекта-множества символов
    // из классического множества символов

    function MakeFromSet(const aSet: TChrSet): TSet;
    var c: char;
        t: TItemChar;
    begin
      Result:= CreateSet;
      for c in aSet do begin
        t:= TItemChar.Create(c);
        Result.Insert(t)
      end;
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - -

var F: Text;            // текстовый файл
    S: string;          // строка файла
    c: char;            // очередной символ
    i: integer;         // индекс
    set1 : TChrSet;     // множество для УНИВЕРСУМа
    set2 : TChrSet;     // множество для подмножества
    subset: TSet;       // очередное подмножество

begin  { GenSubsFromText }

  Result:= TBuffer.Create;
  set1:= [];
  set2:= [];
  Assign(F, aName); Reset(F);
  while not Eof(F) do begin
    if Result.GetCount >= aCnt then break;
    Readln(F, S);
    for i:= 1 to Length(S) do begin
      // Обработка очередной строки файла:
      if Result.GetCount >= aCnt then break;
      c:= Char(Ord(S[i]) or $20);        // low case (буквы 'a'..'z')
      if c in ['a'..'z'] then begin
        set1:= set1 + [c];               // для УНИВЕРСУМА
        if c in set2 then begin          // если символ встретился повторно
           subset:= MakeFromSet(set2);   // создаём подмножество
           Result.Put(subset);           // и заносим в результат
           set2:= [c];                   // начинаем следующее подмножество
        end else begin                   // если символ встретился впервые
           set2:= set2 + [c];            // расширяем текущее подмножество
        end;
```

```
      end;
    end;
  end;
  Close(F);
  subset:= MakeFromSet(set2);  // оставшееся множество
  Result.Put(subset);          // вставляем в результат
  aSet:= MakeFromSet(set1);    // и создаём универсум
end;

end.
```

Приложение F
Модуль Assembly

```
unit Assembly;
{$I Common.inc}
//*********************************************************************
//                       Сборка множеств                              *
// Функции:                                                           *
// CollectMinDissect -- Задача о наименьшем разбиениии (ЗНР)          *
// CollectMinCover   -- Задача о наименьшем покрытии (ЗНП)            *
// CollectGradCover  -- быстрый градиентный алгоритм для ЗНП          *
// Кристофидес стр. 53                                                *
//*********************************************************************

interface

uses Root, Items, SetList;

//  Задача о наименьшем разбиениии (ЗНР)

function CollectMinDissect(aUniv: TSet;      // целевой универсум
                           aBuf: TBuffer   // буфер с подмножествами
                           ):TCostSet;

// Задача о наименьшем покрытии (ЗНП)
// Кристофидес стр. 53

function CollectMinCover(aUniv: TSet;      // целевой универсум
                         aBuf: TBuffer   // буфер с подмножествами
                         ):TCostSet;

// Градиентный алгоритм для задачи о наименьшем покрытии ЗНП

function CollectGradCover(aBase: TSet;      // целевой универсум
                          aBuf: TBuffer   // буфер с подмножествами
                          ):TCostSet;

///  Оценка трудоёмкости полного перебора множества блоков

function CalcDifficult(aSet: TSet): Extended;

//////////////////////////////////////////////////////////////////////////
implementation
uses SetUtils;

//////////////////////////////////////////////////////////////////////////
// TSetBlock
// Специальный тип множества для решения задач
// о минимальном разбиении (ЗНР) и покрытии (ЗНП)
//////////////////////////////////////////////////////////////////////////

type

  TSetBlock = class (TSetList)
    mLabel : TItem;    // Метка блока
    constructor Create(aLabel: TItem);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: Text); override;
  end;
```

```
constructor TSetBlock.Create(aLabel: TItem);
begin
  inherited Create;
  mLabel:= aLabel;
end;

function TSetBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;  // порядок элементов совпадает с порядком вставки
end;

procedure TSetBlock.Print(var aFile: Text);
begin
  mLabel.Print(aFile);
  inherited;
end;

//////////////////////////////////////////////////////////////////////////
//  Проверка множества подмножеств на предмет того,
//  можно ли составить разбиение или покрытие?
//////////////////////////////////////////////////////////////////////////

function TestCoverOrDissect(aUniv: TSet;     // универсум
                            aBuf: TBuffer  // буфер подмножеств
                           ): boolean;
var  CS : TCostSet;  // очередное подмножество с оценкой
    Sum : TSet;      // накопитель универсума
      i : integer;
begin
  Result:= false;
  Sum:= CreateSet;
  // Суммируем подмножества буфера до тех пор,
  // пока либо сумма достигнет универсума, либо исчерпается буфер
  for i:=1 to aBuf.GetCount do begin
    CS:= aBuf.GetByIndex(i) as TCostSet;
    Sum.Add(CS.mSet);
    if Sum.Compare(aUniv) in [cmpEq, cmpGreate] then begin
      Result:= True;
      Break;
    end;
  end;
  Sum.Free;
end;

//////////////////////////////////////////////////////////////////////////
///   М И Н И М А Л Ь Н О Е   Р А З Б И Е Н И Е  (DISSECTION)
//////////////////////////////////////////////////////////////////////////

// Просеивание дубликатов бОльшей или равной стоимости

function GenFilter(aBuf: TBuffer): TBuffer;
var Ci, Cj : TCostSet;
      i, j : integer;
      Comp : TCompare;
begin
  // Предварительный сброс флажков:
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    Ci.mFlag:= false;
  end;
  // Установка флажков для пропускаемых элементов
  // Отмечаем совпадающие подмножества с бОльшнй либо равной ценой
```

```
  for i:= 1 to aBuf.GetCount-1 do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    if Ci.mFlag then Continue;              // если обработан
    for j:= i+1 to aBuf.GetCount do begin
      Cj:= aBuf.GetByIndex(j) as TCostSet;
      if Cj.mFlag then Continue;            // если обработан
      Comp:= Ci.mSet.Compare(Cj.mSet);      // сравниваем подмножества
      if Comp = cmpEq then begin
        // Если сопадают, сравниваем цены:
        if Ci.mCost >= Cj.mCost
          then Ci.mFlag:= true    // i-й будет пропущен
          else Cj.mFlag:= true;   // j-й будет пропущен
      end;
    end;
  end;

  Result:= TBuffer.Create;
  // Перепись в буфер результата элементов со сброшенным флажком
  // Прокручиваем циклически исходный буфер aBuf
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.Get as TCostSet;
    aBuf.Put(Ci);
    // Если флаг не установлен, то заносим в буфер результата
    if not Ci.mFlag  then Result.Put(Ci);
  end;
end;

//////////////////////////////////////////////////////////////////////////////
// Создаёт блоки подмножеств для поиска разбиений
// aSet  - универсум
// aBuf  - совокупность разных его подмножеств (оценённых)
//////////////////////////////////////////////////////////////////////////////

function GenDissectBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
var i, j: integer;
    t: TItem;          // очередной элемент из универсума
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet;                   // создаём множество-результат
  for i:=1 to aUniv.GetCount do begin  // цикл по элементам универсума
    if aBuf.GetCount=0 then Break;      // стоп при исчерпании буфера
    // Выбираем очередной элемент универсума:
    t:= aUniv.GetItem(i);
    Block:= TSetBlock.Create(t); // создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.Get as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)        // если оно содержит текущий элемент
        then Block.Insert(S)    // то помещаем подмножество S в блок
        else aBuf.Put(S);       // а иначе возвращаем подмножество в буфер
    end;
    if Block.GetCount>0         // если блок не пуст
      then Result.Insert(Block) // помещаем его в множество
      else Block.Free;          // а иначе удаляем пустой
  end;
  aBuf.Free;  // освобождаем буфер
end;
```

```
////////////////////////////////////////////////////////////////////////////
//  Оценка трудоёмкости полного перебора множества блоков
////////////////////////////////////////////////////////////////////////////

function CalcDifficult(aSet: TSet): Extended;
var S: TSet;  // очередной блок
begin
  Result:=1;
  // Перемножаем мощности всех блоков (пустых нет)
  S:= aSet.GetFirst as TSet;
  while Assigned(S) do begin
    if S.GetCount=1
      then Result:= Result +1
      else Result:= Result * S.GetCount;
    S:= aSet.GetNext as TSet;
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Задача о наименьшем разбиениии ЗНР
// Кристофидес стр. 53
////////////////////////////////////////////////////////////////////////////

function CollectMinDissect(aUniv: TSet;      // целевой универсум
                           aBuf: TBuffer   // буфер с подмножествами
                           ): TCostSet;
var
    Buf     : TBuffer;  // Отфильтрованная копия входного буфера
    Univ    : TSet;       // Накопитель универсума
    BestCost: integer;    // Лучшая стоимость
    Blocks  : TSet;       // Вспомогательные блоки
    Res : TCostSet;       // Исходное пустое разбиение
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блока

  procedure BlockHandle(aBlock: integer;  // номер блока
                        aCost: integer;   // накопленная сумма
                        aSum: TSet;       // накопленные элементы
                        aRes: TCostSet    // накопленные подмножества
                        );
  var Block: TSetBlock; // текущий блок
      Sum : TSet;       // Накопленное множество элементов
      Res : TCostSet;   // Накопленное разбиение (множество подмножеств)
       CS : TCostSet;   // очередное оценённое подмножество из блока
  begin
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Если элемент текущего блока содержится в накопителе
    if aSum.Exist(Block.mLabel) then begin
      // то пропускаем этот блок и входим в следующий (если не последний)
      if aBlock < Blocks.GetCount then begin
        BlockHandle(aBlock+1, aCost, aSum, aRes) // вход в следующий блок
      end
    end else begin
      // перебор подмножеств текущего блока
      CS:= Block.GetFirst as TCostSet;
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        // Если сумма не превышена
        // и CS не пересекается с накопл. универсумом, то обрабатываем
        if ((aCost+CS.mCost) < BestCost) and
           not aSum.TestIntersect(CS.mSet) then begin
          // Пристраиваем к разбиению очередной блок:
```

```
          Sum:= aSum.Copy as TSet;      // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного разбиения
          Sum.Add(CS.mSet);      // накапливаем универсум
          Res.Insert(CS);        // пристраиваем к разбиению очередной блок
          // Проверяем, накоплен ли целевой универсум
          if Sum.GetCount = aUniv.GetCount then begin
            // Здесь найдено очередное разбиение.
            // Поскольку оно лучше прежнего,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;    // новая лучшая сумма
            Result.Free;                    // удаляем прежнее разбиение
            Result:= Res.Copy as TCostSet; // и создаём новое лучшее
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res)
          end; // else
          // Перед выбором следующего подмножества в блоке освобождаем:
          // возвращаем из стека предыдущие значения
          Res.Free;    // накопленное разбиение
          Sum.Free;    // накопленный универсум
        end; // if
        CS:= Block.GetNext as TCostSet;
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------
begin  { CollectMinDissect }

  Result:= TCostSet.CreateEmpty;  // Создаём пустое множество-разбиение

  // Проверяем возможность найти разбиение, иначе выход

  if not TestCoverOrDissect(aUniv, aBuf) then Exit;

  Buf:= GenFilter(aBuf);  // Создаём фильтрованную копию входного буфера

  Blocks:= GenDissectBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  // Подготовка переменных:
  BestCost:= MaxInt;           // Лучшая сумма
  Univ := CreateSet;           // Накопитель универсума
  Res:= TCostSet.CreateEmpty; // Создаём пустое множество-разбиение

  // Рекурсивная обработка блоков:
  BlockHandle(1, 0, Univ, Res);

  // Очистка:
  Res.Free;                    // Исходное пустое разбиение
  Univ.Free;                   // Освобождаем накопитель
  Blocks.ClrAndDestroy;        // Удаляем блоки
  Blocks.Free;                 // и множество блоков
end;

////////////////////////////////////////////////////////////////////////
//    М И Н И М А Л Ь Н О Е   П О К Р Ы Т И Е  (COVER)
////////////////////////////////////////////////////////////////////////

// Создаёт блоки подмножеств для поиска минимальных покрытий
// aUniv - универсум
// aBuf  - совокупность разных его подмножеств (оценённых)

function GenCoverBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
```

```
var i, j: integer;
    t: TItem;          // очередной элемент из aSet
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet; // создаём буфер результата
  // Цикл по элементам универсума:
  for i:=1 to aUniv.GetCount do begin
    t:= aUniv.GetItem(i);         // выбираем очередной элемент универсума
    Block:= TSetBlock.Create(t); // создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.GetByIndex(j) as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)       // если оно содержит текущий элемент
        then Block.Insert(S)   // то помещаем подмножество S в блок
    end;
    Result.Insert(Block) // помещаем блок в результирующее множество
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Задача о наименьшем покрытии (ЗНП)
// Кристофидес стр. 53
////////////////////////////////////////////////////////////////////////////

function CollectMinCover(aUniv: TSet;       // целевой универсум
                         aBuf: TBuffer    // буфер подмножеств с оценками
                        ): TCostSet;
var Buf: TBuffer;        // отфильтрованный входной буфер
    Univ    : TSet;        // Накопленный универсум
    BestCost: integer;     // Лучшая сумма
    Blocks  : TSet;        // Вспомогательные блоки
    Res : TCostSet;        // Исходное пустое покрытие
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блоков

  procedure BlockHandle(aBlock, aCost: integer; aSum: TSet; aRes: TCostSet);
  var Block : TSetBlock; // текущий блок
      Sum : TSet;        // Накопленный универсум
      Res : TCostSet;    // Накопленное покрытие (множество подмножеств)
       CS : TCostSet;    // очередное оценённое подмножество из блока
  begin
    // Выбираем блок из множества блоков
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Проверяем наличие метки текущего блока в накопителе универсума
    if aSum.Exist(Block.mLabel) then begin
      // Метка уже есть, пропускаем данный блок
      // и входим в следующий, если текущий не последний
      if aBlock<Blocks.GetCount then BlockHandle(aBlock+1,aCost,aSum,aRes)
    end else begin
      // Начало обработки очередного блока
      CS:= Block.GetFirst as TCostSet; // первое подмножество блока
      // Перебор подмножеств блока
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        if (aCost+CS.mCost) < BestCost then begin // сумма меньше ?
          Sum:= aSum.Copy as TSet;      // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного покрытия
          Sum.Add(CS.mSet);      // накапливаем универсум
          Res.Insert(CS);        // и покрытие
          // Проверяем, накоплен ли целевой универсум (покрытие):
          if Sum.GetCount = aUniv.GetCount then begin
```

```
            // Найдено очередное покрытие,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;    // новая лучшая сумма
            Result.Free;                    // удаляем прежнее покрытие
            Result:= Res.Copy as TCostSet; // и сохраняем копию нового
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res);
          end; // else
          // Перед выбором следующего подмножества из блока удаляем:
          Res.Free;  // накопленное покрытие
          Sum.Free;  // накопленный универсум
        end; // then
        CS:= Block.GetNext as TCostSet;  // следующий элемент блока
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------

begin  { CollectMinCover }

  Result:= TCostSet.CreateEmpty; // Создаём пустое покрытие

  // Проверяем, если покрытие недостижимо, то выход:
  if not TestCoverOrDissect(aUniv, aBuf) then Exit;

  Buf:= GenFilter(aBuf); // Создаём фильтрованную копию входного буфера
  Blocks:= GenCoverBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  Buf.Free;                             // Буфер уже не нужен, удаляем
  Res:= TCostSet.CreateEmpty; // Начальное пустое покрытие
  Univ := CreateSet;          // Накопитель элементов (универсум)
  BestCost:= MaxInt;          // Лучшая сумма
  BlockHandle(1, 0, Univ, Res);  // Рекурсивная обработка блоков
  // Очистка:
  Res.Free;              // Начальное пустое покрытие
  Univ.Free;             // удаляем накопитель элементов (универсум)
  Blocks.ClrAndDestroy; // удаляем блоки
  Blocks.Free;
end;

////////////////////////////////////////////////////////////////////////////
// Градиентный алгоритм задачи о наименьшем покрытии ЗНП
////////////////////////////////////////////////////////////////////////////

function CollectGradCover(aBase: TSet;
                          aBuf: TBuffer
                         ): TCostSet;
var    Buf : TBuffer;     // отфильтрованный входной буфер
      Univ : TSet;        // накопитель универсума
    Blocks : TSet;        // блоки
     Block : TSetBlock;   // текущий блок
    BestCS : TCostSet;    // текущее лучшее покрытие
         i : Integer;
  //-------------------------------------------------------------
  function GetBest: TCostSet;
  var      CS : TCostSet;    // очередное подмножество с оценкой
            S : TSet;        // рабочее подмножество
    BestDelta : integer;     // лучшее приращение
     BestCost : integer;     // лучшая цена
            i : integer;
  begin
```

```
    Result:= nil;
    S:= CreateSet;  // создаём рабочее множество
    BestDelta:=0;   // лучшая мощность
    BestCost:= 1;   // лучшая цена
    // перебор элементов блока:
    for i:= 1 to Block.GetCount do begin
      CS:= Block.GetItem(i) as TCostSet;  // очередное подмнож. с оценкой
      S.CopyItems(CS.mSet);               // копируем подмножество
      S.Sub(Univ);                        // оставляем приращение множества
      // Если данное приращение лучшее, то запоминаем
      if S.GetCount * BestCost > CS.mCost * BestDelta then begin
        Result:= CS;              // лучшее множество
        BestCost:= CS.mCost;      // лучшая цена
        BestDelta:= S.GetCount;   // лучшее приращение
      end;
    end;
    S.Free;  // удаляем рабочее множество
  end;
  //-------------------------------------------------------------

begin  { CollectGradCover }

  Result:= TCostSet.CreateEmpty;  // Покрытие = пустое множество

  // Если покрытие невозможно, вернуть пустой результат

  if not TestCoverOrDissect(aBase, aBuf) then Exit;

  Buf:= GenFilter(aBuf);// Отфильтрованная копия входного буфера
  Blocks:= GenCoverBlocks(aBase, Buf);  // Вспомогательные блоки
  Buf.Free;             // Буфер уже не нужен
  Univ:= CreateSet;     // Создаём накопитель универсума
  // Цикл обработки блоков:
  for i:= 1 to Blocks.GetCount do begin
    Block:= Blocks.GetItem(i) as TSetBlock;
    if Univ.Exist(Block.mLabel)  // если метка блока найдена в накопителе
      then Continue;             // то пропускаем блок
    BestCS:= GetBest;            // выбор лучшего подмножества в блоке
    if Assigned(BestCS) then begin
      // Если лучшее подмножество в блоке найдено:
      Univ.Add(BestCS.mSet);     // накапливаем универсум
      Result.Insert(BestCS);     // и покрытие
    end;
  end;
  // Очистка памяти:
  Univ.Free;              // освободить накопитель универсума
  Blocks.ClrAndDestroy;   // очистить вспомогательные блоки
  Blocks.Free;
end;

end.
```

Приложение G
Модуль Graph

```
unit Graph;
{$I Common.inc}
//*********************************************************************
//                           Базовый ГРАФ                            *
//                                                                   *
// Содержит объекты:                                                 *
//                                                                   *
// TNode -- базовая вершина (узел)                                   *
// TNodeInt -- вершина, помеченная числом                            *
// TCondensNode -- вершина конденсата                                *
// TLink -- базовый линк                                             *
// TFarLink -- дальний указатель линк                                *
// TGraph -- базовый граф                                            *
// THugeGraph -- огромный граф с вершинами-числами                   *
// TCostNodes -- множество вершин с оценкой                          *
// TSortedNum -- самосортирующиеся числа                             *
// TGamma -- множество вершин                                        *
// TAreal -- объект для поиска центров                               *
// TTreeLink -- объект для связывания поддеревьев                    *
// TPair -- пара вершин для поиска паросочетаний                     *
// TPairsBlock -- блок пар для поиска паросочетаний                  *
// THamNode -- вершина вспомогательного графа для задачи Гамильтона  *
// THamLink -- дуга вспомогательного графа для задачи Гамильтона     *
// THamGraph -- вспомогательный граф для задачи Гамильтона           *
// TClosed -- множество закрытых линков задачи Гамильтона            *
//*********************************************************************

interface

uses Root, Items;

type

  TReal = Single;  // данные действительного типа

  // Варианты карты дальних указателей при построении центров и медиан:

  TCenter = ( Undefined,  // не определена
              InCenter,   // внутренний центр и медиана
              OutCenter,  // внешний центр и медиана
              InOutCenter // внешне-внутренний центр и медиана
            );

  // Варианты формирования паросочетаний:

  TPairs = (pMaxN,  // максимальной мощности
            pMaxW,  // максимального веса
            pMinW   // минимального веса
           );

  // Предварительные объявления:

  TNode = class;      // вершина
  TLink = class;      // связь
  TGraph = class;     // граф
  TFarLink = class;   // дальний указатель
```

```
// Базовая вершина

TNode = class (TItem)       // Вершина (узел) графа
  private
    mLnkOut : TSet;         // Множество исходящих связей
    mLnkIn  : TSet;         // Множество входящих связей
    mFarLinks: TSet;        // Множество дальних связей для карты
    function GetNear(aFar: TNode): TNode;
    function GetFarLink(aFar: TNode): TFarLink;
    procedure SortFarLinks;
    procedure PrintFarLinks(var aFile: TextFile);
    procedure ExpoFarLinks;
    procedure ExpoData;
  protected
    mOwner  : TGraph;    // Ссылка на содержащий узел граф
    mColor  : integer;   // Цвет
    mDist   : integer;   // Расстояние (Дейкстра)
                         // Текущий штраф (цепь Гамильтона)
                         // Текущий профит от корня дерева (паросочетания)
    mPred   : TNode;     // Предшествующий узел (Дейкстра)
    mLink   : TLink;     // Дуга в цепочке (потоки, паросочетания)
    mRoot   : TItem;     // Корень (остовные деревья, цепь Гамильтона)
    mFlow   : integer;   // Допустимый поток (потоки)
                         // Приращение штрафа (цепь Гамильтона)
    mLimit  : integer;   // Предел посещаемости (задача почтальона)
    mPower  : integer;   // Текущая степень вершины (цепь Гамильтона)
    //--------------------
    // Для паросочетаний
    mBlossom: TNode;     // Ссылка на цветок (TBlossom)
    mPair   : TLink;     // Линк для связи с парной вершиной или цветком
    mRight  : TNode;     // Следующая вершина при обходе по часовой
    mLeft   : TNode;     // Следующая вершина при обходе против часовой
    mRLnk   : TLink;     // Линк при обходе цветка по часовой
    mOut    : boolean;   // Положение в альтернирующей цепи:
                         // false -- внутр, true -- внешняя
    mOwnDest : boolean;  // Простая парная вершина в цветке:
                         // false -- источник, true -- приёмник линка
    mProfit : integer;   // Профит (выгода) внутри цветка
                         // от перемещении базы в данную вершину
    //--------------------

    function  MakeLink(aDest: TNode; aVal : integer): TLink;
    function  GetLink(aDest: TNode): TLink;
    procedure RemoveLink(aDest: TNode);
    procedure AddTreeLinks(aMaxTree: boolean; aLinks: TSet);
    procedure SetLinksLimit(aLimit: integer);
    function  IsLeft: boolean;
    function  IsRight: boolean;
    procedure ResetNode;
    // Для задачи о паросочетаниях
    function GetBlossom: TNode; // Возвращает крайний охватывающий цветок
    function GetPair: TNode;    // Возвращает парную простую вершину
  public
    mValue : integer;       // Вес вершины
    constructor Create(aVal: integer; aOwner: TGraph);
    destructor Destroy;  override;
    function  Compare(arg: TItem): TCompare; override;
    function  GetName: string;  virtual;
    procedure PrintLinks(var aFile: TextFile);
    procedure Print(var aFile: TextFile); override;
    function  GenStrongArea: TSet;
    // вспомогательные функции для работы с исходящими связями
```

```
    function  OutGetCnt: integer;
    function  OutLinkFirst: TLink;
    function  OutLinkNext: TLink;
    procedure OutPosPush;
    procedure OutPosPop;
    function  OutGetLink(aIndex: integer): TLink;
    procedure OutGammaAdd(aRes : TSet);
    procedure OutGammaGet(aRes : TSet);
    function  CalcCost: integer;
    function  GenGammaOut: TSet;
    // вспомогательные функции для работы с входящими связями
    function  InGetCnt: integer;
    function  InLinkFirst: TLink;
    function  InLinkNext: TLink;
    procedure InPosPush;
    procedure InPosPop;
    function  InGetLink(aIndex: integer): TLink;
    procedure InGammaAdd(aRes : TSet);
    function  GenGammaIn: TSet;
end;

// Вершина, помеченная числом

TNodeInt = class (TNode)
  public
    mNumber  : integer;  // хранимое число
    constructor Create(aNumber, aVal: integer; aOwner: TGraph);
    function Compare(arg: TItem): TCompare; override;
    function Copy: TItem; override;
    function GetName: string;  override;
    function GetLinkByNumber(aNumber : integer): TLink;
end;

// Линк -- ребро или дуга

TLink = class (TItem)
  protected
    mLoadLinks : boolean;  // Признак нагруженных связей
    mLimit  : integer;     // Предел посещаемости (циклы)
  public
    mValue: integer;  // вес (стоимость, расстояние) дуги
    mOwner: TNode;    // источник дуги (владелец)
    mDest : TNode;    // приёмник дуги
    mColor: integer;  // цвет дуги (паросочетания)
    // Для решения потоковых задач:
    mLow   : integer; // допустимые минимальный поток  mLow <= mHigh
    mHigh  : integer; // допустимые максимальный поток
    mFlow  : integer; // текущий поток
    mTemp  : integer; // для временного хранения mHigh
    mDirect: integer; // направление дуги (+1 -- прямая, -1 -- обратная)
    constructor Create(aOwner, aDest: TNode; aVal: integer;
                       aLoadLinks: boolean);
    function  Compare(arg: TItem): TCompare; override;
    function  Copy: TItem; override;
    function  GetReverse: TLink; // поиск встречного линка
    procedure Print(var aFile: TextFile); override;
    procedure Print2(var aFile: TextFile);
end;

// Базовый граф

TGraph = class (TItem)
```

```
    private
      mInitMap : boolean;    // Признак инициализации карты дальних связей
      // Для поиска р-центров:
      mMapDirect : TCenter; // направление для карты дальних связей
    protected
      mNodes : TSet;         // Множество вершин
    public
      mName : String;        // Произвольное имя
      mDirect : boolean;     // Признак орграфа
      mLoadNodes : boolean; // Признак нагруженных вершин
      mLoadLinks : boolean; // Признак нагруженных связей
    protected
    // Вспомогательные методы
      procedure PosPush;
      procedure PosPop;
      procedure ResetNodes;                      // очистка вершин
      function  GetNode(aNode: TNode): TNode;   // поиск вершин по значению
полей
      procedure SetLink(aSource, aDest : TNode; aVal : integer);
      procedure SetLinksLimit(aLimit: integer);
      procedure Compress; // Замена длин дуг кратчайшими расстояниями
    public
    // Базовые методы
      constructor Create(const aName: string; aDir, aLNodes, aLLinks :
boolean);
      destructor Destroy;  override;
      function  InsertNode(aNode: TNode): boolean;
      procedure RemoveNode(aNode: TNode);  // удаление вершины из графа
      function  Compare(arg: TItem): TCompare; override;
      procedure Print(var aFile: TextFile); override;
      procedure Save(const aName: String); virtual; abstract;
      function  Nodes: integer;
      procedure ExpoFarLinks;
      procedure ExpoLinksData;
    // Перебор вершин
      function  NodeFirst: TNode;
      function  NodeNext: TNode;
    // Копирование и порождение подграфов
      function  CopyGr(aRevers: boolean): TGraph;
      function  Copy: TItem; override;
      function  CopyByNodes(aNodes: TSet): TGraph;
      function  CopyByLinks(aLinks: TSet): TGraph;
    // Достижимость
      function  TestLink(arg1, arg2 : TNode): boolean;
      function  CalcSteps(arg1, arg2 : TNode): integer;
      function  GenPath(aSource, aDest : TNode): TBuffer;
      function  GenDijkstra(aSource, aDest : TNode): TBuffer;
      function  GenQuickPath(aSource, aDest : TNode): TBuffer;
      function  GetQuickPathStr(aSource, aDest : TNode): String;
    private
      procedure InitMap_Floyd; // построение карты маршрутов
    public
    // Проверка связности (граф и орграф)
      function IsLinked: boolean;
    // Области связности в неориентированом графе
      function  CalcAreas: integer;  // подсчёт областей
      function  GenAreas: TSet;      // генерация областей связности
    // Достижимость и связность в ориентированом графе
      function  GenStrongAreas: TSet;
      function  GenCondens: TGraph;
    // Независимые множества, клики, доминирующие множества
      function  GenUndepend: TSet;
```

```
      function  GenClique: TSet;
      function  GenDominating: TCostSet;
    // Раскраска
      function  GenPaints(aQuick: boolean): TSet;
    // Центры
      function  GenCenter(var aLambda: integer; aDirect: TCenter): TSet;
      function  GenPCentLambda(aLambda: integer; aDirect: TCenter): TCostSet;
      function  GenPCenters(aPoly: integer;
                            var aLambda: integer;
                            aCenter: TCenter): TCostSet;
    private
      procedure InitMap(aDirect: TCenter);
      procedure DoneMap;
      function  GenIntersections(aGams: TSet): TBuffer;
      function  GenLimitGams(aLambda: integer): TSet;
      function  GenLimitGamma(aNode: TNode; aLambda: integer): TSet;
    // P-Медианы
    public
      // Поиск полным перебором
      function GenPMedian_A (aPoly: integer;    // кратность (P)
                             aDirect: TCenter;  // внешняя или внутр. медиана
                             var aCost: integer // стоимость (результат)
                            ): TSet;
      // Поиск частичным перебором
      function GenPMedian (aPoly: integer;      // кратность (P)
                           aDirect: TCenter;    // внешняя или внутр. медиана
                           var aCost: integer   // стоимость (результат)
                          ): TSet;
    private
      function MedianCost: integer;      // Вычисление стоимости медианы
      procedure MedianCopy(aRes: TSet); // Копирование медианы в множество
    // Покрывающие деревья
    public
      function GenCoverTree(aMaxTree: boolean; var aCost: integer): TSet;
      function GenCoverDir(var aRoot: TNode; var aCost: integer): TSet;
      procedure ExpoAllCovers;
    // Потоки
    public
      function CalcMaxFlow0(aSource, aDest : TNode): integer;
      function CalcMinFlow (aSource, aDest : TNode): integer;
      function CalcMaxFlow (aSource, aDest : TNode): integer;
      function CalcMinCostFlow (aSource, aDest : TNode;
                                aFlow: integer ): integer;
    // Паросочетания
    public
      function GenPairs_Dissect(aMode: TPairs): TCostSet;
      function GenPairs_Edmonds(aMode: TPairs): TCostSet;
      function MarkMinPairsDicoty(var aPairs: integer): integer;
      function GenMinPairsDicoty: TCostSet;
      function GenMinPairsDicotyFlow: TCostSet;
    // Задача почтальона
    protected
      function GenPostPathUndir(var aCost: integer): TBuffer;
      function GenPostPathDir(var aCost: integer): TBuffer;
    public
      function GenPostPath(var aCost: integer): TBuffer;
    public
    // Цепь Гамильтона (насыщенный граф)
      function GenHamPath(aStart, aFin: TNode; var aCost: integer): TBuffer;
      function GenHamPathDir(aStart, aFin: TNode; var aCost: integer):
TBuffer;
      function GenHamPathStrip(aStart, aFin, aStrip: TNode;
```

```
                                var aCost: integer): TBuffer;
      function GenHamPath_Full(aStart, aFin: TNode; var aCost: integer):
TBuffer;
      function GenHamPath_Greed(aStart, aFin: TNode;
                                var aCost: integer;
                                aGreed: boolean): TBuffer;

    // Путь и цикл Гамильтона (насыщенный граф)
      function GenHamilton_Full(var aCost: integer): TBuffer;
      function GenHamilton_Greed(var aCost: integer;
                                 aGreed: boolean): TBuffer;
      function GenHamilton(var aCost: integer): TBuffer;
      function GenHamilton_Quick(var aCost: integer): TBuffer;
    // Задачи коммивояжёра на ненасыщенном графе
      procedure MakeFull(aValue: integer);
      // Разомкнутая задача коммивояжёра (путь)
      function GenTravelPath(aStart, aFin: TNode; var aCost: integer):
TBuffer;
      // Замкнутая задача коммивояжёра (цикл)
      function GenTravelCycle(var aCost: integer): TBuffer;
  end;

  // Огромный Граф, вершины которого помечены числами

  THugeGraph = class (TGraph)
    private
      function  GetLinkByNumber(aSource, aDest : integer): TLink;
      procedure MakeLink(aSource, aDest, aVal : integer);
    public
      constructor GenRandom(aDir: boolean;          // направленность
                            aLoadNodes, aLoadLinks, // нагруженность
                            aNodes,                 // количество вершин
                            aLinks                  // плотность связей,%
                            : integer);
      constructor GenFull(aDir: boolean;            // направленность
                          aLoadNodes, aLoadLinks,   // нагруженность
                          aNodes: integer);         // количество вершин
      constructor GenDicoty(aLoadLinks,             // нагруженность дуг
                            aPairs: integer);       // количество пар вершин
      function  GetNode(aNumber: integer): TNodeInt;
  end;

  // Дальний указатель для карты дальних связей

  TFarLink = class(TItem)    // Дальний указатель
    mNodeFar : TNode;        // целевая вершина
    mNodeNear: TNode;        // ближайшая вершина на пути к целевой
    mDist    : integer;      // расстояние от текущей к целевой
    mStep    : integer;      // цикл, на котором обрабатывается указатель
    constructor Create(aNear, aFar: TNode; aDist: integer);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile);  override;
  end;

  // Узел для создания конденсата (сильной компоненты)

  TCondensNode = class (TNode)
    public
      mName  : string;// имя = сумма входящих в компоненту имён вершин
      mNodes : TSet;  // собственные узлы
      mOut   : TSet;  // ссылки за пределы сильной компоненты (узлы)
      constructor Create(aNodes: TSet; aOwner: TGraph);
```

```
      destructor Destroy;  override;
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;

  // Элемент для представления доминирующих множеств

  TCostNodes = class (TCostSet)
    mNode: TNode;  // вершина-элемент доминирующего множества
    constructor Create(aCost: integer; aSet: TSet; aNode: TNode);
    procedure   Print(var aFile: TextFile); override;
  end;

  // Функция для создания огромного графа

function GenHugeGraph (aDir: boolean;           // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                 // количество вершин
                       aLinks                  // плотность связей,%
                       : integer): TGraph;

////////////////////////////////////////////////////////////////////////
implementation

uses Assembly, SetList, SetUtils, SysUtils, DateUtils, GrChars;
////////////////////////////////////////////////////////////////////////

const              // "цвета" вершин:
    CWhite = 0;    // белая (не тронута)
    CGray  = 1;    // серая
    CBlack = 2;    // чёрная
    CRed   = 3;    // красная

    CInfinity = MaxInt;  // бесконечность

////////////////////////////////////////////////////////////////////////
// Выбор минимального из двух чисел
////////////////////////////////////////////////////////////////////////

function Minimum(a1, a2 : integer): integer;
begin
  if a1 < a2 then Result:= a1 else Result:= a2;
end;

////////////////////////////////////////////////////////////////////////
// Вспомогательная функция для преобразования
// множества вершин в строку символов
////////////////////////////////////////////////////////////////////////

function NodesToStr(aSet: TSet): string;
var N: TNode;
begin
  Result:= '';
  N:= aSet.GetFirst as TNode;
  while Assigned(N) do begin
    if Length(Result)<>0 then Result:= Result+',';
    Result:= Result + N.GetName;
    N:= aSet.GetNext as TNode;
  end;
end;

////////////////////////////////////////////////////////////////////////
```

```
// TNode -- вершина (узел) графа
//////////////////////////////////////////////////////////////////////////

constructor TNode.Create(aVal: integer; aOwner: TGraph);
begin
  inherited Create;              // унаследованный конструктор
  mOwner:= aOwner;               // владелец узла (граф)
  mBlossom:= Self;               // Указатель на цветок
  mLnkOut:= CreateSet;           // пустое множество исходящих связей
  if Assigned(mOwner) then begin
    if mOwner.mLoadNodes         // если вершины нагружены
      then mValue:= aVal;        // то запомнить вес (по умолчанию = 0)
    if mOwner.mDirect            // если орграф
      then mLnkIn := CreateSet; // то создать пустое множество входящих
связей
  end
end;

destructor TNode.Destroy;
begin
  // Очистка и уничтожение:
  mLnkOut.ClrAndDestroy;    // исходящие линки
  mLnkOut.Free;             // множество исходящих линков
  mLnkIn.Free;              // множество входящих линков
  // Дальние указатели:
  if Assigned(mFarLinks) then begin
    mFarLinks.ClrAndDestroy;
    mFarLinks.Free;
  end;
  inherited;
end;

// Сравнение вершин:

function TNode.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  if GetName < (arg as TNode).GetName
    then Result:= cmpLess
  else if GetName > (arg as TNode).GetName
    then Result:= cmpGreate
end;

// Имени у абстрактного узла не существует,
// поэтому формируем случайно

function TNode.GetName: string;
var N: integer;
begin
  N:= integer(Self) mod 99;
  Result:='x'+Char(Ord('0')+N div 10)+Char(Ord('0')+N mod 10);
end;

// Установка степеней для всех рёбер (исходящих дуг)

procedure TNode.SetLinksLimit(aLimit: integer);
var L : TLink;
begin
  L:= OutLinkFirst;
  while Assigned(L) do begin
    L.mLimit:= aLimit;
```

```
    L:= OutLinkNext;
  end;
end;

// Поиск дальней связи при формировании пути
// по распределённой карте дальних связей

function TNode.GetFarLink(aFar: TNode): TFarLink;
var  FL : TFarLink; // элемент дальней связи
begin
  Result:= nil;
  if not Assigned(mFarLinks) then Exit;
  mFarLinks.PositionPush;
  FL:= mFarLinks.GetFirst as TFarLink;   // первый элемент
  while Assigned(FL) do begin            // перебор
    if FL.mNodeFar = aFar then begin     // если нашли
      Result:= FL;                       // вернуть результат
      Break;
    end;
    FL:= mFarLinks.GetNext as TFarLink;  // следующий элемент
  end;
  mFarLinks.PositionPop;
end;

// Поиск ближайшего узла при построении маршрута по карте

function TNode.GetNear(aFar: TNode): TNode;
var  FL : TFarLink; // элемент дальней связи
begin
  Result:= nil;
  if not Assigned(mFarLinks) then Exit;
  mFarLinks.PositionPush;
  FL:= mFarLinks.GetFirst as TFarLink;   // первый элемент
  while Assigned(FL) do begin            // перебор
    if FL.mNodeFar = aFar then begin     // если нашли
      Result:= FL.mNodeNear;             // вернуть результат
      Break;
    end;
    FL:= mFarLinks.GetNext as TFarLink;  // следующий элемент
  end;
  mFarLinks.PositionPop;
end;

// Сортировка дальних указателей по неубыванию дистанции

procedure TNode.SortFarLinks;
var NewList : TSet;
begin
  if not Assigned(mFarLinks) then Exit;
  NewList:= CreateSet;
  NewList.CopyItems(mFarLinks);
  mFarLinks.Free;
  mFarLinks:= NewList;
end;

// Сброс узла перед поиском кратчайших путей, потоков и др.

procedure TNode.ResetNode;
begin
  mColor   := 0;       // цвет
  mPred    := nil;     // ссылка на предшествующий узел
  mLink    := nil;     // связывающая дуга в цепочке потока
```

```
  mDist    := MaxInt;  // расстояние от исходного узла
  mFlow    := MaxInt;  // допустимый поток
  mRoot    := nil;     // Корень дерева
  mBlossom:= Self;     // Ссылка на цветок
  mPair    := nil;     // Парная вершина паросочетания
  mRight   := nil;     // Следующая вершина в цикле цветка
  mLeft    := nil;     // Предыдущая вершина в цикле цветка
  mOut     := false;   // false -- внутр, true -- внешняя вершина в цепи
end;

// Установка связи между данной вершиной и заданной параметром
// aDest - вершина-приёмник
// aVal  - "вес" связи

function TNode.MakeLink(aDest: TNode; aVal: integer): TLink;
begin
  Result:= nil;
  if not Assigned(aDest) then Exit;
  Result:= GetLink(aDest);             // уже существует?
  if Assigned(Result) then Exit;       // да, выход
  // нет, создаём новую
  Result:= TLink.Create(Self, aDest, aVal, mOwner.mLoadLinks);
  // и вставляем исходящую связь
  if not mLnkOut.Insert(Result) then begin
    Result.Free;
    Result:= nil;
    Exit;
  end;
  if mOwner.mDirect // если орграф
    // то вставляем в приёмнике как входящую
    then aDest.mLnkIn.Insert(Result);
end;

// Удаление связи между вершинами

procedure TNode.RemoveLink(aDest: TNode);
var  L: TLink;
begin
  if not Assigned(aDest) then Exit;
  OutPosPush;
  L:= OutLinkFirst;
  while Assigned(L) and (L.mDest <> aDest) do  L:= OutLinkNext;
  OutPosPop;
  if not Assigned(L) then Exit;
  mLnkOut.Delete(L);
  if mOwner.mDirect  // если орграф
    // то удаляем в приёмнике как входящую
    then aDest.mLnkIn.Delete(L);
  L.Free;
end;

// Поиск линка к указанной вершине

function TNode.GetLink(aDest: TNode): TLink;
begin
  OutPosPush;
  Result:= OutLinkFirst;
  while Assigned(Result) and (Result.mDest <> aDest)
    do  Result:= OutLinkNext;
  OutPosPop;
end;
```

```
// Создание множества сильной компоненты
// или связной области для графа
// Кристофидес, стр. 33

function TNode.GenStrongArea: TSet;
var Temp: TSet;
begin
  Result:= GenGammaOut;                // множество достижимых вершин
  if not mOwner.mDirect then Exit;     // выход, если это не орграф
  Temp:= GenGammaIn;                   // множество, из которых достижима
  Result.Mul(Temp);                    // пересечение множеств
  Temp.Free;                           // освобождаем
end;


// Вспомогательные методы для работы с исходящими связями

// Извлечение первого исходящего линка

function TNode.OutLinkFirst: TLink;
begin
  Result:= mLnkOut.GetFirst as TLink;
end;

// Извлечение следующего исходящего линка

function TNode.OutLinkNext: TLink;
begin
  Result:= mLnkOut.GetNext as TLink;
end;

// Сохранение позиции перебора

procedure TNode.OutPosPush;
begin
  mLnkOut.PositionPush;
end;

// Восстановление позиции перебора

procedure TNode.OutPosPop;
begin
  mLnkOut.PositionPop;
end;

// Получение исходящей связи по индексу

function TNode.OutGetLink(aIndex: integer): TLink;
begin
  Result:= TLink(mLnkOut.GetItem(aIndex));
end;

// Кол-во исходящих связей

function TNode.OutGetCnt: integer;
begin
  Result:= 0;
  if Assigned(mLnkOut) then Result:= mLnkOut.GetCount;
end;

// Накопление соседних узлов в множестве aSet
```

```
procedure TNode.OutGammaAdd(aRes : TSet);
var L: TLink;
begin
  L:= OutLinkFirst;             // первая исходящая связь
  while Assigned(L) do begin    // пока существуют
    aRes.Insert(L.mDest);       // вставить целевую вершину в результат
    L:= OutLinkNext;            // следующая исходящая связь
  end;
end;

// Получение исходящей гаммы

procedure TNode.OutGammaGet(aRes: TSet);
begin
  aRes.Clear;
  OutGammaAdd(aRes);
end;

// Создание множества вершин, достижимых из данной

function TNode.GenGammaOut: TSet;
var Node: TNode;
    Temp: TSet;
begin
  Result:= CreateSet;
  Result.Insert(Self);         // начианем расширение гаммы с текущей
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);            // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;     // начало перебора вершин
    while Assigned(Node) do begin
      Node.OutGammaAdd(Result);        // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;

// Подсчёт стоимости узла для поиска минимальных покрытий

function TNode.CalcCost: integer;
var L: TLink;
begin
  Result:=1;
  if mOwner.mLoadNodes then Result:= mValue;
  if not mOwner.mLoadLinks then Exit;
  // добавляем суммарную стоимость исходящих дуг (рёбер)
  L:= OutLinkFirst;
  while Assigned(L) do begin
    Inc(Result, L.mValue);
    L:= OutLinkNext;
  end;
end;

// Вспомогательные методы для работы с входящими связями

// Извлечение первого входящего линка

function TNode.InLinkFirst: TLink;
begin
  if mOwner.mDirect
    then Result:= mLnkIn.GetFirst as TLink
```

```
    else Result:= OutLinkFirst
end;

// Извлечение следующего входящего линка

function TNode.InLinkNext: TLink;
begin
  if mOwner.mDirect
    then Result:= mLnkIn.GetNext as TLink
    else Result:= OutLinkNext
end;

// Сохранение позиции перебора

procedure TNode.InPosPush;
begin
  if mOwner.mDirect
    then mLnkIn.PositionPush
    else OutPosPush
end;

// Восстановление позиции перебора

procedure TNode.InPosPop;
begin
  if mOwner.mDirect
    then mLnkIn.PositionPop
    else OutPosPop
end;

// Получение входящей связи по индексу

function TNode.InGetLink(aIndex: integer): TLink;
begin
  if mOwner.mDirect
    then Result:= TLink(mLnkIn.GetItem(aIndex))
    else Result:= OutGetLink(aIndex);
end;

// Накопление соседних узлов в множестве aSet

procedure TNode.InGammaAdd(aRes: TSet);
var L: TLink;
begin
  L:= InLinkFirst;
  while Assigned(L) do begin
    aRes.Insert(L.mOwner);
    L:= InLinkNext;
  end;
end;

// Количество входящих связей

function TNode.InGetCnt: integer;
begin
  Result:= 0;
  if mOwner.mDirect
    then begin
      if Assigned(mLnkIn) then Result:= mLnkIn.GetCount
    end
    else Result:= OutGetCnt
end;
```

```
// Возвращает TRUE, если вершина относится к левой доле
// направленного двудольного графа

function TNode.IsLeft: boolean;
begin
  Result:= mLnkOut.GetCount <> 0;
end;

// Возвращает TRUE, если вершина относится к правой доле
// направленного двудольного графа

function TNode.IsRight: boolean;
begin
  Result:= mLnkIn.GetCount <> 0;
end;

// Вывод в файл (на экран) исходящих связей вершины

procedure TNode.PrintLinks(var aFile: TextFile);
begin
  if Assigned(mLnkOut) then mLnkOut.Print(aFile);
end;

// Вывод в файл (на экран) вершины

procedure TNode.Print(var aFile: TextFile);
begin
  Write(aFile, GetName+' ');
  if Assigned(mOwner)
    then if mOwner.mLoadNodes then begin
      if mOut
        then Write(aFile,'=', mValue:4 )
        else Write(aFile,'-', mValue:4 );
    end;
  Write(aFile, ' -> ');
  PrintLinks(aFile);
end;

// Вывод в файл (на экран) списка дальних связей

procedure TNode.PrintFarLinks(var aFile: TextFile);
var FL: TFarLink;
begin
  Print(aFile);
  with mFarLinks do begin
    PositionPush;
    FL:= GetFirst as TFarLink;
    while Assigned(FL) do begin
      FL.Print(aFile);
      FL:= GetNext as TFarLink;
    end;
    PositionPop;
  end;
end;

// Отображение дополнительных данных (для отладки)

procedure TNode.ExpoData;
var L: TLink;
    Flag: boolean;
begin
```

```
  OutPosPush;
  Flag:= false;
  // Исходящие связи:
  L:= OutLinkFirst;
  while Assigned(L) do begin
    with L do begin
      Write(mOwner.GetName+'->'+mDest.GetName, '  C=', mValue:2);
      // Данные о потоке:
      Writeln('  L=', mLow:2,  '  H=', mHigh:2,
              '  F=', mFlow:2, '  T=', mTemp:2);
    end;
    Flag:= true;
    L:= OutLinkNext;
  end;
  if Flag then Writeln;
  OutPosPop;
end;

// Отображение дальних указателей (для отладки)

procedure TNode.ExpoFarLinks;
begin
  PrintFarLinks(Output);
end;

// Создание множества вершин, из которых достижима данная

function TNode.GenGammaIn: TSet;
var Node: TNode;
    Temp: TSet;
begin
  if not mOwner.mDirect then begin
    // Для неорграфа входящая гамма совпадает с исходящей
    Result:= GenGammaOut;
    Exit;
  end;
  Result:= CreateSet;
  Result.Insert(Self);        // начианем расширение гаммы с текущей
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);           // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;    // начало перебора вершин
    while Assigned(Node) do begin
      Node.InGammaAdd(Result);        // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;

// #########  Для поиска паросочетаний  #########

// Возвращает крайний (наружный) цветок вершины

function TNode.GetBlossom: TNode;
begin
  // Для цветка вызывается рекурсивно
  if mBlossom <> Self
    then Result:= mBlossom.GetBlossom
    else Result:= Self;
end;
```

```
// Получение ссылки на простую парную вершину,
// если она существует (паросочетания)
// Используются поля:
// mPair    : TLink;      -- Парный линк
// mOwnDest  : boolean; -- Положение данной вершины в паре:
//                       false -- внутр, true -- внешняя

function TNode.GetPair: TNode;
begin
  Result:= nil;
  if not Assigned(mPair) then Exit;
  if mOwnDest
    then Result:= mPair.mOwner  // источник линка
    else Result:= mPair.mDest   // приёмник линка
end;

////////////////////////////////////////////////////////////////////////////
// TNodeInt -- Вершина, помеченная числом
////////////////////////////////////////////////////////////////////////////

constructor TNodeInt.Create(aNumber, aVal: integer; aOwner: TGraph);
begin
  inherited Create(aVal, aOwner);
  mNumber:= aNumber;
end;


function TNodeInt.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  Result:= cmpLess;
  if (arg is TNodeInt) and (mNumber > (arg as TNodeInt).mNumber)
    then Result:= cmpGreate
end;


function TNodeInt.Copy: TItem;
begin
  Result:= TNodeInt.Create(mNumber, mValue, mOwner);
end;

// Получение линка на соседнюю вершину по номеру

function TNodeInt.GetLinkByNumber(aNumber: integer): TLink;
begin
  OutPosPush;
  Result:= OutLinkFirst;
  while Assigned(Result) do begin
    if (Result.mDest as TNodeInt).mNumber = aNumber then Break;
    Result:= OutLinkNext;
  end;
  OutPosPop;
end;

function TNodeInt.GetName: string;
begin
  Result:= IntToStr(mNumber);
  while Length(Result) < 3 do Insert('0', Result, 1);
end;

////////////////////////////////////////////////////////////////////////////
```

```
// TLink - элемент связи между вершинами
////////////////////////////////////////////////////////////////////////////

constructor TLink.Create(aOwner, aDest: TNode; aVal: integer;
                         aLoadLinks: boolean);
begin
  inherited Create;          // унаследованный конструктор
  mOwner:= aOwner;           // узел-источник связи и её владелец
  mDest:= aDest;             // узел-приёмник связи
  mLoadLinks:= aLoadLinks;   // признак нагруженной связи
  if aLoadLinks              // если нагружена
    then mValue:= aVal       // то запоминаем "вес"
    else mValue:= 1;         // а иначе по умолчанию = 1
end;

// Поиск встречного линка

function TLink.GetReverse: TLink;
begin
  Result:= mDest.GetLink(mOwner);
end;

function TLink.Copy: TItem;
begin
  Result:= TLink.Create(nil, nil, mValue, mLoadLinks);
end;


function TLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  // Сравниваем источники дуг
  Result:= mOwner.Compare((arg as TLink).mOwner);
  if Result in [cmpLess, cmpGreate] then Exit;
  // Сравниваем приёмники дуг
  Result:= mDest.Compare((arg as TLink).mDest);
end;

// Печать только приёмника и длины

procedure TLink.Print(var aFile: TextFile);
begin
  Write(aFile, ' '+mDest.GetName);
  if mLoadLinks then Write(aFile, '=', mValue:2);
end;

// Печать источника, приёмника и длины

procedure TLink.Print2(var aFile: TextFile);
var N1, N2 : TNode;
begin
  N1:=mOwner;  N2:=mDest;
  if not mOwner.mOwner.mDirect and
      (mOwner.Compare(mDest) = cmpGreate) then begin
    N1:=mDest;   N2:=mOwner;
  end;
  Write(aFile, ' ' + N1.GetName + '-' + N2.GetName);
  if mLoadLinks then Write(aFile, '=', mValue:3, '':1);
end;

////////////////////////////////////////////////////////////////////////////
```

```
// TGraph  -- базовый класс для графов
//////////////////////////////////////////////////////////////////////////

constructor TGraph.Create(const aName: string;
                          aDir, aLNodes, aLLinks: boolean);
begin
  inherited Create;          // вызов унаследованного конструктора
  mName:= aName;             // имя графа
  mDirect := aDir;           // признак орграфа
  mLoadNodes := aLNodes;     // признак нагруженных вершин
  mLoadLinks := aLLinks;     // признак нагруженных связей
  mNodes:= CreateSet;        // пустое множество вершин
  mMapDirect:= Undefined;    // направление для карты дальних связей
end;


destructor TGraph.Destroy;
begin
  DoneMap;                // ликвидация карты дальних указателей
  mNodes.ClrAndDestroy;   // ликвидация всех вершин
  mNodes.Free;            // ликвидация множества
  inherited;
end;

// Вспомогательные методы:

// Возвращает количество вершин

function TGraph.Nodes: integer;
begin
  Result:= mNodes.GetCount;
end;

// Выбор первой вершины

function TGraph.NodeFirst: TNode;
begin
  Result:= mNodes.GetFirst as TNode;
end;

// Выбор следующей вершины

function TGraph.NodeNext: TNode;
begin
  Result:= mNodes.GetNext as TNode;
end;

// Сохранение позиции перебора

procedure TGraph.PosPush;
begin
  mNodes.PositionPush;
end;

// Восстановление позиции перебора

procedure TGraph.PosPop;
begin
  mNodes.PositionPop;
end;

// Сброс вершин перед построением кратчайших путей
```

```
procedure TGraph.ResetNodes;
var N : TNode;
begin
  PosPush;
  N:= NodeFirst;
  while Assigned(N) do begin
    N.ResetNode;
    N:= NodeNext;
  end;
  PosPop;
end;

// Поиск копии узла в графе (nil - если не существует)

function TGraph.GetNode(aNode: TNode): TNode;
begin
  Result:= mNodes.GetObject(aNode) as TNode;
end;

// Сравнение графов

function TGraph.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result <> cmpEq then Result:= cmpGreate;
end;

// Компрессия дуг: замена длин дуг кратчайшими расстояниями
// Используется для поиска гамильтонова цикла

procedure TGraph.Compress;
var  N: TNode;
     L: TLink;
    FL: TFarLink;
begin
  // Строим карту с дальними указателями:
  InitMap_Floyd;
  // Заменяем длины дуг кратчайшими путями:
  N:= NodeFirst;
  while Assigned(N) do begin
    // Перебор дальних указателей из вершины N:
    FL:= N.mFarLinks.GetFirst as TFarLink;
    while Assigned(FL) do begin
      if (FL.mNodeFar <> N) then begin
        // Ищем линк из данной вершины N --> mNodeFar
        L:= N.GetLink(FL.mNodeFar);
        // Ели он существует, меняем длину на кратчайшее расстояние
        if Assigned(L) and (L.mValue > FL.mDist)
            // заменяем кратчайшим расстоянием
            then L.mValue:= FL.mDist;
      end;
      FL:= N.mFarLinks.GetNext as TFarLink;
    end;
    N:= NodeNext;
  end;
end;

// Вывод графа в файл

procedure TGraph.Print(var aFile: TextFile);
begin
```

```
  Writeln(aFile, mName);
  mNodes.Print(aFile);
end;

// Вставка вершины, возвращает FALSE при попытке вставить дубликат

function TGraph.InsertNode(aNode: TNode): boolean;
begin
  Result:= mNodes.Insert(aNode)
end;

// Удаление вершины с разрывом соответствующих связей
// Вершина удаляется, но не уничтожается!

procedure TGraph.RemoveNode(aNode: TNode);
var LS, LD: TLink;
    N: TNode;
begin
  if not mNodes.Exist(aNode) then Exit;
  // Две версии: для орграфа и неорграфа
  if mDirect then begin // ОРИЕНТИРОВАННЫЙ ГРАФ
    // Дуга образована одним линком, вставленным в два множества
    // Обработка исходящих линков удаляемой вершины:
    LS:= aNode.OutLinkFirst;
    while Assigned(LS) do begin
      N:= LS.mDest;                // концевая вершина
      N.mLnkIn.Delete(LS);         // удаляем из множества входящих линков
      LS.Free;
      LS:= aNode.OutLinkNext;
    end;
    // Обработка входящих линков удаляемой вершины:
    LS:= aNode.InLinkFirst;
    while Assigned(LS) do begin
      N:= LS.mOwner;               // концевая вершина
      N.mLnkOut.Delete(LS);        // удаляем из множества исходящих линков
      LS.Free;                     // уничтожаем неактуальный линк
      LS:= aNode.InLinkNext;
    end;
  end else begin // НЕОРИЕНТИРОВАННЫЙ ГРАФ
    // Ребро образовано двумя линками, вставленными в множества mLnkOut
    // Обработка исходящих линков удаляемой вершины:
    LS:= aNode.OutLinkFirst;
    while Assigned(LS) do begin
      N:= LS.mDest;                // концевая вершина
      // В концевой вершине уничтожаем встречный линк на удаляемую вершину:
      LD:= N.OutLinkFirst;
      while Assigned(LD) and (LD.mDest <> aNode)
        do LD:= N.OutLinkNext;
      N.mLnkOut.Delete(LD);        // удаляем встречный линк
      LD.Free;                     // и уничтожаем его
      LS.Free;                     // уничтожаем неактуальный прямой линк
      LS:= aNode.OutLinkNext;
    end;
  end;  // if
  // Очищаем множества связей
  with aNode do begin
    mLnkOut.Clear;
    if Assigned(mLnkIn) then mLnkIn.Clear;
  end;
  mNodes.Delete(aNode);     // удаляем вершину из графа
end;
```

```
// Установка связи дугой или ребром
// aSource - вершина-источник
// aDest   - вершина-приёмник
// aVal    - "вес" дуги или ребра

procedure TGraph.SetLink(aSource, aDest: TNode; aVal: integer);
begin
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
  aSource.MakeLink(aDest, aVal);          // уст. прямой связи
  if not mDirect                          // если не орграф,
    then aDest.MakeLink(aSource, aVal);   // то уст. обратной связи
end;

// Установка степеней для всех исходящих дуг

procedure TGraph.SetLinksLimit(aLimit: integer);
var N: TNode;
begin
  N:= NodeFirst;
  while Assigned(N) do begin
    N.SetLinksLimit(aLimit);
    N:= NodeNext;
  end;
end;

//#######################################################################
//                                                                      #
//                Копирование графа и порождение подграфов              #
//                                                                      #
//#######################################################################

// Создание копии данного графа с возможностью реверса дуг

function TGraph.CopyGr(aRevers: boolean): TGraph;
var  N1, N2 : TNode;
         ND : TNode;
          L : TLink;
begin
  Result:= TGraph.Create(mName, mDirect, mLoadNodes, mLoadLinks);
  // Копирование вершин
  N1:= NodeFirst;
  while Assigned(N1) do begin
    N2:= TNode(N1.Copy);
    Result.InsertNode(N2);
    N1:= NodeNext;
  end;
  // Копирование связей
  N1:= NodeFirst;    // первый узел исходного
  while Assigned(N1) do begin
    N2:= Result.GetNode(N1);      // узел копии
    L:= N1.OutLinkFirst;          // связь в исходном
    while Assigned(L) do begin
      ND:= Result.GetNode(L.mDest);  // приёмник в копии
      // связать узлы в зависимочти от признака реверса
      if aRevers
        then Result.SetLink(ND, N2, L.mValue)
        else Result.SetLink(N2, ND, L.mValue);
       L:= N1.OutLinkNext;  // следующая связь
    end;
    N1:= NodeNext;    // следующий узел исходного
  end;
end;
```

```
// Создание копии данного графа без реверса дуг

function TGraph.Copy: TItem;
begin
  Result:= CopyGr(false);
end;

// Порождение подграфа подмножеством его вершин

function TGraph.CopyByNodes(aNodes: TSet): TGraph;
var  N1, N2 : TNode;     // исходный узел и его копия
         ND : TNode;     // приёмник в копии
          L : TLink;
begin
  Result:= TGraph.Create('CopyByNodes', mDirect, mLoadNodes, mLoadLinks);
  // Копирование вершин:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin  // если существует в множестве аргумента
      N2:= N1.Copy as TNode;         // то создаём копию
      Result.InsertNode(N2);         // и вставляем в граф
    end;
    N1:= NodeNext;
  end;
  // Копирование связей:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin  // если существует в множестве аргумента
      N2:= Result.GetNode(N1);       // узел копии
      L:= N1.OutLinkFirst;           // связь в исходном
      while Assigned(L) do begin     // перебор связей исходного
        ND:= Result.GetNode(L.mDest);      // приёмник в копии
        Result.SetLink(N2, ND, L.mValue); // связать узлы
        L:= N1.OutLinkNext;          // следующая связь
      end;
    end;
    N1:= NodeNext;    // следующий узел исходного
  end;
end;

// Порождение подграфа подмножеством его дуг (рёбер)

function TGraph.CopyByLinks(aLinks: TSet): TGraph;
var  NS, ND : TNode;
          L : TLink;
begin
  Result:= TGraph.Create('CopyByLinks', mDirect, mLoadNodes, mLoadLinks);

  // Порождение и связывание вершин, инцидентных заданным дугам

  L:= TLink(aLinks.GetFirst);       // Первая связь из множества
  while Assigned(L) do begin
    NS:= Result.GetNode(L.mOwner); // Взять источник связи в копии
    if not Assigned(NS)            // если нет
      then begin
        NS:= TNode(L.mOwner.Copy); // то создать
        Result.InsertNode(NS);     // и втавить
      end;
    ND:= Result.GetNode(L.mDest);  // Взять приёмник связи в копии
    if not Assigned(ND)            // если нет
      then begin
```

```
        ND:= TNode(L.mDest.Copy);  // то создать
        Result.InsertNode(ND);     // и втавить
      end;
    Result.SetLink(NS, ND, L.mValue); // связать узлы
    L:= TLink(aLinks.GetNext); // Следующая связь из множества
  end;
end;


//#######################################################################
//                                                                      #
//          Достижимость, минимальные расстояния и пути                 #
//                                                                      #
//#######################################################################

// Возвращает TRUE при наличии пути из вершины arg1 в вершину arg2
// (граф и орграф)

function TGraph.TestLink(arg1, arg2: TNode): boolean;
var SN1 : TSet;     // предыдущее множество вершин
    SN2 : TSet;     // следующее  множество вершин (накопитель)
   node : TNode;    // очередная вершина из SN1
    old : integer; // мощность предыдущей гаммы
      i : integer;
begin
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    Result:= SN1.Exist(arg2);   // конечная вершина в гамме?
    if Result then Break;       // да, выход из цикла
    old:= SN2.GetCount;         // запоминаем мощность
    // расширяем гамму SN2 := Gamma(SN1)
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если не расширилась
    SN1.CopyItems(SN2);   // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
end;

// Возвращает количество шагов из вершины arg1 в вершину arg2
// без учёта веса рёбер или дуг
// (граф и орграф)

function TGraph.CalcSteps(arg1, arg2: TNode): integer;
var SN1 : TSet;     // предыдущее множество вершин
    SN2 : TSet;     // следующее  множество вершин (накопитель)
   node : TNode;    // очередная вершина из SN1
     ok : boolean;  // признак достижения целевой вершины
    old : integer;  // предыдущая мощность накопленной гаммы
      i : integer;
begin
  Result:= 0;
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    ok:= SN1.Exist(arg2);  // признак достижения целевой вершины
    if ok then Break;
    old:= SN2.GetCount;    // запоминаем мощность до расширения
    // Накопление гаммы:  SN2 := Gamma (SN1)
```

```
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если гамма не расширилась
    Inc(Result);
    SN1.CopyItems(SN2);   // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
  if not ok then Result:= -1;   // признак недостижимости
end;

// Генерация кратчайшего пути между двумя вершинами
// без учёта веса дуг и узлов (граф и орграф)
// aSource - исходная вершина
// aDest - целевая вершина

function TGraph.GenPath(aSource, aDest: TNode): TBuffer;

var  Que : TBuffer;  // очередь вершин
    Node : TNode;      // текущая вершина
    Link : TLink;      // исходящая связь
      OK : boolean;    // признак достижения целевой вершины
begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)
  ResetNodes;              // сброс узлов в исходное состояние
  aSource.mColor:= CGray;  // исходная = серая
  Que.Put(aSource);        // исходную в очередь
  // повторяем, пока очередь не пуста:
  repeat
    OK:= false;                    // признак достижения конечной вершины
    Node:= TNode(Que.Get);         // извлечь очередную вершину
    Link:= Node.OutLinkFirst;      // первая исходящая связь
    while Assigned(Link) do begin // перебор исходящих связей
      with Link do if mDest.mColor = CWhite then begin
        // если смежная не тронута (белая)
        mDest.mPred:= Node;        // указатель на предыдущую
        OK:= mDest = aDest;        // целевая?
        if OK then Break;          // выход из цикла while
        mDest.mColor:= CGray;      // иначе красим серым
        Que.Put(mDest);            // и ставим в очередь
      end;
      Link:= Node.OutLinkNext; // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;      // обработанную вершину красим чёрным
    // выход, если достигнута целевая либо исчерпана очередь
  until OK or (Que.GetCount = 0);

  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая
    Que.Clear;                        // очистить буфер
    Que.Put(aDest);                   // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Put(Node);                  // вершину --> в результат
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
```

```
    Que.Free;        // если целевая не достигнута, ликвидируем буфер
  end;
end;

// Генерация кратчайшего пути между двумя вершинами
// по алгоритму Дейкстры с учётом весов рёбер и вершин
// (граф и орграф)

function TGraph.GenDijkstra(aSource, aDest: TNode): TBuffer;

var  Que : TBuffer;  // очередь вершин
    Node : TNode;      // текущая вершина
    Link : TLink;      // исходящая связь

begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)

  ResetNodes;               // сброс узлов в исходное состояние
  aSource.mColor:= CGray;   // исходная = серая
  aSource.mDist:= 0;        // расстояние к исходной = 0
  Que.Put(aSource);         // исходную в очередь

  repeat                    // повторяем, пока очередь не пуста
    Node:= TNode(Que.Get); // извлечь очередную вершину
    Link:= Node.OutLinkFirst; // первая исходящая связь
    while Assigned(Link) do with Link do begin  // обраб. текущего линка
      if mDest.mDist > Node.mDist + Ord(mLoadNodes)*Node.mValue + mValue
        then begin
        // поскольку расстояние в смежной больше нового,
        // обновляем расстояние и обратную ссылку
        mDest.mDist:= Node.mDist + Ord(mLoadNodes)*Node.mValue + mValue;
        mDest.mPred:= Node;
        // Если смежная не стоит в очереди, то красим белым,
        // чтобы поставить её в очередь (возможно, что повторно)
        if mDest.mColor <> CGray then mDest.mColor:= CWhite;
      end;
      if mDest.mColor = CWhite then begin
        // если смежная не стояла в очереди
        mDest.mColor:= CGray;    // то красим серым
        Que.Put(mDest);          // и ставим в очередь
      end;
      Link:= Node.OutLinkNext;  // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;    // признак, что вершина обработана
  until Que.GetCount = 0;    // выход, если исчерпана очередь

  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая
    Que.Clear;                        // очистить буфер
    Que.Put(aDest);                   // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Put(Node);                  // вершину --> в результат
      if Node = aSource then Break;   // выход, если достигли исходную
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
    Que.Free;    // если целевая не достигнута, ликвидируем буфер
```

```
  end;
end;

// Генерация распределённой по вершинам
// карты маршрутизации по алгоритму Флойда-Уоршелла
// для быстрого поиска кратчайших путей
// (граф и орграф)
// Майника, "Алгоритмы оптимизации на сетях и графах", стр. 53

// TFarLink - объект для карты маршрутов

constructor TFarLink.Create(aNear, aFar: TNode; aDist: integer);
begin
  inherited Create;
  mNodeNear:= aNear;  // соседняя вершина
  mNodeFar:= aFar;    // целевая вершина
  mDist:= aDist;      // расстояние к целевой вершине
end;

// Упорядочено в порядке неубывания дистанции

function TFarLink.Compare(arg: TItem): TCompare;
begin
  if mDist > (arg as TFarLink).mDist
    then Result:= cmpGreate
  else if mDist < (arg as TFarLink).mDist
    then Result:= cmpLess
  else Result:= mNodeFar.Compare((arg as TFarLink).mNodeFar)
end;


procedure TFarLink.Print(var aFile: TextFile);
begin
  Write(aFile,'Far  = ');          // целевая вершина
  if Assigned(mNodeFar)
    then mNodeFar.Print(aFile)
    else Writeln('NIL');
  Write(aFile,'Near = ');          // соседняя вершина
  if Assigned(mNodeNear)
    then mNodeNear.Print(aFile)
    else Writeln('NIL');           // расстояние к целевой вершине
  Writeln(aFile,'Dist = ', mDist);
  Writeln(aFile,'- - - - - - -');
end;

// Метод генерации карты маршрутизации по алгоритму Флойда-Уоршелла

procedure TGraph.InitMap_Floyd;

  //- - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей узла
  procedure InitFarLinks(aNode: TNode);
  var     Node : TNode;     // текущая вершина
          Link : TLink;     // ближний указатель
            FL : TFarLink;  // дальний указатель
  begin
    PosPush;
    Node:= NodeFirst;
    while Assigned(Node) do begin   // цикл создания дальних связей
      if Node = aNode then begin
        FL:= TFarLink.Create(Node, Node, 0);   // связь на себя
      end else with aNode do begin
```

```
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        Link:= OutLinkFirst;            // первая исходящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mDest = Node then begin
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue; // + Ord(mLoadNodes)*Node.mValue;
            FL.mStep:= 1;      // указатель будет обработан на первом шаге
            Break;
          end;
          Link:= OutLinkNext;     // следующая исходящая связь
        end; // while
      end; // else
      aNode.mFarLinks.Insert(FL);        // вставить в множество
      Node:= NodeNext;
    end;
    PosPop;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
         FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin  // Handle
    Result:= false;
    // указатель из вершины aNode
    FL1:= aNode.mFarLinks.GetFirst as TFarLink;
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)             // если определён
             and (mDist<>0)                  // и не сам на себя
             and (mNodeFar<>aNode)           // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear)  // то проверяем расстояние
              then Result:= true;            // признак того, что изменён
          FL2:= GetNext as TFarLink;         // следующая дальняя связь
        end; // while
        PositionPop;
      end;
```

```
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -

var  Node : TNode;     // текущая вершина
     Step : integer;   // этап обработки (номер цикла)
     Flag : boolean;   // признак продолжения обработки

begin { InitMap_Floyd }

  if mInitMap then Exit;  // Защита от повторной инициализации
  mInitMap:= true;        // Признак, что карта создана

  // Предварительная очистка карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin        // перебор вершин
    with Node do begin
      if Assigned(mFarLinks)           // если карта существует
        then mFarLinks.ClrAndDestroy   // очищаем
        else mFarLinks:= CreateSet;    // иначе создаём пустую
    end;
    InitFarLinks(Node);   // инициализация дальних указателей
    Node:= NodeNext;
  end;

  // Обработка вершин
  for Step:= 1 to mNodes.GetCount-2  do begin
    Flag:= false;
    Node:= NodeFirst;
    while Assigned(Node) do begin     // перебор вершин
      if Handle(Node, Step)           // если обновлялись дальние указатели
        then Flag:= true;             // то отметить
      Node:= NodeNext;
    end;
    if not Flag then Break;  // если указатели не обновлялись, то выйти
  end;
end;

// Поиск кратчайшего маршрута по карте маршрутизации
// с формированием буфера

function TGraph.GenQuickPath(aSource, aDest: TNode): TBuffer;
var Next: TNode;

begin
  Result:= nil;
  InitMap_Floyd; // Если карта ещё не построена, то строим её

  // Проверяем достижимость целевой вершины:
  Next:= aSource.GetNear(aDest);
  if not Assigned(Next) then Exit;

  Result:= TBuffer.Create;   // созд. буфер результата
  Result.Put(aSource);       // заносим начальную вершину
  // Следуем по маршруту:
  while Next <> aDest do begin
    Result.Put(Next);        // заносим промежуточную вершину
    Next:= Next.GetNear(aDest);
  end;
  Result.Put(Next);   // заносим конечную вершину
end;
```

```
// Поиск кратчайшего маршрута по карте маршрутизации
// с формированием строки

function TGraph.GetQuickPathStr(aSource, aDest: TNode): String;
var Next : TNode;
      FL : TFarLink;
       S : String;
begin
  Result:= aSource.GetName;
  InitMap_Floyd;// Если карта ещё не построена, то строим её

  // Проверяем достижимость целевой вершины:
  Next:= aSource.GetNear(aDest);
  if not Assigned(Next) then Exit;   // если не достижима
  // Следуем по маршруту:
  repeat
    Result:= Result + ' -> ' + Next.GetName;
    Next:= Next.GetNear(aDest);
  until Next = aDest;
  FL:= aSource.GetFarLink(aDest); // дальний указатель на конечную
  Str(FL.mDist, S);               // растояние до конечной
  Result:= Result + ' -> ' + Next.GetName + ' : ' + S;
end;

//#######################################################################
//                                                                      #
//                   Области связности                                  #
//               Кристофидес, Глава 2 стр. 29                           #
//                                                                      #
//#######################################################################

// TCondensNode --  конденсат
// Конструктор принимает сильную компонету графа
// (множество сильно связанных вершин)

constructor TCondensNode.Create(aNodes: TSet; aOwner: TGraph);
var Name : string;
     Val : integer;
      N  : TNode;
begin
  Name:='';  Val:= 0;
  // Цикл формирования имени и цены (веса)
  N:= TNode(aNodes.GetFirst);
  while Assigned(N) do begin
    Name:= Name+ N.GetName;       // объединение имён
    Val:= Val + N.mValue;         // и весов вершин
    N:= TNode(aNodes.GetNext);
  end;
  inherited Create(Val, aOwner);
  mName:= Name;
  // Копируем сильную компоненту
  mNodes:= TSet(aNodes.Copy);
  // Формируем mOut - ссылки за пределы сильной компоненты (узлы)
  mOut:= CreateSet;
  N:= TNode(mNodes.GetFirst);
  while Assigned(N) do begin
    N.OutGammaAdd(mOut);
    N:= TNode(mNodes.GetNext);
  end;
  mOut.Sub(mNodes);  // за исключением внутренних узлов СК
end;
```

```
destructor TCondensNode.Destroy;
begin
  mNodes.Free;
  mOut.Free;
  inherited;
end;

function TCondensNode.GetName: string;
begin
  Result:= mName;
end;

function TCondensNode.Copy: TItem;
begin
  Result:= nil;
end;

// Проверка связности (граф и орграф)

function TGraph.IsLinked: boolean;
var Node: TNode;
    S : TSet;
    NIn, NOut : integer;
begin
  Result:= false;
  // Берём произвольную вершину:
  Node:= NodeFirst;
  if not Assigned(Node) then Exit;
  // Подсчитываем вершины, из которых достижима вершина Node:
  S:= Node.GenGammaIn;
  NIn:= S.GetCount;
  S.Free;
  // Подсчитываем вершины, которые достижимы из вершины Node:
  S:= Node.GenGammaOut;
  NOut:= S.GetCount;
  S.Free;
  // Граф связан или сильно связан, если оба множества
  // покрывают все вершины графа:
  Result:= (mNodes.GetCount = NIn) and (mNodes.GetCount = NOut);
end;

// Подсчёт количества (сильно) связанных областей

function TGraph.CalcAreas: integer;
var Copy: TSet;     // копия множества всех вершин графа
    Node: TNode;    // текущая вершина
    Strong: TSet;   // область сильной связи текущей вершины
begin
  Result:= 0;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Inc(Result);                      // счётчик областей
    Node:= Copy.GetFirst as TNode;    // первая вершина из копии
    Strong:= Node.GenStrongArea;      // область её связности
    Copy.Sub(Strong);                 // вычитаем из копии
    Strong.Free                       // освобождаем область
  until Copy.GetCount = 0;
  Copy.Free;
end;
```

```
// Генерация множества (сильно) связанных областей (подграфов)

function TGraph.GenAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    Node: TNode;    // текущая вершина
    Strong: TSet;   // область сильной связи текущей вершины
    Gr: TGraph;     // граф, порождённый на множестве вершин Strong
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Node:= Copy.GetFirst as TNode;  // первая вершина из копии
    Strong:= Node.GenStrongArea;    // область её связности
    Gr:= CopyByNodes(Strong);       // создание порождённого подграфа
    Result.Insert(Gr);              // вставка порождённого подграфа
    Copy.Sub(Strong);               // вычитаем из копии
    Strong.Free                     // освобождаем область
  until Copy.GetCount = 0;          // пока копия не пуста
  Copy.Free;
end;

// Формирование множества-конденсата
// (подмножеств сильных компонент)
// Кристофидес, стр. 33

function TGraph.GenStrongAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    N : TNode;
    Strong: TSet;   // сильная компонента
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Пока копия не пуста:
  while Copy.GetCount <> 0 do begin
    N:= Copy.GetFirst as TNode;       // первая вершина в копии
    Strong:= N.GenStrongArea;         // находим сильную компоненту
    Result.Insert(Strong);            // и вставляем в результат
    Copy.Sub(Strong);                 // удаляем её из копии
  end;
  Copy.Free;  // освобождаем копию
end;

// Создание графа-конденсата
// Кристофидес, стр. 33

function TGraph.GenCondens: TGraph;
var Strongs : TSet;          // множество сильных компонент
          S : TSet;          // вспомогательное множество
         CN : TCondensNode;  // текущий узел-конденсат
     Target : TCondensNode;  // целевой узел-конденсат
begin
  Result:= TGraph.Create('Condens', true, false, false);  // пустой граф
  Strongs:= GenStrongAreas;                 // формируем множество-конденсат
  S:= Strongs.GetFirst as TSet;             // подмнож. вершин в узле
конденсата
  while Assigned(S) do begin                // перебор множеств конденсата
    CN:= TCondensNode.Create(S, Result);    // создаём вершину конденсата
    Result.InsertNode(CN);                  // и вставляем в граф-конденсат
```

```
    S:= Strongs.GetNext as TSet;          // подмнож. вершин в узле
конденсата
  end;
  // формирование связей между вершинами конденсата:
  S:= CreateSet;  // вспомогат. множество вершин исходного графа
  CN:= Result.NodeFirst as TCondensNode;        // перебор вершин конденсата
  while Assigned(CN) do begin
    if CN.mOut.GetCount <> 0 then begin         // если степень исхода <> 0
      Result.PosPush;                           // сохр. позицию перебора
      Target:= Result.NodeFirst as TCondensNode;// вершина конденсата
      while Assigned(Target) do begin           // внутр. цикл перебора
        if CN <> Target then begin              // если не текущая вершина
          S.CopyItems(CN.mOut);                 // исходящие связи текущей
          S.Mul(Target.mNodes);                 // пересекаем с целевыми
          if S.GetCount<>0                      // если есть связи в исх.
графе
            then Result.SetLink(CN, Target, 1); // то уст. связь в конденсате
        end;
        Target:= Result.NodeNext as TCondensNode; // след. узел конденсата
      end;
      Result.PosPop;                           // восст. позицию перебора
    end;
    CN:= Result.NodeNext as TCondensNode;      // след. узел конденсата
  end;
  S.Free;
  Strongs.ClrAndDestroy;  // очищаем множество-конденсат
  Strongs.Free;
end;

//#######################################################################
//                                                                      #
//     Формирование максимальных подмножеств независимых вершин         #
//                    Кристофидес-2.3,  стр. 46                         #
//              Алгоритм_Брона_—_Кербоша (Bron-Kerbosh)                 #
//                    Неориентированый граф                             #
//            Результат - множество независимых подмножеств             #
//                                                                      #
//#######################################################################

function TGraph.GenUndepend: TSet;
var
  Res: TSet;    // очередное максимальное независимое множество
  Gamma: TSet;  // для доступа к соседним вершинам
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возвращает TRUE, если среди обработанных вершин (aTested)
  // есть хоть одна, не соседствующая ни с одним кандидатом (aCand)

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);   // первый из обработанных
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи обработанного
      Gamma.Mul(aCand);        // Gamma = пересечение соседей с кандидатами
      if Gamma.GetCount=0 then begin // если пересечение пусто,
        Result:= true;       // значит не соседствует ни с одним кандидатом
        Break;
      end;
      Node:= TNode(aTested.GetNext);  // следующий из обработанных
    end;
  end;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура расширения независимого множества
  // aCand, aTested - множества с предыдущего уровня

  procedure Extend(aCand, aTested : TSet);
  var
    newCand : TSet;  // новые вершины-кандидаты
    newTested: TSet; // новые обработанные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и
    // нет независимых от них обработанных вершин
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst);     // взять первого кандидата
      Res.Insert(Node);                 // вставить в результат
      Node.OutGammaGet(Gamma);          // Gamma= его соседи
      // Создать копии множеств кандидатов и проверенных
      newCand:= TSet(aCand.Copy);
      newTested:= TSet(aTested.Copy);

      newCand.Delete(Node);   // удалить текущий узел из кандидатов
      newCand.Sub(Gamma);     // и удалить его соседей
      newTested.Sub(Gamma);   // удалить соседей из копии проверенных

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
             // то найдено очередное макс. независимое множество
             Result.Insert(Res.Copy)
        else if newCand.GetCount<>0 then
               // если остались кандидаты, то рекурсивный вызов
               // с новыми наборами кандидатов и проверенных
               Extend(newCand, newTested);

      // Возвращение
      newTested.Free;       // удалить копию проверенных
      newCand.Free;         // удалить копию кандидатов
      Res.Delete(Node);     // удалить текущий из результата
      aCand.Delete(Node);   // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Cand : TSet;   // вершины-кандидаты
  Tested: TSet;  // проверенные вершины

begin  { TGraph.GenUndepend}

  // Инициализация
  Result:= CreateSet;       // создать пустое множество-результат
  Res:= CreateSet;          // очередное макс. независимое множество
  Cand := CreateSet;        // кандидаты
  Cand.CopyItems(mNodes);   // Cand = все вершины графа
  Tested:= CreateSet;       // проверенные вершины
  Gamma:= CreateSet;        // соседние вершины

  Extend(Cand, Tested); // рекурсив. проц. построения независ. множеств
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
```

```
  Res.Free;
end;

//#######################################################################
//                                                                      #
//            Формирование множества максимальных  клик                 #
//                   Кристофидес-2.3,  стр. 46                          #
//               Алгоритм_Брона_—_Кербоша ( Bron-Kerbos                 #
//                      Неориентированый граф                           #
//                                                                      #
//#######################################################################

function TGraph.GenClique: TSet;
var
  Res: TSet;      // очередная клика
  Gamma : TSet;   // соседние вершины
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возвращает TRUE, если среди проверенных вершин в aTested
  // есть хоть одна, примыкающая ко всем кандидатами в aCand

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи Node
      Gamma.Mul(aCand);        // Gamma = соседи * кандидаты
      if Gamma.GetCount = aCand.GetCount then begin
        // если примыкает ко всем кандидатам
        Result:= true;
        Break;
      end;
      Node:= TNode(aTested.GetNext);
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура расширения клики

  procedure Extend(aCand, aTested : TSet);
  var
    newCand : TSet;  // вершины-кандидаты
    newTested: TSet; // проверенные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и в проверенных aTested нет таких,
    // что примыкают ко всем кандидатами в aCand
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst); // очередной кандидат
      Res.Insert(Node);             // вставляем в клику
      Node.OutGammaGet(Gamma);      // Gamma:= соседи Node

      newCand:= TSet(aCand.Copy);   // копия кандидатов
      newCand.Delete(Node);         // удаляем текущую вершину
      newCand.Mul(Gamma);           // и оставляем только соседние

      newTested:= TSet(aTested.Copy); // копия проверенных
      newTested.Mul(Gamma);           // оставляем только соседние

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
```

```
            // то найдена очередная клика
            Result.Insert(Res.Copy)

        else if newCand.GetCount<>0 then
            // если остались кандидаты, то рекурсивный вызов
            // с новыми наборами кандидатов и проверенных
            Extend(newCand, newTested);

      // Возвращение
      newTested.Free;        // удалить копию проверенных
      newCand.Free;          // удалить копию кандидатов
      Res.Delete(Node);      // удалить текущий из результата
      aCand.Delete(Node);    // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Cand : TSet;    // вершины-кандидаты
  Tested: TSet;   // проверенные вершины

begin   { TGraph.GenClique }

  // Инициализация
  Result:= CreateSet;       // создать пустое множество-результат
  Res:= CreateSet;          // очередная клика
  Cand := CreateSet;        // кандидаты
  Cand.CopyItems(mNodes);   // Cand = все вершины графа
  Tested:= CreateSet;       // проверенные вершины
  Gamma:= CreateSet;        // соседние вершины

  Extend(Cand, Tested);  // рекурсивная процедура
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
  Res.Free;

end;

//#######################################################################
//                                                                     #
//            Формирование доминирующего множества                     #
//                  Кристофидес-3,  стр. 50                            #
//                       Граф + орграф                                 #
//                                                                     #
//#######################################################################

// TCostNodes - Элемент для ппоиска доминирующих множеств

constructor TCostNodes.Create(aCost: integer; aSet: TSet; aNode: TNode);
begin
  inherited Create(aCost, aSet, true);
  mNode:= aNode;
end;


procedure TCostNodes.Print(var aFile: TextFile);
var Node: TNode;
begin
  Write(aFile, mNode.GetName, ' = ', mCost, ' -> { ');
  Node:= TNode(mSet.GetFirst);
```

```
  while Assigned(Node) do begin
    Write(aFile, Node.GetName + ' ');
    Node:= TNode(mSet.GetNext);
  end;
  Writeln(aFile, '}');
end;

// Метод поиска доминирующих множеств

function TGraph.GenDominating: TCostSet;
var  Buf: TBuffer;    // промежуточный буфер
     Node: TNode;       // Вершина графа
     Gamma: TSet;       // для множества соседей
     CS: TCostNodes;    // оценённое подмножество

begin
  Buf:= TBuffer.Create;
  // В промежуточный буфер заносим оценённые подмножества,
  // каждое из которых соответствует гамме одной вершины
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Gamma:= CreateSet;         // создаём пустое множество
    Node.OutGammaGet(Gamma);   // Gamma= соседние вершины
    Gamma.Insert(Node);        // + текущая
    // Node.CalcCost вычисляет цену узла := вес узла + веса дуг
    CS:= TCostNodes.Create(Node.CalcCost, Gamma, Node);
    Buf.Put(CS);           // заносим в буфер
    Node:= NodeNext;
  end;
  // Поиск минимального покрытия
  Result:= CollectMinCover(mNodes, Buf);  // unit Assembly
  // Удаляем результат из буфера {unit SetUtils}
  RemoveItemsFromBuf(Buf, Result.mSet);
  // а буфер очищаем и уничтожаем
  Buf.ClrAndDestroy;
  Buf.Free;
end;

//#######################################################################
//                                                                     #
//                            Раскраска                                #
//                    Кристофидес-4,  стр. 86                          #
//                                                                     #
//#######################################################################

// Построение раскраски
// aQuick - параметр скорости:
// false = точно, но долго
// true  = быстро, но неточно

function TGraph.GenPaints(aQuick: boolean): TSet;
var  Undep : TSet;        // множество независимых множеств
       Buf : TBuffer;   // рабочий буфер
         S : TSet;        // текущее подмножество вершин
        CS : TCostSet;    // оценённое подмножество вершин
     Cover : TCostSet;    // минимальное покрытие
begin
  Undep:= GenUndepend;    // генерируем все независимые подмножества вершин
  // Переносим независимые множества в буфер оценённых подмножеств
  // для последующего поиска минимального покрытия
  Buf:= TBuffer.Create;       // создаём рабочий буфер
  S:= Undep.GetFirst as TSet;  // первое независимое множество
```

```
  while Assigned(S) do begin   // перебор независимых множеств
    // Создаём оценённые подмножества (цена Cost=1)
    // и помещаем в буфер:
    CS:= TCostSet.Create(1, S.Copy as TSet, true);
    Buf.Put(CS);
    S:= Undep.GetNext as TSet;
  end;
  // Уничтожаем множество максимальных независимых множеств
  Undep.ClrAndDestroy;
  Undep.Free;
  // Формируем минимальное покрытие вершин (unit Assembly):
  if aQuick
    then Cover:= CollectGradCover(mNodes, Buf)  // быстро
    else Cover:= CollectMinCover(mNodes, Buf);  // точно
  RemoveItemsFromBuf(Buf, Cover.mSet);  // удаляем результат из буфера
  Buf.ClrAndDestroy;                    // а буфер очищаем
  Buf.Free;                             // и удаляем
  // Формируем результат:
  Result:= Cover.mSet.Copy as TSet;  // копия множества-покрытия
  Cover.Free;                        // удаляем покрытие
  Result.CoverToDissect;             // превращаем покрытие в разбиение
end;

//#######################################################################
//                                                                      #
//                        Центры и P-центры                             #
//                      Кристофидес-5,  стр. 111                        #
//                                                                      #
//#######################################################################

// Элемент для представления ограниченных входящих гамм
// Используется для поиска p-центров

type  TGamma = class(TItem)
        mRoot  : TNode;  // достигаемая вершина
        mGamma : TSet;   // вершины, из которых достижима mRoot
        constructor Create(aRoot: TNode; aGamma: TSet);
        destructor Destroy;  override;
        function  Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: TextFile); override;
      end;

constructor TGamma.Create(aRoot: TNode; aGamma: TSet);
begin
  inherited Create;
  mRoot:= aRoot;       // центр гаммы
  mGamma:= aGamma;     // подмножество достижимых вершин
end;

destructor TGamma.Destroy;
begin
  mGamma.Free;
  inherited;
end;

function TGamma.Compare(arg: TItem): TCompare;
begin
  Result:= mRoot.Compare((arg as TGamma).mRoot)
end;

procedure TGamma.Print(var aFile: TextFile);
var S: string;
```

```
begin
  S:= mRoot.GetName + ' -> ' + NodesToStr(mGamma);
  Writeln(aFile, S);
end;

// TSortedNum - вспомогательный класс для формирования
// множества возрастающих чисел

type  TSortedNum = class(TItem)
        mNumber : integer;
        constructor Create(aNumber : integer);
        function  Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: TextFile); override;
      end;

constructor TSortedNum.Create(aNumber: integer);
begin
  inherited Create;
  mNumber:= aNumber;
end;

function TSortedNum.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result= cmpEq then Exit;
  if mNumber < (arg as TSortedNum).mNumber
    then Result:= cmpLess
  else if mNumber > (arg as TSortedNum).mNumber
    then Result:= cmpGreate  else Result:= cmpEq  ;
end;

procedure TSortedNum.Print(var aFile: TextFile);
begin
  Writeln(aFile,'Num= ',mNumber);
end;


// TAreal
// Элемент для представления областей
// Используется для поиска p-центров
// aAreal - достигаемые вершины (поле mSet)
// aGamma - вершины, из которых они достигаются (поле mGamma)

type  TAreal = class(TCostSet)
        // поле mSet - достигаемые вершины, унаследовано от TCostSet
        mGamma : TSet;  // вершины, из которых достижимы вершины в mSet
        constructor Create(aAreal, aGamma: TSet);
        destructor Destroy;  override;
        procedure Print(var aFile: TextFile); override;
      end;

constructor TAreal.Create(aAreal {куда},
                          aGamma {откуда} : TSet);
begin
  inherited Create(1, aAreal, true);
  mGamma:= aGamma;  // вершины, из которых достижимы вершины в mSet
end;

destructor TAreal.Destroy;
begin
  mGamma.Free;
  inherited;
```

```
end;

procedure TAreal.Print(var aFile: TextFile);
var S: string;
begin
  S:= NodesToStr(mGamma) + ' ==> ' + NodesToStr(mSet);
  Writeln(aFile, S);
end;


// Функция формирования центра графа.
// Возвращает множество вершин центра и взвешенный радиус aLambda
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenCenter(var aLambda: integer; aDirect: TCenter): TSet;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск максимального взвешенного расстояния от вершины aNode.
  // Использует ранее построенную карту дальних указателей mFarLinks

  function GetMaxDist(aNode: TNode): integer;
  var FL: TFarLink;
      Value: integer;
  begin
    Result:= 0;
    with aNode.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if mLoadNodes                      // если вершины графа нагружены...
          then Value:= FL.mNodeFar.mValue // для графа с нагруж. вершинами
          else Value:= 1;                  // если вершины не нагружены
        if Result < FL.mDist * Value
          then Result:= FL.mDist * Value;
        FL:= GetNext as TFarLink;
      end;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Dist, MinDist: integer;  // текущее и минимальное взвеш. расстояние
    Node: TNode;

begin  { TGraph.GenCenter }

  // Предварительно формируем карту дальних указателей,
  // в которой каждый элемент содержит расстояние между парами вершин.
  DoneMap;            // если карта создана, то ликвидируем её

  InitMap(aDirect);  // строим карту с заданным направлением

  // Выбор минимального из максимальных взвешенных расстояний
  MinDist:= MaxInt;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);
    if MinDist > Dist then MinDist:= Dist;
    Node:= NodeNext;
  end;
  Result:= CreateSet;

  // Формирование множества-центра
  Node:= NodeFirst;
```

```
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);     // число разделения очередной вершины
    if Dist = MinDist then Result.Insert(Node);
    Node:= NodeNext;
  end;
  aLambda:= MinDist;  // минимальное из максимальных расстояний
end;


// Вспомогательная процедура генерации распределённой карты
// aDirect = InCenter | OutCenter | InOutCenter

procedure TGraph.InitMap(aDirect: TCenter);

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей узла
  procedure InitFarLinks(aNode: TNode; aDirect : TCenter);
  var      Node : TNode;      // текущая вершина
           Link : TLink;      // ближний указатель
             FL : TFarLink;   // дальний указатель
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Для поиска внутреннего центра орграфа или центра графа
    procedure LocalIn;
    begin
      with Node do begin
        Link:= OutLinkFirst;             // первая исходящая связь
        while Assigned(Link) do begin    // пока существуют связи
          if Link.mDest = aNode then begin  // сравнение с конечной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;    // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= OutLinkNext;      // следующая исходящая связь
        end // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Для поиска внешнего центра орграфа
    procedure LocalOut;
    begin
      with Node do begin
        Link:= InLinkFirst;              // первая входящая связь
        while Assigned(Link) do begin    // пока существуют связи
          if Link.mOwner = aNode then begin // сравнение с исходной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;    // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= InLinkNext;      // следующая исходящая связь
        end; // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  begin   { InitFarLinks }
    PosPush;  // сохранить позицию перебора вершин
    Node:= NodeFirst;                 // перебор всех вершин графа
    while Assigned(Node) do begin     // цикл создания дальних связей
      if Node = aNode then begin
```

```
        FL:= TFarLink.Create(Node, Node, 0);   // это связь на себя
      end else with Node do begin
        // Создать связь на очередную вершину
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        if (aDirect=InCenter) or not mDirect
          then LocalIn   // Внутренний центр орграфа или центр графа
          else LocalOut; // Внешний центр орграфа
      end; // else
      aNode.mFarLinks.Insert(FL); // вставить в множество
      Node:= NodeNext;  // перебор всех вершин графа
    end;
    PosPop;  // восстановить позицию перебора вершин
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
         FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin  { Handle }
    Result:= false;
    FL1:= aNode.mFarLinks.GetFirst as TFarLink;  // указатель из вершины
aNode
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)             // если определён
             and (mDist<>0)                  // и не сам на себя
             and (mNodeFar<>aNode)           // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear) // то проверяем расстояние
                   then Result:= true;       // признак того, что изменён
          FL2:= GetNext as TFarLink;         // следующая дальняя связь
        end; // while
        PositionPop;
      end;
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure Create(aDirect : TCenter);
```

```
  var  Node : TNode;     // текущая вершина
       Step : integer;   // этап обработки (номер цикла)
       Flag : boolean;   // признак продолжения обработки
  begin
    // Предварительная очистка карты:
    Node:= NodeFirst;
    while Assigned(Node) do begin          // перебор вершин
      with Node do if Assigned(mFarLinks) // если карта существует
        then mFarLinks.ClrAndDestroy       // очищаем
        else mFarLinks:= CreateSet;        // иначе создаём пустую
      InitFarLinks(Node, aDirect);         // инициализация дальних указателей
      Node:= NodeNext;
    end;
    // Обработка вершин
    for Step:= 1 to mNodes.GetCount-2  do begin
      Flag:= false;
      Node:= NodeFirst;
      while Assigned(Node) do begin      // перебор вершин
        if Handle(Node, Step)            // если обновлялись дальние
указатели,
          then Flag:= true;              // то отметить это установкой флага
        Node:= NodeNext;
      end;
      // если дальние указатели не обновлялись, то прервать цикл
      if not Flag then Break;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var  Node : TNode;       // текущая вершина
     Buf : TBuffer;  // буфер для хранения списков дальних указателей
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Сохранение в буфере текущих расстояний
  procedure SaveInBuf;
  begin
    Node:= NodeFirst;
    // Сохранение списков в буфере:
    while Assigned(Node) do begin
      Buf.Put(Node.mFarLinks);
      Node.mFarLinks:= nil;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение из буфера и добавление расстояний
  procedure AddFromBuf;
  var     S : TSet;        // список из буфера
        FL1 : TFarLink;    // дальний указатель из S
        FL2 : TFarLink;    // дальний указатель из Node
  begin
    Node:= NodeFirst;
    // Перебор вершин:
    while Assigned(Node) do begin
      S:= Buf.Get as TSet;
      FL1:= S.GetFirst as TFarLink;
      while Assigned(FL1) do begin
        FL2:= Node.GetFarLink(FL1.mNodeFar);
        if Assigned(FL2) then FL2.mDist:= FL2.mDist + FL1.mDist;
        FL1:= S.GetNext as TFarLink;
      end;
      S.ClrAndDestroy;
      S.Free;
```

```
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.InitMap }

  if mInitMap then Exit;  // Защита от повторной инициализации
  mInitMap:= true;        // Признак, что карта создана

  mMapDirect:= aDirect;   // запоминаем направление карты

  case aDirect of
    InCenter,   // внешний центр и медиана
    OutCenter:  // внутренний центр и медиана
      Create(aDirect);
    InOutCenter: begin
      // внешне-внутренний центр и медиана
      // Строим карту для входящих гамм:
      Create(InCenter);
      // Создаём буфер и сохраняем в нём расстояния между вершинами:
      Buf:= TBuffer.Create;
      SaveInBuf;
      // Строим карту для исходящих гамм:
      Create(OutCenter);
      // Прибавляем расстояния, сохранённые в буфере,
      // и освобождаем буфер:
      AddFromBuf;
      Buf.ClrAndDestroy;
      Buf.Free;
    end;
  end;

  // Сортируем списки дальних указателей по неубыванию расстояния
  // (необходимо при поиске медиан)

  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.SortFarLinks;
    Node:= NodeNext;
  end;

end;

// Освобождение карты дальних указателей,
// применяемой при построении кратчайших путей и центров

procedure TGraph.DoneMap;
var  Node : TNode;    // текущая вершина
begin
  // Если очищена, то выход:
  if not mInitMap then Exit;
  mInitMap:= false;       // признак очищенной карты
  mMapDirect:= Undefined; // признак очищенной карты

  // Очистка распределённой по вершинам карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      if Assigned(mFarLinks) then begin
        mFarLinks.ClrAndDestroy;   // уничтожаем указатели
        mFarLinks.Free;            // и список
```

```
        mFarLinks:= nil;
      end;
    end;
    Node:= NodeNext;
  end;
end;

// Отображение распределённой карты

procedure TGraph.ExpoFarLinks;
var Node: TNode;
begin
  PosPush;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.ExpoFarLinks;
    Node:= NodeNext;
  end;
  PosPop;
end;

// Отображение дополнительных данных в связях

procedure TGraph.ExpoLinksData;
var Node: TNode;
begin
  PosPush;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.ExpoData;
    Node:= NodeNext;
  end;
  PosPop;
end;

// Формирование множества вершин, из которых узел aNode
// достижим в пределах взвешенного расстояния aLambda
// (ограниченная входящая гамма)
// Используется созданная ранее карта дальних указателей.

function TGraph.GenLimitGamma(aNode: TNode; aLambda: integer): TSet;
var FL : TFarLink;    // очередная дальняя связь
begin
  Result:= CreateSet;
  with aNode.mFarLinks do begin  // просматриваем дальние связи вершины
    FL:= GetFirst as TFarLink;
    while Assigned(FL) do begin
      if aLambda >= (FL.mDist * aNode.mValue) // если в достижымых пределах
        then Result.Insert(FL.mNodeFar);      // то вставляем в множество
      FL:= GetNext as TFarLink;
    end;
  end;
end;

// Построение множества ограниченных гамм TGamma

function TGraph.GenLimitGams(aLambda: integer): TSet;
var  Node : TNode;
     GammaSet: TSet;
     GammaItem: TGamma;
begin
  Result:= CreateSet;
```

```
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // формируем ограниченнную входящую гамму
    GammaSet:= GenLimitGamma(Node, aLambda);
    // формируем пару вершина+гамма
    GammaItem:= TGamma.Create(Node, GammaSet);
    Result.Insert(GammaItem);
    Node:= NodeNext;
  end;
end;

// Формирование буфера, содержащего пересечения гамм
// aGams - исходное множество областей достижимости,
// где достижимыми являются отдельные вершины

function TGraph.GenIntersections(aGams: TSet): TBuffer;
var
    GammaItem : TGamma;    // элемент для входящих гамм
        Nodes : TSet;      // достигаемые вершины (TAreal.mSet)
     GammaSet : TSet;      // откуда достигаются (TAreal.mGamma)
         Temp : TSet;      // здесь вычисляется пересечение гамм
      Existed : TSet;      // множество множеств уже обработанных областей
        Areal : TAreal;    // очередной элемент "область"
         Flag : boolean;   // признак пополнения буфера
          Cnt : integer;   // текущее количество элементов в буф. результ.
            i : integer;   // индекс
begin
  Result:= TBuffer.Create;     // буфер результата
  Temp:= CreateSet;            // здесь вычисляется пересечение гамм
  Existed:= CreateSet;         // множество множеств обработанных областей
  Nodes:= CreateSet;           // достигаемые вершины (TAreal.mSet)
  GammaSet:= CreateSet;        // откуда достигаются (TAreal.mGamma)
  GammaSet.CopyItems(mNodes); // копия множества всех вершин
  // Создаём и помещаем в буфер элемент-область TAreal такой, что:
  // mNodes - множество достигаемых вершин ПУСТО
  // mGamma - множество вершин, из которых достигается, содержит ВСЕ вершины
  Result.Put(TAreal.Create(Nodes, GammaSet));
  Flag:= false;
  repeat
    Cnt:= Result.GetCount;  // текущее количество элементов в буфере
    for i:=1 to Cnt do begin
      // Цикл по всем элементам буфера
      Flag:= false;                  // сброс признака пополнения буфера
      Areal:= Result.Get as TAreal; // очередной элемент из буфера
      // Перебор всех элементов исходного множества
      GammaItem:= aGams.GetFirst as TGamma;
      while Assigned(GammaItem) do begin
        if not Areal.mSet.Exist(GammaItem.mRoot) then begin
          // Если текущая область ещё не содержит корневой вершины
          // то вычисляем пересечение входящих гамм
          Temp.CopyItems(Areal.mGamma);      // гамма в текущей области
          Temp.Mul(GammaItem.mGamma);        // пересекаем с гаммой вершины
          if Temp.GetCount <> 0 then begin
            // если пересечение не пусто,
            // пытаемя создать и добавить в буфер новую область
            Nodes:= CreateSet;                // достигаемые (TAreal.mSet)
            Nodes.CopyItems(Areal.mSet);      // к существующим вершинам
            Nodes.Insert(GammaItem.mRoot);    // добавляем ещё одну вершину
            if Existed.Exist(Nodes) then begin
              // Если это подмножество достигаемых уже существует в наборе
              Nodes.Free;     // то удаляем дубликат
              //Nodes:= nil; // и ничего не вставляем
```

```
            end else begin
              // Если получено новое подмножество достигаемых вершин
              Existed.Insert(Nodes);      // запоминаем для будущих проверок
              // создаём новую область и вставляем в буфер
              GammaSet:= CreateSet;       // откуда достигаются
(TAreal.mGamma)
              GammaSet.CopyItems(Temp);  // копия пересечения GammaSet= Temp
              Result.Put(TAreal.Create(Nodes, GammaSet));
            end; // if
            Flag:= true;  // установить признак пополнения буфера
          end; // if
        end; // if
        GammaItem:= aGams.GetNext as TGamma;  // перебор множества гамм
      end; // while
      // После обработки очередного элемента буфера проверяем:
      // Если обработанный элемент буфера породил хотя бы один новый элемент,
      // то он является подмножеством этого нового и исключается
      // из дальнейшей обработки, а иначе возвращается в буфер
      if Flag then begin
        Existed.Delete(Areal.mSet);// удаляем достигаемые из проверяемых
        Areal.Free;                // удаляем ненужный элемент буфера
      end else begin
        Result.Put(Areal);         // а иначе возвращаем элемент в буфер
      end;
    end; // for
    // выход, если буфер не изменился
  until not Flag and (Cnt=Result.GetCount);
  Temp.Free;
  Existed.Free;
end;


// Поиск множества центров
// с заданной константой проникновения aLambda (дистанцией)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPCentLambda(aLambda: integer; aDirect: TCenter): TCostSet;
var   Gams : TSet;
       Buf : TBuffer;
begin
  // Формирование карты дальних указателей (направление карты инверсно):
  case aDirect of
    InCenter:    InitMap(OutCenter);
    OutCenter:   InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование входящих гамм
  // с заданной константой проникновения (дистанцией) aLambda
  Gams:= GenLimitGams(aLambda);
  // Формирование всех непустых пересечений
  Buf:= GenIntersections(Gams);
  // Решение задачи о минимальном покрытии (ЗНП)
  Result:= CollectMinCover(mNodes, Buf);
  // Удаляем результат из буфера
  RemoveItemsFromBuf(Buf, Result.mSet);
  // Освобождаем множество входящих гамм и буфер
  Gams.ClrAndDestroy;
  Gams.Free;
  Buf.ClrAndDestroy;
  Buf.Free;
end;
```

```
// Поиск заданного количества P-центров
// с возвращением взвешенного радиуса aLambda
// (обратная задача).
// aPoly - максимальное количество центров
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPCenters(aPoly: integer;
                            var aLambda: integer;
                            aCenter: TCenter): TCostSet;

var Lambdes : TSet;       // лямбда-список (возрастающее множество
расстояний)
       LRec : TSortedNum; // элемент лямбда-списка
     Lambda : integer;    // текущий взвешенный радиус

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества взвешенных расстояний (уникальных)
  procedure LambdasCreate;

    procedure AddLambdas(aNode: TNode);
    var FL : TFarLink;     // очередная дальняя связь
        SN : TSortedNum;   // элемент сортированного списка чисел
        Lambda: integer;   // взвешенное расстояние от текущей вершины
    begin
      with aNode.mFarLinks do begin
        FL:= GetFirst as TFarLink;
        while Assigned(FL) do begin
          Lambda:= FL.mDist * aNode.mValue;
          SN:= TSortedNum.Create(Lambda);
          if not Lambdes.Insert(SN) then SN.Free;
          FL:= GetNext as TFarLink;
        end;
      end;
    end;

  var  Node : TNode;     // текущая вершина

  begin  { LambdasCreate }
    Lambdes:= CreateSet;
    Node:= NodeFirst;
    while Assigned(Node) do begin    // перебор вершин
      AddLambdas(Node);              // обработать вершину
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenPCenters }
  // Формирование карты дальних указателей:
  case aCenter of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование лямбда-списка (возрастающего множества расстояний)
  LambdasCreate;
  // Выбор первого (наименьшего) элемента взвешенного расстояния
  LRec:= Lambdes.GetFirst as TSortedNum;
  repeat
    Lambda:= LRec.mNumber;            // очередное взвешенное расстояние
    Result:= GenPCentLambda(Lambda, aCenter);  // множество центров
```

```
    // Если количество центров достигнуто, то выход:
    if Result.mSet.GetCount <= aPoly then Break;
    // а иначе очищаем результат и берём следующий по величине радиус
    Result.ClrAndDestroy;
    Result.Free;
    Result:= nil;
    // Выбор очередного элемента взвешенного радиуса (по возрастанию)
    LRec:= Lambdes.GetNext as TSortedNum;
  until not Assigned(LRec);
  // Возвращаем взвешенный радиус:
  aLambda:= Lambda;
  // Очистка и уничтожение списка расстояний
  Lambdes.ClrAndDestroy;
  Lambdes.Free;
  // Очистка карты дальних указателей:
  DoneMap;
end;

//#######################################################################
//                                                                      #
//                          P-Медианы                                   #
//                   Кристофидес,  стр. 135                             #
//                                                                      #
//#######################################################################

// Копирование текущей "медианы" (чёрных вершин)

procedure TGraph.MedianCopy(aRes: TSet);
var N: TNode;
begin
  aRes.Clear;
  N:= NodeFirst;
  while Assigned(N) do begin
    if N.mColor = CBlack then aRes.Insert(N);
    N:= NodeNext;
  end;
end;

// Вычисление стоимости p-подмножества
// (вершины в этот момент уже раскрашены)
// Цвета вершин означают:
// CWhite -- белые ещё не опробованы
// CBlack -- чёрные пробуются в качестве медианных
// CGray  -- серые уже побывали чёрными и не могут быть медианными

function TGraph.MedianCost: integer;
var Node: TNode;        // текущая вершина
    Dist: integer;      // расстояние к ближайшей чёрной
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возвращает расстояние к ближайшей чёрной вершине
  function FindMinDist: integer;
  var FL: TFarLink;     // элемент дальней связи
  begin
    Result:= CInfinity;
    with Node.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if FL.mNodeFar <> Node then begin
          if FL.mNodeFar.mColor = CBlack then begin
            Result:= FL.mDist;
            Break;
          end;
```

```
        end;
        FL:= GetNext as TFarLink;
      end;
    end; // with
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin  { MedianCost }
  // Перебираем вершины графа (матрицу расстояний),
  // (вклад в стоимость дают только белые и серые вершины)
  Result:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mColor <> CBlack then begin
      // чёрные вершины не дают вклад в стоимость
      Dist:= FindMinDist;  // минимальное расстояние для Node
      Inc(Result, Node.mValue * Dist);   // накопление результата
    end;
    Node:= NodeNext;
  end;
end;

// Поиск медианы "в лоб" перебором
// всех комбинаций медианных подмножеств (демонстрационный метод)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian_A (
                     aPoly: integer;     // кратность (P)
                     aDirect: TCenter;   // внешняя, внутр., внешне-внутр.
                     var aCost: integer  // стоимость (результат)
                    ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивный поиск перебором всех возможных
  // комбинаций чёрных вершин
  // (кандидатов в медианное множество)

  procedure Searching(aIndex: integer { стартовый индекс } );
  var i: integer;
      N: TNode;         // текущий узел
      Cost: integer;    // текущая стоимость p-подмножества
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже оптимальной, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут,
        // погружаемся на следующий уровень
        Inc(Level);        // Level+1
        Searching(i);      // рекурсивный вызов следующего уровня
```

```
        Dec(Level);        // Level-1
      end;
      N.mColor:= CGray;    // проверенной вершине назначаем серый цвет
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenPMedian_A }
  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
  // Сформировать списки ближайших вершин:
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;                      // начальный уровень в дереве перебора
  NN:= mNodes.GetCount+1-aPoly; // количество узлов, обраб. на каждом уровне
  Searching(1);          // начать перебор с первой вершины
  DoneMap;               // удалить множества дальних указателей
  aCost:= BestCost;      // вернуть цену
end;


// Поиск медианы частичным перебором p-подмножеств (основной метод)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian(aPoly: integer;     // кратность
                           aDirect: TCenter;   // внешняя, внутр., внешне-
внутр.
                           var aCost: integer  // стоимость (результат)
                           ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Оценка минимальной стоимости текущего распределения вершин
  // Цвета вершин означают:
  // CWhite -- белые ещё не опробованы
  // CBlack -- чёрные пробуются в качестве медианных
  // CGray  -- серые побывали чёрными и не могут быть медианными

  function EvalBest(aBest: integer): boolean;
  var Node: TNode;        // текущая вершина
        FL: TFarLink;     // элемент дальней связи
       Sum: integer;      // накопитель суммы
      Dist: integer;      // расстояние к ближайшей не серой вершине
         S: TSet;         // неубывающее множество элементов FL
      WhiteCnt: integer; // счётчик белых вершин

    // Поиск ближайшей не серой вершины
    // Node -- текущий не чёрный узел
    function FindMinDist: integer;
    var FL: TFarLink;     // элемент дальней связи
    begin
      Result:= CInfinity;
      with Node.mFarLinks do begin
        FL:= GetFirst as TFarLink;
```

```
        while Assigned(FL) do begin
          if (FL.mNodeFar <> Node) and
             (FL.mNodeFar.mColor <> CGray) then begin
            // Можно прикреплять к чёрным и белым
            Result:= FL.mDist;
            Break;
          end;
          FL:= GetNext as TFarLink;
        end;
      end; // with
    end;

  begin  { Eval }
    // Перебираем вершины графа (матрицу расстояний),
    // определяем минимальное расстояние к не серым вершинам,
    // и формируем список из неубывающих чисел (расстояний)
    // (вклад в стоимость дают только белые и серые вершины)

    Sum:= 0;  // накопитель суммы
    // начальное значение счётчика суммируемых белых вершин
    WhiteCnt:= mNodes.GetCount - aPoly;
    S:= CreateSet;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mColor <> CBlack then begin
        // чёрные не дают вклад в стоимость
        Dist:= FindMinDist;  // минимальное расстояние для Node
        case Node.mColor of
          CWhite: begin
                    // взвешенное расстояние к белой вершине
                    // вставляем в неубывающий список
                    FL:= TFarLink.Create(nil, Node, Node.mValue*Dist);
                    S.Insert(FL);
                  end;
          CGray:  begin
                    // серую суммируем и уменьшаем счётчик белых
                    Inc(Sum, Node.mValue*Dist); // накопление результата
                    Dec(WhiteCnt);      // счётчик суммируемых вершин -1
                    if Sum>=aBest then Break;
                  end;
        end;
      end;
      Node:= NodeNext;
    end;
    // По окончании формирования множества расстояний
    // добавляем WhiteCnt первых из них к накопленному результату
    FL:= S.GetFirst as TFarLink;
    while (WhiteCnt > 0) and (Sum < aBest) do begin
      Inc(Sum, FL.mDist);            // накопление суммы
      FL:= S.GetNext as TFarLink;    // следующий элемент списка
      Dec(WhiteCnt)
    end;
    // Очистка и удаление временного списка:
    S.ClrAndDestroy;
    S.Free;
    Result:= Sum < aBest;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивный поиск перебором части комбинаций чёрных вершин
  // (кандидатов в медианное множество)

  procedure Searching(aIndex: integer { стартовый индекс } );
```

```
  var i: integer;
      N: TNode;          // текущий узел
      Cost: integer;     // текущая стоимость p-подмножества
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже оптимальной, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут.
        // Сравним оценку минимальной стоимости с лучшей текущей.
        // Если она меньше текущей, то "ныряем" на уровень вниз
        if  EvalBest(BestCost) then begin
          // погружаемся на следующий уровень
          Inc(Level);     // Level+1
          Searching(i);   // рекурсивный вызов следующего уровня
          Dec(Level);     // Level-1
        end;
      end;
      N.mColor:= CGray;  // проверенной вершине назначаем серый цвет
    end;
    // Перед возвратом на уровень вверх возвращаем белый цвет
    // всем ОБРАБОТАННЫМ вершинам текущего уровня
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CWhite;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.GenPMedian }

  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
  // Сформировать множества ближайших вершин:
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;             // текущий уровень в дереве поиска
  NN:= mNodes.GetCount+1-aPoly; // кол-во узлов, обраб. на каждом уровне
  Searching(1);         // начать перебор с первой вершины
  DoneMap;              // удалить множества дальних указателей
  aCost:= BestCost;     // вернуть цену
end;


//#######################################################################
//                                                                      #
```

```
//                  ПОКРЫВАЮЩИЕ  ДЕРЕВЬЯ                                    #
//                                                                         #
//#########################################################################

type // Линк для построения покрывающего дерева (остова)

  TTreeLink = class (TLink)
    mMaxTree: boolean;  // false -- миним., true -- макс. остов
    constructor Create(aLink: TLink; aMaxTree: boolean);
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;

// Конструируется по данным исходного линка

constructor TTreeLink.Create(aLink: TLink; aMaxTree: boolean);
begin
  inherited Create(aLink.mOwner,  // источник связи
                   aLink.mDest,   // приёмник связи
                   aLink.mValue,  // вес связи
                   true           // нагруженность связей
                   );
  mMaxTree:= aMaxTree;
end;

function TTreeLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  // Для неориентированного графа сравниваем исходную и конечную вершины
  // на предмет встречных линков
  if not mOwner.mOwner.mDirect and  // если не орграф
    (mOwner=(arg as TTreeLink).mDest) and (mDest=(arg as TTreeLink).mOwner)
    then begin
      // Здесь линки направлены встречно, отвергаем дубликат:
      Result:= cmpEq;
      Exit;
    end;
  // Если линки не совпадают, то сортируем по неубыванию веса (длины)
  Result:= cmpLess;
  if mValue > (arg as TTreeLink).mValue then Result:= cmpGreate;
  // Для построения дерева максимального веса результат инвертируем
  if mMaxTree
    then if Result=cmpLess then Result:= cmpGreate else Result:= cmpLess;
end;

// В орграфе выводим по направлению стрелок,
// а в графе - по алфавиту

procedure TTreeLink.Print(var aFile: TextFile);
var N1, N2 : TNode;
begin
  N1:=mOwner;  N2:=mDest;
  if not mOwner.mOwner.mDirect and
     (mOwner.Compare(mDest) = cmpGreate) then begin
    N1:=mDest;   N2:=mOwner;
  end;
  Write(aFile, ' ' + N1.GetName + N2.GetName);
  if mLoadLinks then Write(aFile, '= ', mValue);
end;

// Добавляе линков, ведущих к ближайшим белым вершинам
```

```
// aLinks - накопитель (буфер) линков
// Линки выстраиваются по неубыванию (aMaxTree=false)
// или невозрастанию дистанции (aMaxTree=true)
// Используется для построения остовных деревьев

procedure TNode.AddTreeLinks(aMaxTree: boolean; aLinks: TSet);
var  L : TLink;
    TL : TTreeLink;
begin
  // Обработка исходящих связей
  L:= OutLinkFirst;
  while Assigned(L) do begin
    // Вставляем только линки, ведущие к белым вершинам
    if (L.mDest.mColor = CWhite) then begin
      TL:= TTreeLink.Create(L, aMaxTree);
      if not aLinks.Insert(TL) then TL.Free;
    end;
    L:= OutLinkNext;
  end;
end;

// Алгоритм Прима (Кристофидес, стр. 162)
// Построение минимального (максимального) покрывающего дерева
// Возвращает множество линков TTreeLink  остова

function TGraph.GenCoverTree(aMaxTree: boolean;  // false= min, true = max
                             var aCost: integer  // возвр. стоимость
                            ): TSet;             // множество TTreeLink

var Node: TNode;       // текущая вершина
      TL: TTreeLink;   // текущий линк для дерева
   Links: TSet;        // множество линков для дерева

begin
  aCost:=0;               // накопитель суммы
  Result:= CreateSet;     // множество линков для результата
  Links:= CreateSet;      // множество линков для буфера
  ResetNodes;             // очистка для всех вершин mColor = CWhite
  Node:= NodeFirst;       // исходная вершина
  // Присоединение всех вершин графа (линков на единицу меньше)
  while Assigned(Node) and
       (Result.GetCount < mNodes.GetCount-1) do  begin
    Node.mColor:= CBlack;      // присоединяем текущую к множеству чёрных
    // Добавляем в множество Links линки к ближайшим белым вершинам:
    Node.AddTreeLinks(aMaxTree, Links);
    // Перебираем линки Links в поиске ближайшей белой вершины
    TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
    Node:= nil;                         // искомая вершина пока пуста
    while Assigned(TL) do begin
      Links.Delete(TL);                 // удаляем линк из буфера Links
      // Искомая вершина может быть как источником,
      // так и приёмником связи:
      if TL.mOwner.mColor = CWhite
        then Node:= TL.mOwner   // источник связи
      else if TL.mDest.mColor = CWhite
        then Node:= TL.mDest;   // приёмник связи
      if Assigned(Node) then begin
        // Здесь ближайшая белая вершина найдена:
        Result.Insert(TL);      // вставляем линк в результат
        Inc(aCost, TL.mValue);  // наращиваем стоимость
        Break;
      end;
```

```
      // здесь линк связывает две чёрные вершины:
      TL.Free;                          // удаляем ненужный линк
      TL:= Links.GetNext as TTreeLink; // и берём следующий
    end;
  end;
  // Если остов не построен, то граф не связан
  if Result.GetCount < mNodes.GetCount-1 then begin
    Result.ClrAndDestroy;
    aCost:=0;
  end;
  // Очистка и удаление вспомогательного множества линков:
  Links.ClrAndDestroy;
  Links.Free;
end;


// Поиск минимального ориентированного дерева
// Возвращает корень и вес дерева и само дерево
// Если дерево не существует, возвращаются пустые значения и MaxInt

function TGraph.GenCoverDir(var aRoot: TNode;   // корень
                            var aCost: integer  // вес (стоимость) дерева
                           ): TSet;             // само дерево
var Tree: TSet;      // текущее дерево
    Links: TSet;     // текущее множество линков
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Локальная функция формирует дерево, начиная с вершины aRoot
  // алгоритмом Прима и возвращает его вес (стоимость).
  // Не сформированное дерево очищается

  function RootCover(aRoot: TNode): integer;
  var Node: TNode;        // очередная вершина
      TL : TTreeLink;     // очередной линк
  begin
    Result:= 0;           // вес дерева
    ResetNodes;           // для всех вершин mColor = CWhite
    Node:= aRoot;         // исходная вершина -- корень
    // Пытаемся присоединенить все вершины графа (линков на единицу меньше)
    while Assigned(Node) and
         (Tree.GetCount < mNodes.GetCount-1) do  begin
      Node.mColor:= CBlack;      // текущую в множество чёрных
      // добавляем линки к ближайшим белым вершинам:
      Node.AddTreeLinks(false, Links);
      // Перебираем линки Links в поиске ближайшей белой вершины
      TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
      Node:= nil;                         // искомая вершина пока пуста
      while Assigned(TL) do begin         // пока существуют линки
        Links.Delete(TL);                 // удаляем линк из буфера Links
        // Искомая вершина может быть только приёмником связи:
        if TL.mDest.mColor = CWhite then begin
          // Здесь ближайшая белая вершина найдена:
          Node:= TL.mDest;         // будет обработана в следующем цикле
          Tree.Insert(TL);         // вставляем линк в результат
          Inc(Result, TL.mValue); // наращиваем стоимость
          Break;
        end;
        // здесь линк связывает две чёрные вершины:
        TL.Free;                          // удаляем ненужный линк
        TL:= Links.GetNext as TTreeLink; // и берём следующий
      end;
    end;
    Links.ClrAndDestroy; // очистка оставшихся в очереди линков
```

```
    // Если остов не построен, очищаем накопитель линков
    if Tree.GetCount < mNodes.GetCount-1 then begin
      Tree.ClrAndDestroy;  // очистка поддерева
      Result:= MaxInt;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var Root: TNode;    // очередной корень дерева
    Cost: integer;  // стоимость (вес) дерева

begin { TGraph.GenCoverDir }
  Result:=nil;  aRoot:= nil;
  aCost:= MaxInt;
  Links:= CreateSet;  // текущее множество линков
  Tree:= CreateSet;   // текущее остовное дерево
  // Перебор всех вершин:
  Root:= NodeFirst;
  while Assigned(Root) do begin
    Cost:= RootCover(Root);
    // Если дерево построено:
    if Tree.GetCount > 0 then begin
      // Если стоимость меньше максимальной
      if Cost<aCost then begin
        // то сохраняем результат
        if Assigned(Result)
          then Result.ClrAndDestroy // удаление существующих линков
          else Result:=CreateSet;   // создание копии дерева
        Result.CopyItems(Tree);     // копируем дерево (линки)
        Tree.Clear;                 // удаляем сохранённые линки
        aRoot:= Root;               // запоминаем корень
        aCost:= Cost;               // запоминаем стоимость (вес)
      end else begin
        Tree.ClrAndDestroy;  // очистка дерева, если оно не минимально
      end;
    end;
    Root:= NodeNext;
  end;
  // Удаление рабочих буферов:
  Tree.Free;
  Links.Free;
end;

// Построение всех остовных деревьев (Кристофидес, стр. 149)
// для неориентированного графа

procedure TGraph.ExpoAllCovers;
var
  Links: TSet;       // множество всех линков графа
   Tree: TSet;       // множество линков дерева
  Level: integer;    // текущий уровень
     NN: integer;    // количество линков, обрабатываемых на уровне
    Cnt: integer;    // счётчик остовных деревьев
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure Local_Init;
  var Node: TNode;   // текущая вершина
  begin
    ResetNodes;          // для всех вершин mColor=0, mPred=nil
    Tree:= CreateSet;    // множество для остовного дерева
    Links:= CreateSet;   // множество для всех линков
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
```

```
      mRoot:= Node;          // корень поддерева
      AddTreeLinks(false, Links);  // добавление линков вершины
    end;
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вспомогательная функция вычисления стоимости остова
// как суммы весов его рёбер

function Cost(aTree: TSet): integer;
var TL: TTreeLink;  // текущий линк для дерева
begin
  Result:= 0;
  TL:= aTree.GetFirst as TTreeLink;
  while Assigned(TL) do begin
    Inc(Result, TL.mValue);
    TL:= aTree.GetNext as TTreeLink;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Объединение двух поддеревьев
// aTL.mLink.mOwner -- принадлежит поддереву T1
// aTL.mLink.mDest  -- принадлежит поддереву T2
function Union(aTL: TTreeLink): boolean;
var   N2: TNode;   // узел в T2
      p2: TNode;   // предшествующий узел
      tmp: TNode;  // временное хранилище
begin
  with aTL do begin
    // Если концы линка принадлежат одному поддереву, то выход
    Result:= mOwner.mRoot <> mDest.mRoot;
    if not Result then Exit;
    // Подготовка к реверсу обратных ссылок mPred
    N2:= mDest;         // текущая вершина в поддереве T2
    p2:= N2.mPred;      // предшествующая вершина в T2
    N2.mPred:=mOwner;   // новая предшествующая взята из T1
    // Реверсирование обратных ссылок:
    while Assigned(p2) do begin
      tmp:= p2.mPred;   // временно сохранить ссылку в T2
      p2.mPred:= N2;    // обновить обратную ссылку
      N2:= p2;          // сдвинуть вершину по направлению к 2-му корню
      p2:= tmp;         // восстановить ссылку в T2
    end;
    // Замена 2-го корня на 1-й:
    // root1 = mOwner.mRoot.Expo
    // root2 = mDest.mRoot.Expo
    p2:= mDest.mRoot as TNode;    {###}
    N2:= NodeFirst;
    while Assigned(N2) do begin
      if N2.mRoot = p2 then N2.mRoot:= mOwner.mRoot;
      N2:= NodeNext;
    end;
  end;  // with
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Удаление ребра и расщепление дерева
procedure Remove(aTL: TTreeLink);
var Root: TNode;  // Новый корень для 2-го поддерева
    S1 : TSet;    // внутренний слой вершин
    S2 : TSet;    // внешний слой вершин
    Node: TNode;
```

```
begin
  S1:= CreateSet;    S2:= CreateSet;
  // Выбираем новый корень для 2-го поддерева
  with aTL do if mOwner.mPred = mDest
    then Root:= mOwner     // источник связи
    else Root:= mDest;     // приёмник связи
  Root.mRoot:= Root;       // пометка корня самим собой
  Root.mPred:= nil;        // обратная ссылка в корне пуста
  S1.Insert(Root);         // организуем нулевой слой из корневой вершины
  repeat
    // Накопление внешнего слоя:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // если примыкает к внутреннему слою, то добавляем к внешнему
      if S1.Exist(Node.mPred) then begin
        Node.mRoot:= Root;
        S2.Insert(Node);
      end;
      Node:= NodeNext;
    end;
    // Выход, если внешний слой пуст
    if S2.GetCount=0 then Break;
    // Копирование внешнего слоя во внутренний и его очистка
    S1.CopyItems(S2);
    S2.Clear;
  until false;
  S1.Free; S2.Free;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Рекурсивный перебор всех подмножеств
// (комбинаций) из L по N-1 линков
// L - количество рёбер или дуг графа = Links.GetCount
// N - количество вершин графа = mNodes.GetCount

procedure Searching(aIndex: integer { стартовый индекс } );
var  i: integer;
    TL: TTreeLink;  // текущий линк для дерева
begin
  // Добавляем последующие вершины
  for i:= aIndex to NN do begin
    TL:= Links.GetItem(i + Level) as TTreeLink;  // очередной линк
    if Union(TL) then begin
      // Этот линк не создаёт цикла, вставляем в остов
      Tree.Insert(TL);
      if Level = mNodes.GetCount-2 then begin
        // Остов сформирован
        Inc(Cnt);       // счётчик остовных деревьев
        Write(Cnt:4, '':2, Cost(Tree):3, '':2);
        Tree.Expo;
      end else begin
        // Здесь размер подмножества не достигнут, остов не сформирован,
        // погружаемся на следующий уровень
        Inc(Level);     // Level+1
        Searching(i);   // рекурсивный вызов следующего уровня
        Dec(Level);     // Level-1
      end;
      // Возвращаем предыдущее состояние поддеревьев
      Tree.Delete(TL);  // удаляем из остовного множества
      Remove(TL);       // и расщепляем дерево
    end; // if
  end; // for
end;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin { TGraph.ExpoAllCovers }
  Local_Init;
  Cnt:= 0;      // счётчик остовных деревьев
  Level:=0;     // текущий уровень
  // количество линков, обрабатываемых на каждом уровне:
  // (mNodes.GetCount - 1) = количество рёбер дерева
  NN:= Links.GetCount + 1 - (mNodes.GetCount - 1);
  Searching(1);
  Links.ClrAndDestroy;
  Links.Free;
end;

//#######################################################################
//                                                                      #
//                    ПОТОКОВЫЕ  АЛГОРИТМЫ                              #
//                                                                      #
//#######################################################################

// Вычисление максимального потока
// при условии отсутствия минимальных потоков (для всех дуг mLow=0)
// Майника Э. Алгоритмы оптимизации на сетях и графах, 4.2 (стр. 91)

function TGraph.CalcMaxFlow0(aSource, aDest: TNode): integer;
var Delta: integer;   // очередное приращение потока
    Que: TBuffer;   // очередь вершин
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт начального потока
  // на случай, когда заданы минимальные потоки в сети (mLow>0)

  function CalcStartFlow: integer;
  var Link: TLink;    // исходящая дуга
  begin
    Result:= 0;
    Link:= aSource.OutLinkFirst;
    while Assigned(Link) do begin
      Inc(Result, Link.mFlow);
      Link:= aSource.OutLinkNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Построение увеличивающей цепи
  // с подсчётом соответствующего приращения потока

  function CalcDelta: integer;
  var Node: TNode;
      Link: TLink;    // исходящая или входящая дуга
  begin
    Result:=0;
    // установка mColor=0, mPred=nil, mDist= MaxInt, mFlow= MaxInt
    ResetNodes;
    Que.Clear;   // очистка очереди вершин
    // Пометку вершин начинаем с источника
    aSource.mColor:= CGray;
    Que.Put(aSource);
    while Que.GetCount>0 do begin
      Node:= Que.Get as TNode;
      // Выход из цикла, если помечен сток
      if Node=aDest then Break;
      // Обработка исходящих связей текущего узла:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
```

```
        // Помечаем только белые вершины, достигаемые через ненасыщенные дуги
        if (Link.mDest.mColor=CWhite)            // если вершина не помечена
           and (Link.mFlow < Link.mHigh) then begin  // и дуга не насыщена
          Link.mDirect:= +1;  // это увеличивающаяся дуга
          // помечаем концевую вершину и ставим её в очередь:
          with Link.mDest do begin
            mColor:= CGray;    // цвет концевой вершины серый
            mPred:= Node;      // предшествующая вершина
            mLink:= Link;      // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mHigh - Link.mFlow); // поток
          end;
          Que.Put(Link.mDest); // в очередь приёмник дуги
        end;
        Link:= Node.OutLinkNext;
      end;
      // Обработка входящих связей текущего узла:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        // Помечаем только белые вершины, из которых поступает ненулевой
поток
        if (Link.mOwner.mColor=CWhite)          // если вершина не помечена
           and (Link.mFlow > Link.mLow) then begin // и поток можно уменьшить
          Link.mDirect:= -1;  // это уменьшающая дуга
          // помечаем вершину (источник дуги) и ставим её в очередь:
          with Link.mOwner do begin
            mColor:= CGray;    // цвет исходной вершины серый
            mPred:= Node;      // предшествующая вершина
            mLink:= Link;      // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
          end;
          Que.Put(Link.mOwner); // в очередь источник дуги
        end;
        Link:= Node.InLinkNext;
      end;  // while
    end;  // while
    // Если сток не помечен, то увеличивающая цепочка не найдена
    // и приращение потока равно нулю
    if aDest.mColor = CWhite then Exit;
    // Здесь  увеличивающая цепочка найдена
    Result:= aDest.mFlow; // допустимое увеличение потока
    // Обратное движение от стока к истоку с корректировкой потока
    Node:= aDest;  // сток
    while Node <> aSource do begin
      with Node.mLink do Inc(mFlow, mDirect * Result); // mDirect = +1 / -1
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.CalcMaxFlow0 }
  Result:= 0;
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
  Result:= CalcStartFlow; // начальный поток
  Que:= TBuffer.Create;   // рабочая очередь вершин
  // Повторяем поиск увеличивающих цепей, пока это возможно
  repeat
    Delta:= CalcDelta;    // подсчёт приращения потока
    Inc(Result, Delta);   // накопление результата
  until Delta=0;
  Que.Free;
end;
```

```
// Вычисление минимального потока
// (когда минимальное значение потока больше нуля)
// Басакер Р. Саати Т. Конечные графы и сети, стр.327 (7.8)

function TGraph.CalcMinFlow(aSource, aDest: TNode): integer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Восстановление данных о потоке
  // после удаления искусственных вершин и дуг
  // arg = false -- минимальный поток не существует
  // arg = true  -- минимальный поток найден

  procedure RestoreFlowData(arg: boolean);
  var Node: TNode;
      Link: TLink;       // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        // Перебор исходящих линков вершины:
        Link:= OutLinkFirst;
        while Assigned(Link) do begin
          with Link do begin
            mLow:= mTemp;            // минимум
            mHigh:= mHigh + mLow;    // максимум
            // текущий минимальный поток:
            if arg
              then mFlow:= mFlow + mLow  // в случае успеха
              else mFlow:= 0;            // если поток не существует
          end; // with
          Link:= OutLinkNext;
        end; // while
      end; // with
      Node:= NodeNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;       // текущая вершина
    Link: TLink;       // текущий линк
    Reverse: TLink;    // обратный линк T->S во вспомогательном графе
    Flow: integer;     // максимальный поток
    SumLow: integer;   // сумма всех минимумов
    Sx: TNode;         // вспомогательный исток
    Tx: TNode;         // вспомогательный сток
    OldLink: TLink;    // линк T->S в исходном графе
    OldLow, OldHigh: integer; // сохранённые данные обратного линка

begin    { TGraph.CalcMinFlow }

  // Если не указаны вершины, то выход
  if not Assigned(aSource) or not Assigned(aDest) then begin
    Result:=-1;
    Exit;
  end;

  // Создаём:
  Sx:= TNode.Create(0, Self); // вспомогательный исток
  Tx:= TNode.Create(0, Self); // вспомогательный сток
  SumLow:=0;     // здесь накапливаем сумму всех минимальных потоков
  // Первый перебор вершин с подсчётом сумм исходящих минимальных потоков:
  Node:= NodeFirst;
```

```
while Assigned(Node) do begin
  Node.mFlow:=0; // сумма исходящих минимумов
  with Node do begin
    // Перебор исходящих линков вершины:
    Link:= OutLinkFirst;
    while Assigned(Link) do begin
      with Link do begin
        Inc(Node.mFlow, mLow);  // сумма исходящих минимумов
        mHigh:= mHigh - mLow;   // пересчёт максимума
        mTemp:= mLow;           // временно сохраняем минимум
        mLow:= 0;               // и обнуляем его
        mFlow:=0;               // обнуляем поток
      end;
      Link:= OutLinkNext;
    end;
  end; // with
  Node:= NodeNext;
end; // while

// Второй перебор вершин с подсчётом сумм входящих минимальных потоков:
Node:= NodeFirst;
while Assigned(Node) do begin
  with Node do begin
    // Перебор входящих линков вершины:
    Link:= InLinkFirst;
    while Assigned(Link) do begin
      Dec(Node.mFlow, Link.mTemp);   // минус сумма исходящих минимумов
      Link:= InLinkNext;
    end;
  end;
  // Если вершина является источником или стоком:
  if Node.mFlow <> 0 then begin
    // Если вершиная является источником:
    if Node.mFlow > 0 then begin
      // создаём дугу из вершины во вспомогательный сток
      Link:= Node.MakeLink(Tx, 0);  // создаём линк
      Link.mHigh:= Node.mFlow;      // с данными о потоке
    end else begin
      // создаём дугу из вспомогательного истока в вершину
      Link:= Sx.MakeLink(Node, 0);  // создаём линк
      Link.mHigh:= -Node.mFlow;      // с данными о потоке
    end
  end;
  Inc(SumLow, Abs(Node.mFlow)); // двойная сумма всех минимальных потоков
  Node:= NodeNext;
end; // while
SumLow:= SumLow div 2;  // одинарная сумма всех минимальных потоков

// Если сумма минимальных потоков = 0, то выходим из процедуры
if SumLow=0 then begin
  // Освобождаем:
  Sx.Free;     // вспомогательный исток
  Tx.Free;     // вспомогательный сток
  // Восстанавливаем данные о потоках:
  RestoreFlowData(false);
  Result:=0;
  Exit;
end;

// Вставляем вспомогательные вершины в граф:
mNodes.Insert(Sx);   // исток
mNodes.Insert(Tx);   // сток
```

```
// Соединяем обратной дугой сток с истоком
// (поток в обратной дуге не ограничен):

// На случай отсутствия обратной дуги:
OldLow:= 0; OldHigh:=0;

// Обратная дуга уже существует?
OldLink:= aDest.GetLink(aSource);
if Assigned(OldLink) then begin
  // да, назначаем её в качестве реверсивной и запоминаем параметры
  Reverse:= OldLink;    // реверсивная дуга
  with OldLink do begin
    // да, запоминаем данные потока
    OldLow := mLow;     // нижний предел
    OldHigh:= mHigh;    // верхний предел
    // и устанавливаем новые
    mLow := 0;          // нижний предел отсутствует
    mHigh:= MaxInt;     // неограниченная пропускная способность
  end // with
end else begin
  // здесь обратной дуги нет, поэтому создаём новую дугу
  Reverse:= aDest.MakeLink(aSource, 0); // дуга
  Reverse.mHigh:= MaxInt;   // с максимальной пропускной способностью
end;

// Во вновь построенном вспомогательном графе
// вычисляем максимальный поток между вспомогательными истоком и стоком

Flow:= CalcMaxFlow0(Sx, Tx);

// После этого минимальный поток протекает через реверсную дугу:
Result:= Reverse.mFlow;  // запоминаем
Reverse.mFlow:= 0;       // и обнуляем

// Удаляем обратную дугу, или восстанавливаем предыдущую, если она была

// Обратная дуга существовала?
if Assigned(OldLink) then with OldLink do begin
  // да восстанавливаем
  mLow := OldLow;
  mHigh:= OldHigh;
end else begin
  // Дуга в исходном графе не существовала,
  // разрываем обратную связь между стоком и истоком
  aDest.RemoveLink(aSource);
end;

// Удаляем из графа и освобождаем:

RemoveNode(Sx);  Sx.Free;     // вспомогательный исток
RemoveNode(Tx);  Tx.Free;     // вспомогательный сток

// Если вспомогательный максимальный поток (Sum)
// равен сумме минимальных потоков (SumLow)
// то минимальный поток в исходном графе существует

if Flow = SumLow then begin
  // вспомогательные дуги насыщены
  RestoreFlowData(true); // формируем новые данные о потоках
end else begin
  // вспомогательные дуги НЕ насыщены
```

```
    Result:=-1;              // минимального потока не существует
    RestoreFlowData(false); // восстанавливаем прежние данные о потоках
  end;

end;


// Вычисление максимального потока
// когда минимальный поток может быть ненулевым

function TGraph.CalcMaxFlow(aSource, aDest: TNode): integer;
begin
  // Сначала вычисляем минимальный поток:
  Result:= CalcMinFlow(aSource, aDest);
  // Если он не существует, то не существует и максимального:
  if Result < 0 then Exit;
  // После формирования минимального потока формируем максимальный:
  Result:= CalcMaxFlow0(aSource, aDest);
end;


// Вычисление минимальной стоимости заданного потока
// Майника, стр. 104
// Кристофидес, стр. 339

function TGraph.CalcMinCostFlow(aSource, aDest: TNode; // источник и сток
                                aFlow: integer         // величина потока
                                ): integer;            // стоимость потока

var  Gray: TSet;       // множество серых (купленных) вершин
     Que: TBuffer;  // очередь вершин используется в CalcDeltaFlow
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка данных потока и вершинных чисел,
  // вызывается единожды в начале метода

  procedure ClearFlowData;
  var Node: TNode;
      Link: TLink;       // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
      Node.mDist:= 0;      // очищаем вершинные числа
      // Перебор исходящих линков вершины:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mFlow:= 0;           // текущий поток = 0
        Link.mColor:= CWhite;     // цвет дуги белый
        Link:= Node.OutLinkNext;
      end; // while
      Node:= NodeNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка цвета (mColor) и окраска вершины-истока
  // Вызывается в начале поиска очередной увеличивающей цепочки

  procedure ClearColors;
  var Node: TNode;       // текущая вершина
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
```

```
      Node.mColor:= CWhite;
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
      Node:= NodeNext;
    end;
    // Красим вершину-исток и помещаем в множество серых вершин:
    aSource.mColor:= CGray;
    Gray.Clear;
    Gray.Insert(aSource);
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Попытка приобретения очережной прямой дуги
  // Исходные данные:
  // - текущее множество серых вершин Gray
  // - текущий поток в дугах
  // - текущее состояние вершинных чисел
  // Возвращает true в случае покупки дуги изменением вершинных чисел

  function Buying: boolean;
  var Node: TNode;       // текущая вершина
      Link: TLink;       // текущий линк
      Delta: integer;    // очередное приращение суммы
      MinDelta: integer;// минимальное приращение суммы
  begin
    MinDelta:= MaxInt;
    // Перебор серых вершин:
    Node:= Gray.GetFirst as TNode;
    while Assigned(Node) do begin
      // Просмотр исходящих дуг:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do if mDest.mColor = CWhite then begin
          // Если поток не насыщен:
          if mFlow < mHigh then begin
            // Прицениваемся к увеличивающей дуге:
            // Link.mValue - цена увеличивающей дуги
            // mOwner.mDist - сумма оплаты цепочки вплоть до Node
            // mDest.mDist - частичная плата вплоть до mDest
            // Delta - то, что нужно доплатить за приобретение дуги Link
            Delta:= mValue - (mDest.mDist - mOwner.mDist);
            if Delta < MinDelta
              then MinDelta:= Delta;  // запоминаем минимальную доплату
          end; // if
        end;
        Link:= Node.OutLinkNext;
      end; // while
      Node:= Gray.GetNext as TNode; // Следующая вершина серого множества
    end; // while Assigned(Node)
    Result:= MinDelta < MaxInt;     // true, если дуга куплена
    if not Result or (MinDelta=0) then Exit;
    // Наращиваем mDist - вершинные числа белых вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mColor = CWhite then Inc(Node.mDist, MinDelta);
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск потока, проходящего по уже купленным дугам
  // Исходные данные:
  // - текущее множество серых вершин
  // - текущий поток
  // - текущие вершинные числа
```

```
// Расширяет множество серых вершин.
// Возвращает допустимое приращение потока (или ноль)
// и стоимость единицы потока

function CalcDeltaFlow(var aCost: integer): integer;
var
    Node: TNode;      // текущая вершина
    Link: TLink;      // исходящие и входящие дуги
begin
  Result:=0; aCost:=0;
  Que.Clear;  // очищаем очередь вершин
  // Ставим в очередь серые вершины
  Node:= Gray.GetFirst as TNode;
  while Assigned(Node) do begin
    Que.Put(Node);
    Node:= Gray.GetNext as TNode;
  end;
  Que.Put(aSource);
  // Обработка вершин из очереди:
  while Que.GetCount>0 do begin
    Node:= Que.Get as TNode;
    // Обработка исходящих дуг текущего узла:
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do if mDest.mColor=CWhite then begin
        // если вершина не помечена
        // Помечаем только белые вершины,
        // достигаемые через купленые ненасыщенные дуги
        // Link.mValue -- стоимость дуги
        // Link.mDest.mDist - mOwner.mDist -- разность вершинных чисел
        if (mValue = mDest.mDist - mOwner.mDist) // дуга куплена
           and (mFlow < mHigh) then begin  // и дуга не насыщена
          mDirect:= +1;  // отмечаем прямую дугу
          // помечаем концевую вершину и ставим в очередь:
          mDest.mColor:= CGray;     // цвет концевой вершины серый
          mDest.mPred:= Node;       // предшествующая вершина
          mDest.mLink:= Link;       // линк на предшествующую вершину
          mDest.mFlow:= Minimum(Node.mFlow, mHigh - mFlow); // поток
          Que.Put(mDest);       // в очередь приёмник дуги
          Gray.Insert(mDest);   // пополняем множество окрашенных
        end;
      end; // if
      Link:= Node.OutLinkNext;
    end;
    // Обработка входящих связей текущего узла:
    Link:= Node.InLinkFirst;
    while Assigned(Link) do begin
      with Link do if mOwner.mColor=CWhite then begin
        // Помечаем только белые вершины,
        // из которых поступает ненулевой поток по купленым дугам
        if mFlow > 0 then begin  // поток в дуге можно уменьшить
          mDirect:= -1;  // отмечаем встречную дугу
          // помечаем вершину и ставим в очередь:
          with Link.mOwner do begin
            mColor:= CGray;     // цвет исходной вершины серый
            mPred:= Node;       // предшествующая вершина
            mLink:= Link;       // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
          end;
          Que.Put(Link.mOwner);      // в очередь источник дуги
          Gray.Insert(Link.mOwner); // пополняем множество окрашенных
        end;
```

```
        end;
        Link:= Node.InLinkNext;
      end;  // while Assigned(Link)
      // Если помечен сток, то выход из цикла:
      if aDest.mColor <>  CWhite then Break;
    end;  // while Que.GetCount>0
    // Если сток помечен, то увеличивающая цепочка найдена
    // возвращаем допустимое увеличение потока и цену единицы потока
    if aDest.mColor <> CWhite then begin
      Result:= aDest.mFlow;
      aCost:= aDest.mDist;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Пересчёт потока вдоль предварительно найденной цепочки
  // aDelta - приращение потока

  procedure RecalcFlow(aDelta: integer);
  var Node: TNode;     // текущая вершина
  begin
    // Обратное движение от стока к истоку с корректировкой потока в дугах
    Node:= aDest;  // сток
    // Цикл, пока не достигнем истока:
    while Node <> aSource do begin
      // Node.mLink - линк из предшествующей вершины
      // Node.mLink.mDirect = +1 / -1
      with Node.mLink do Inc(mFlow, mDirect * aDelta); // поток
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Flow: integer;         // накопленный поток
  DeltaFlow: integer;    // приращение потока
  Cost1: integer;        // цена единицы потока
  OK: boolean;           // признак приобретения дуги

begin { TGraph.CalcMinCostFlow }

  Result:= 0;        // накопленная стоимость потока
  Flow:= 0;          // накопленный поток
  Gray:= CreateSet; // множество окрашенных (купленных) вершин
  Que:= TBuffer.Create;  // создаём очередь вершин
  ClearFlowData;     // очистка потока в дугах (mFlow=MaxInt)
  ClearColors;       // очистка цвета и начальная установка серого множества

  // Пока поток не достиг заданного значения:
  while Flow < aFlow do begin
    // Пытаемся "купить" увеличивающую цепочку (дугу)
    // и формируем вершинные числа.
    // Если куплена хотя бы одна дуга, пытаемся провести поток:
    repeat
      // Пытаемся провести поток и определить цену единицы потока
      DeltaFlow:= CalcDeltaFlow(Cost1);
      if DeltaFlow > 0  then begin
        OK:= true;
        // Поток возможен, ограничиваем его приращение:
        if DeltaFlow > aFlow - Flow then DeltaFlow:= aFlow - Flow;
        // Распределяем поток по дугам и и определяем цену единицы потока:
        RecalcFlow(DeltaFlow);         // пересчёт потока в дугах
        Inc(Flow, DeltaFlow);          // суммарный поток
        Inc(Result, Cost1*DeltaFlow); // суммарная стоимость потока
```

```
        ClearColors;                    // очистка цвета и серого множества
      end else begin
        // Если поток не проведен, пытаемся купить хотя бы одну дугу
        // (функция Buying модифицирует вершинные числа)
        OK:= Buying;
        // Если цепочку купить нельзя, то заданный поток не существует
        if not OK then Break;
      end;
      // пока поток не достигнет нужного, либо не найдено приращение
    until Flow = aFlow;
    // Если цепочку купить нельзя, то заданный поток не существует
    if not OK then begin
      Result:=-1;  // стоимость = -1
      Break;
    end;
  end; // while Flow < aFlow
  // Очистка памяти:
  Gray.Free; // серое множество
  Que.Free;  // рабочая очередь
end;


//#######################################################################
//                                                                      #
//                  П А Р О С О Ч Е Т А Н И Я                           #
//                                                                      #
//#######################################################################

{ Варианты формирования паросочетаний:

  TPairs = (pMaxN,  // максимальной мощности
            pMaxW   // максимального веса (стоимости)
            pMinW   // минимального веса (стоимости)
           );
}

type  // TPair -- Ребро представлено парой вершин

     TPair = class (TCostSet)
       function Compare(arg: TItem): TCompare;  override;
       procedure Print(var aFile: TextFile); override;
     end;

function TPair.Compare(arg: TItem): TCompare;
begin
  if Self = arg then begin Result:= cmpEq; Exit; end;
  if mCost = (arg as TCostSet).mCost then begin
    // При равной цене сравнением множества
    Result:= mSet.Compare((arg as TCostSet).mSet);
  end else begin
    // Если цены не совпали, сравнениваем по цене (дешёвые - в начало)
    if mCost > (arg as TCostSet).mCost
      then Result:= cmpGreate
      else Result:= cmpLess
  end;
end;

// Вывод множества вершин (у пары их будет две)

procedure TPair.Print(var aFile: TextFile);
var N: TNode;
    k: integer;
```

```
begin
  k:= mSet.GetCount;
  N:= mSet.GetFirst as TNode;
  while Assigned(N) do begin
    Write(aFile, N.GetName);
    Dec(k);
    if k > 0 then Write(aFile, '-');
    N:= mSet.GetNext as TNode;
  end;
  Write(aFile, ' = ', mCost);
end;

type  // TPairsBlock -- Блок, содержащий пары

      TPairsBlock = class (TSetList)
        mLabel : TNode;     // метка блока (центральная вершина)
        mCost  : integer;   // вес наиболее лёгкой пары в блоке
        function Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: Text); override;
        destructor Destroy;  override;
      end;

destructor TPairsBlock.Destroy;
begin
  ClrAndDestroy;  // очистка с уничтожением элементов
  Inherited;
end;

// Блоки сортируются в порядке неубывания размера блока

function TPairsBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;
  if GetCount > (arg as TPairsBlock).GetCount  then Result:= cmpGreate
end;

procedure TPairsBlock.Print(var aFile: Text);
begin
  Writeln(aFile,mLabel.GetName,' Cost=', mCost:3,'  Count=',GetCount:2);
  inherited Print(aFile);
  Writeln(aFile,'- - - - -');
end;

//#######################################################################
// Метод генерации паросочетаний (п/с) трёх видов:
// pMaxN -- максимальной мощности (без учёта веса рёбер)
// pMaxW -- максимального веса
// pMinW -- минимального веса
//#######################################################################

function TGraph.GenPairs_Dissect(aMode: TPairs): TCostSet;

var Blocks: TSet;         // Блоки пар (рёбер)
    BestCost: integer;    // Текущая лучшая (наименьшая) сумма
    BestCnt: integer;     // Текущая лучшая (наибольшая) мощность п/с
    Pairs: TCostSet;      // Текущее паросочетание
    Nodes: TSet;          // Текущее множество присоединённых вершин
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инициализация вершин и дуг
  procedure InitNodes;
  var Node: TNode;
      LD, LR : TLink; // прямой и обратный линки
```

```
begin
  // Предварительная очистка поля мощности
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mPower:= 0;  // здесь будут подсчитываться смежные линки
    Node:= NodeNext;
  end;
  // Перебор всех линков с целью отбора наименьших или наибольших
  // (в случае наличия встречных линков)
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      // Красим вершину белым
      mColor:= CWhite;
      // Перебор прямых линков LD
      LD:= OutLinkFirst;
      while Assigned(LD) do begin
        // Находим LR -- встречный линк
        LR:= LD.mDest.GetLink(Node);
        if Assigned(LR) then begin
          // Встречный линк найден
          if (aMode=pMaxW) and (LR.mValue < LD.mValue) then begin
            // При поиске максимума берём больший
            LD.mColor:= CBlack; // будет обрабатываться
            LR.mColor:= CWhite; // НЕ будет обрабатываться
          end else begin
            // При поиске минимума берём меньший
            LR.mColor:= CBlack; // будет обрабатываться
            LD.mColor:= CWhite; // НЕ будет обрабатываться
          end;
        end else begin
          // встречный линк НЕ найден
          LD.mColor:= CBlack; // будет обрабатываться прямой линк
        end;
        // Подсчёт инцидентных линков в двух соседних вершинах
        Inc(LD.mOwner.mPower);
        Inc(LD.mDest.mPower);
        LD:= OutLinkNext;
      end;
    end;
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск вершины, способной дать блок максимального размера

function GetBestNode: TNode;
var Node: TNode;
    Max : integer;
begin
  Max:= 0; Result:= nil;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do if (mColor=CWhite)     // вершина ещё не обработана
                    and (mPower > Max) // и её текущая мощность больше
    then begin
      Result:= Node;  // очередная лучшая вершина
      Max:= mPower;   // текущее число смежных чёрных линков
    end;
    Node:= NodeNext;
  end;
end;
```

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Формирование всех возможных пар вершин (рёбер)
// и группировка их в блоки

function GenBlocks: TSet;

  // Создание пары и вставка её в блок:
  function MakePair(aLink: TLink): TPair;
  begin
    Result:= TPair.CreateEmpty;
    with aLink do begin
      // В множество пары вставляем смежные с линком вершины:
      Result.mSet.Insert(mOwner);
      Result.mSet.Insert(mDest);
      // Уменьшаем мощности соседних вершин:
      Dec(mOwner.mPower);
      Dec(mDest.mPower);
    end;
    // Цену (вес) пары формируем в зависимости от варианта п/с
    case aMode of
      pMaxN: Result.mCost:=  1;              // наибольшее
      pMinW: Result.mCost:=  aLink.mValue;   // минимального веса
      pMaxW: Result.mCost:= -aLink.mValue;   // максимального веса ***
    end;
  end;

var Node: TNode;          // текущая вершина
    Link: TLink;          // текущий исходящий линк
    Pair: TPair;          // текущая пара
    Min: integer;         // наименьший вес пары в блоке
    Block: TPairsBlock;   // текущий блок

begin  { GenBlocks }
  InitNodes;            // инициализация полей mPower
  Result:= CreateSet;   // множество блоков
  // Перебор вершин:
  Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
  while Assigned(Node) do begin
    Min:= MaxInt;     // здесь формируется вес самой лёгкой пары в блоке
    Block:=nil;       // текущий блок пока не создан

    // Перебор исходящих линков:
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      // Если линк не обработан
      if Link.mColor = CBlack then begin
        Link.mColor:= CWhite;  // восстанавливаем цвет
        Pair:= MakePair(Link); // создаём пару
        // Запоминаем вес легчайшей пары в блоке
        if Min > Pair.mCost then Min:= Pair.mCost;
        // Если блок ещё не создан, то создаём:
        if not Assigned(Block) then Block:= TPairsBlock.Create;
        Block.Insert(Pair);  // вставляем пару в блок
      end;
      Link:= Node.OutLinkNext;  // следующий линк
    end; // while

    // Перебор входящих линков:
    Link:= Node.InLinkFirst;
    while Assigned(Link) do begin
      // Если линк не обработан
      if Link.mColor = CBlack then begin
```

```
          Link.mColor:= CWhite;  // восстанавливаем цвет
          Pair:= MakePair(Link); // создаём пару
          // Запоминаем вес легчайшей пары в блоке
          if Min > Pair.mCost then Min:= Pair.mCost;
          // Если блок ещё не создан, то создаём:
          if not Assigned(Block) then Block:= TPairsBlock.Create;
          Block.Insert(Pair);  // вставляем пару в блок
        end;
        Link:= Node.InLinkNext;  // следующий линк
      end; // while

      // Если блок создан, вставляем в множество результата:
      if Assigned(Block) then begin
        Block.mLabel:= Node;    // метка блока
        Block.mCost:= Min;      // вес легчайшей пары в блоке
        Result.Insert(Block);   // вставка блока
      end;
      Node.mColor:= CBlack; // отмечаем, что вершина обработана
      Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура обработки блока

  procedure BlockHandle(aBlock: integer;  // номер блока
                        aNodes: TSet;     // накопленные вершины
                        aRes: TCostSet    // накопленные пары
                        );
  var Block: TPairsBlock; // текущий блок
      Nodes: TSet;        // Накопленное множество вершин
      Res  : TCostSet;    // Накопленное множество пар (рёбер)
      Pair : TPair;       // очередная пара из блока

      // Прогноз результата

      function Prediction(aBlock: TPairsBlock;  // текущий блок
                          aNodes: TSet;         // накопленные вершины
                          aRes: TCostSet        // накопленные пары
                          ): boolean;
      var B: TPairsBlock;         // очередной блок
          Cost, Count : integer;  // прогнозируемые цена и мощность п/с
      begin
        Result:= not Assigned(aRes);
        if Result then Exit;
        // Начинаем накопление с текущей стоимости и мощности
        Cost:= aRes.mCost;  Count:= 0;
        if Assigned(aRes.mSet) then Count:= aRes.mSet.GetCount;
        // Пропускаем блоки вплоть до текущего
        B:= Blocks.GetFirst as TPairsBlock;
        while B <> aBlock do B:= Blocks.GetNext as TPairsBlock;
        // Переход к следующему блоку после текущего
        B:= Blocks.GetNext as TPairsBlock;
        // Перебор оставшихся блоков
        while Assigned(B) do begin
          if not aNodes.Exist(B.mLabel) then begin
            Inc(Count);          // накопление мощности паросочетания
            Inc(Cost, B.mCost); // накопление прогнозируемой стоимости
          end;
          B:= Blocks.GetNext as TPairsBlock;
        end;
        // Формируем положительный результат,
        // если выполняется одно из условий:
```

```
        // - есть шанс увеличить мощность паросочетания, или
        // - при той же мощности уменьшить стоимость паросочетания
        Result:= (Count > BestCnt) or
                 (Count = BestCnt) and (Cost < BestCost)
      end;
      var OK: boolean;
  begin  { BlockHandle }
    // Извлекаем текущий блок:
    Block:= Blocks.GetItem(aBlock) as TPairsBlock;
    // Если метка текущего блока НЕ содержится в накопителе вершин,
    // то перебираем пары текущего блока
    if not aNodes.Exist(Block.mLabel) then begin
      // Вершина, которою помечен блок, ещё не входит в паросочетание
      // Перебор пар текущего блока:
      Pair:= Block.GetFirst as TPair;
      while Assigned(Pair) do begin
        // Попытка прилепить очередную пару (ребро)
        // Пара присоединяется, если множество пары
        // не пересекается с текущим множ. чёрных вершин
        if  not aNodes.TestIntersect(Pair.mSet) then begin
          // Здесь пару можно присоединить
          Nodes:= aNodes.Copy as TSet;     // копия накопленных вершин
          Res:= aRes.Copy as TCostSet;     // копия накопленных пар
          // Пристраиваем к множеству пар очередную пару
          Nodes.Add(Pair.mSet);   // накапливаем множество вершин
          Res.Insert(Pair);       // накапливаем множество пар
          if aBlock < Blocks.GetCount then begin
            // Это не последний блок.
            // Если обработка последующих блоков может увеличить
            // мощность паросочетания или снизить цену,
            // то войти в следующий блок
            if Prediction(Block,Nodes,Res)
              then BlockHandle(aBlock+1,Nodes,Res);
          end else begin
            // Достигнут последний блок (aBlock = Blocks.GetCount),
            // Запоминаем результат, если выполняется одно из условий:
            // - текущий результат Result ещё не определён
            // - мощность нового п/с превышает мощность текущего
            // - мощности п/с равны, но стоимость нового меньше
            OK:= not Assigned(Result) or
               (Res.mSet.GetCount > BestCnt) or
               (Res.mSet.GetCount = BestCnt) and (BestCost > Res.mCost);
            if OK then begin
              BestCost:= Res.mCost;          // лучшая (меньшая) стоимость
              BestCnt:= Res.mSet.GetCount;   // текущая лучшая мощность п/с
              Result.Free;                   // удаляем прежнее паросочетание
              Result:= Res.Copy as TCostSet;// и сохраняем новое
            end;
          end;
          // Перед выбором следующей пары в блоке освобождаем текущие:
          Res.Free;   // накопленные пары
          Nodes.Free; // накопленные вершины
        end; // if
        Pair:= Block.GetNext as TPair;  // следующая пара в блоке
      end; // while
    end; // if
    // Если ищется наибольшее паросочетание без учёта веса,
    // и оно найдено, то выход
    if (aMode = pMaxN) and (BestCnt = Self.Nodes div 2) then Exit;
    // После перебора блока входим в следующий, если он существует
    if aBlock < Blocks.GetCount then begin
      // Текущий блок не последний.
```

```
      // Если обработка последующих блоков может увеличить
      // мощность паросочетания или снизить цену, то войти в следующий блок
      if Prediction(Block,aNodes,aRes) then
BlockHandle(aBlock+1,aNodes,aRes);
    end else begin
      // Достигнут последний блок (aBlock = Blocks.GetCount),
      // Запоминаем результат, если выполняется одно из условий:
      // - текущий результат Result ещё не определён
      // - мощность нового п/с превышает мощность текущего
      // - мощности п/с равны, но стоимость нового меньше
      if not Assigned(Result) or
         (aRes.mSet.GetCount > BestCnt) or
         (aRes.mSet.GetCount = BestCnt) and (BestCost > aRes.mCost)
        then begin
          BestCost:= aRes.mCost;          // лучшая (меньшая) стоимость
          BestCnt:= aRes.mSet.GetCount;   // текущая лучшая мощность п/с
          Result.Free;                    // удаляем прежнее паросочетание
          Result:= aRes.Copy as TCostSet;// и сохраняем новое
        end;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инверсия знаков весов рёбер паросочетания
  // (используется при поиске паросочетания максимального веса)

  procedure RestoreCost;
  var Pair : TPair;
      Buf: TBuffer;
  begin
    Buf:= TBuffer.Create;
    with Result.mSet do begin
      // Перебираем множество пар паросочетания Result.mSet
      Pair:= GetFirst as TPair;
      while Assigned(Pair) do begin
        Pair.mCost:= -Pair.mCost;
        Buf.Push(Pair);
        Pair:= GetNext as TPair;
      end;
      // Очищаем множество Result и повторно вставляем пары
      // для сортировки пар в порядке возрастания веса
      Result.Clear;
      while Buf.GetCount > 0 do begin
        Result.Insert(Buf.Pop as TPair);
      end;
    end;
    Buf.Free;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Перед уничтожением блоков удаляем из них найденные паросочетания

  procedure RemoveFromBlocks;
  var  Block: TPairsBlock;
  begin
    with Blocks do begin
      Block:= GetFirst as TPairsBlock;
      while Assigned(Block) do begin
        Block.Sub(Result.mSet);
        Block:= GetNext as TPairsBlock;
      end;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
begin { TGraph.GenPairs_Dissect }

  Result:= nil;        // Результат -- лучшее паросочетание

  Blocks:= GenBlocks;  // Создаём блоки
  if Blocks.GetCount = 0 then begin Blocks.Free; Exit; end;

  // Подготовка переменных:
  BestCost:= MaxInt;            // текущая лучшая сумма
  BestCnt:= 0;                  // текущая лучшая (наибольшая) мощность п/с
  Nodes:= CreateSet;            // множество вершин паросочетания
  Pairs:= TCostSet.CreateEmpty; // начальное пустое паросочетание

  // Рекурсивная обработка блоков:
  BlockHandle(1, Nodes, Pairs);  // формирует Result

  // Очистка памяти:
  Pairs.Free;        // Начальное паросочетание (пустое)
  Nodes.Free;        // Начальное множество присоединённых вершин
  // Удаляем из блоков множество Result.mSet
  if Assigned(Result) then  RemoveFromBlocks;
  Blocks.ClrAndDestroy;  // Удаляем сами блоки
  Blocks.Free;           // и множество блоков

  // После поиска паросочетания максимальной стоимости
  // восстанавливаем веса рёбер:
  if aMode = pMaxW then RestoreCost;
end;

//#######################################################################
//                                                                      #
//           Паросочетания в произвольном взвешенном графе              #
//                     методом увеличивающих цепей                      #
//                                                                      #
//#######################################################################

type  // Цветок для поиска паросочетаний

  TBlossom = class(TNodeInt)
    mBase: TNode;      // Первичная база цветка
    mNewBase: TNode;   // Новая (текущая) база цветка
    mNext: TNode;      // Парная вершина текущей базы
                       // указывает направление роспуска цветка
               // Проверяет наличие вершины в данном цветке
    function IsPresent(aNode: TNode): boolean;
               // Устанавливает новую базу цветка
               // и возвращает поле mNext
    function SetNewBase(aNewBase: TNode): TNode;
               // Возвращает корневую первичную базу цветка
    function  GetBase: TNode;
               // Возвращает корневую новую базу цветка
    function  GetNewBase: TNode;
               // Функция возвращает выгоду внутреннего пути
               // между простыми вершинами aIn и aOut
    function GetProfit(aIn, aOut : TNode): integer;
    function GetName: string; override;
    procedure Print(var aFile: TextFile); override;
  end;

{ TBlossom }
```

```
// Проверка наличия вершины aNode в данном цветке
// на любом уровне вложения

function TBlossom.IsPresent(aNode: TNode): boolean;
var N: TNode;
begin
  N:= aNode;
  repeat
    N:= N.mBlossom;      // Поле mBlossom
                         // указывает на непосредственный цветок
    Result:= N = Self;
    if Result then Break;
  until N = N.mBlossom; // Выход, если достигнут крайний цветок
end;

// Установка новой базы цветка aNewBase.
// Возвращает внутреннюю пару новой базы

function TBlossom.SetNewBase(aNewBase: TNode): TNode;
var Bloss: TBlossom;
begin
  // Берём непосредственный цветок новой базы
  Bloss:= aNewBase.mBlossom as TBlossom;
  if aNewBase <> mNewBase
    then Bloss.mNewBase:= aNewBase;  // Новая база цветка
  Result:= aNewBase.GetPair;  // Внутренняя пара новой базы
  // Выход, если это крайний наружный цветок
  if Bloss.mBlossom = Bloss then Exit;
  // А иначе устанавливаем базу внешнего цветка,
  // двигаясь рекурсивно изнутри цветка наружу
  Result:= (Bloss.mBlossom as TBlossom).SetNewBase(Bloss);
end;

// Возвращает корневую первичную базу цветка
// Корневая база является простой вершиной

function  TBlossom.GetBase: TNode;
begin
  // Двигаемся рекурсивно от наружного цветка внутрь
  if mBase is TBlossom
    then Result:= (mBase as TBlossom).GetBase
    else Result:= mBase;  // здесь достигли простой вершины
end;

// Возвращает корневую новую базу цветка
// Корневая база является простой вершиной

function  TBlossom.GetNewBase: TNode;
begin
  // Двигаемся рекурсивно от наружного цветка внутрь
  if mNewBase is TBlossom
    then Result:= (mNewBase as TBlossom).GetNewBase
    else Result:= mNewBase;  // здесь достигли простой вершины
end;

// Функция возвращает выгоду (профит) внутреннего пути
// между двумя вершинами:
// aIn  -- входная вершина (текущая база цветка)
// aOut -- выходная вершина (в потенциале новая база)
// Профит = плюс веса непарных рёбер, минус веса парных

function TBlossom.GetProfit(aIn, aOut : TNode): integer;
```

```
  // Локальная функция вычисляет профит вершины aNode
  // внутри цветка aBloss с учётом вложенности цветков

  function CalcProfit(aBloss: TBlossom; aNode: TNode): integer;
  var Bloss: TBlossom;
  begin
    // Вначале берём профит вершины
    Result:= aNode.mProfit;
    // Выход, если достигнут данный цветок
    if aBloss = Self then Exit;
    // А иначе берём Bloss -- цветок следующего уровня
    Bloss:= aBloss.mBlossom as TBlossom;
    // Выполняем рекурсию изнутри цветка наружу,
    // суммируя профиты вложенных цветков
    Result:= Result + CalcProfit(Bloss, aBloss);
  end;

var Bloss: TBlossom;       // непосредственный цветок вершины
    Profit_In: integer;    // профит входной вершины
    Profit_Out: integer;   // профит выходной вершины
begin
  // Берём непосредственный цветок входной вершины
  Bloss:= aIn.mBlossom as TBlossom;
  // Вычисляем профит для этой вершины
  Profit_In:= CalcProfit(Bloss, aIn);
  // Берём непосредственный цветок выходной вершины
  Bloss:= aOut.mBlossom as TBlossom;
  // Вычисляем профит для этой вершины
  Profit_Out:= CalcProfit(Bloss, aOut);
  // Результат является разностью профитов
  Result:= Profit_Out - Profit_In;
end;

// Формирование строки для отображения цветка

function TBlossom.GetName: string;

  function Short(const arg: string): string;
  begin
    Result:= System.Copy(arg,1,3)
  end;

var N: TNode;
    Flag: boolean;
begin
  Result:= inherited GetName+': ';
  Result:= Result+' {';
  N:= mBase;  Flag:= false;
  repeat
    if Flag
      then Result:= Result+Short(N.GetName)+'='
      else Result:= Result+Short(N.GetName)+'-';
    N:= N.mRight;  Flag:= not Flag;
  until N = mBase;

  if (mBase <> mNewBase) and Assigned(mNewBase) and (mNewBase <> Self)
    then Result:= Result+' <'+Short(mNewBase.GetName)+'>';

  if Assigned(mPair) then with mPair do
    Result:= Result+' '+mOwner.GetName+'+'+mDest.GetName;
```

```
  if mValue <> 0 then if mOut
    then Result:= Result+' ->> '+IntToStr(mValue)
    else Result:= Result+' --> '+IntToStr(mValue);

  Result:= Result+' : '+IntToStr(mProfit);
  Result:= Result+'}';
end;

procedure TBlossom.Print(var aFile: TextFile);
begin
  Writeln(aFile, GetName);
end;

//#######################################################################
// Вспомогательные функции

// Функция возвращает статус крайнего охватывающего
// вершину наружного цветка
// (true -- внешний, false -- внутренний)

function  GetOut(aNode: TNode): boolean;
begin
  Result:= aNode.GetBlossom.mOut;
end;

// Функция возвращает корень текущего дерева,
// к которому принадлежит крайний цветок вершины

function  GetRoot(aNode: TNode): TNode;
begin
  Result:= aNode.GetBlossom.mRoot as TNode;
end;

//#######################################################################
//                                                                      #
//           Паросочетания в произвольном взвешенном графе              #
//                       алгоритмом Эдмондса                            #
//                                                                      #
//#######################################################################
// Параметры aMode: TPairs  -- искомое паросочетание:
//    pMaxN  -- максимальное без учёта веса рёбер
//    pMaxW  -- совершенное максимального веса
//    pMinW  -- совершенное минимального веса

function TGraph.GenPairs_Edmonds(aMode: TPairs): TCostSet;

var // Глобальные переменные метода

  gMainQue : TBuffer;      // Главная очередь вершин
  gQue : TBuffer;          // Очередь вершин для текущего дерева
  gBlossomStack : TBuffer;// Стек для хранения цветков
  gBlosNum: integer;       // Уникальный идентификатор цветка
  gAppendix: TSet;         // Множество добавляемых в очередь вершин
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Предварительные объявления

  // Роспуск цветка
  procedure BlossExpand(aBloss: TBlossom); forward;

  // Удаление цветков
  procedure RemoveBlossoms; forward;
```

```
// Разрыв пары aNode <-> aNode.mPair
procedure Disconnect(aNode, aBloss : TNode); forward;

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск ребра максимального веса
// и установка начальных значений вершинных чисел (полей mValue)

procedure InitGraphValues;
var N: TNode;    // Текущая вершина
    L: TLink;    // Текущее ребро
    MaxVal: integer;  // Максимальный вес ребра
begin
  MaxVal:=0;
  // Поиск ребра максимального веса и сохранение весов рёбер
  N:= NodeFirst;
  while Assigned(N) do begin
    // Перебор рёбер текущей вершины
    L:= N.OutLinkFirst;
    while Assigned(L) do with L do begin
      if mValue > MaxVal then MaxVal:= mValue;
      mTemp:= mValue;      // сохраняем исходный вес ребра
      L:= N.OutLinkNext;   // следующее ребро
    end;
    N:= NodeNext; // следующая вершина
  end;
  MaxVal:= (MaxVal+1) div 2;
  // Установка начальных значений вершинных чисел
  N:= NodeFirst;
  while Assigned(N) do begin
    N.mLimit:= N.mValue; // сохраняем исходный вес вершины
    N.mValue:= MaxVal;
    N:= NodeNext;
  end;

  if aMode = pMinW then begin
    // При поиске паросочетаний минимального веса
    // инвертируем веса рёбер
    N:= NodeFirst;
    while Assigned(N) do begin
      // Перебор рёбер текущей вершины
      L:= N.OutLinkFirst;
      while Assigned(L) do with L do begin
        // Инвертируем веса рёбер
        // с добавлением положительного смещения
        mValue:= 2*MaxVal - mValue;
        L:= N.OutLinkNext;
      end;
      N:= NodeNext;
    end;
  end;
end;

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Восстановление исходных вершинных чисел и рёберных весов

procedure RestoreGraphValues;
var N: TNode;    // Текущая вершина
    L: TLink;    // Текущее ребро
begin
  N:= NodeFirst;
  while Assigned(N) do begin
    N.mValue:= N.mLimit;  // восстанавливаем поле mValue
```

```
      L:= N.OutLinkFirst;
      while Assigned(L) do with L do begin
        mValue:= mTemp;    // восстанавливаем поле mValue
        L:= N.OutLinkNext;
      end;
      N:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Установка в очередь aQue вершины или цветка aNode

  procedure PutInQue(aQue: TBuffer; aNode : TNode);
  var N: TNode;
  begin
    if aNode is TBlossom then with aNode as TBlossom do begin
      // Если это цветок, вызываем PutInQue рекурсивно
      N:= mBase;
      repeat
        PutInQue(aQue, N);
        N:= N.mRight;
      until N = mBase;
    end else begin
      // Если простая вершина, ставим в очередь
      aQue.Put(aNode);
    end;
  end;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Создание цветка на базе замыкающего его линка aLink

  function BlossomMake(aLink: TLink): TBlossom;

  var Base : TNode; // База создаваемого цветка

    // Локальная функция вычисляет профит вершины aNode
    // относительно корневой базы цветка

    function CalcProfit(aNode: TNode): integer;
    var Direct: boolean;  // Направление обхода цветка:
                          // true - обход по часовой стрелке
        Sign: boolean;    // Текущий знак ребра:
                          // true - парное, false - непарное
        N: TNode;         // Простая вершина внутри вложенного цветка
        N1, N2: TNode;    // Пара соседних вершин
        Link: TLink;      // Текущий линк
        Delta: integer;   // Очередное приращение результата
    begin
      Result:= 0;
      if aNode = Base then Exit;  // Профит базовой вершины = 0
      // Если aNode является вложенным цветком...
      if aNode is TBlossom then with aNode as TBlossom do begin
        // то вычисляем профит от переключения
        // базы с aNode.mNewBase на aNode.mBase
        N1:= GetNewBase;  // новая корневая база цветка
        N2:= GetBase;     // исходная корневая база цветка
        Result:= GetProfit(N1,N2);
      end;
      // Определяем направление дальнейшего обхода цветка
      // от вершины aNode к исходной базе цветка Base
      // (true - обход по часовой стрелке)
      Direct:= aNode.mPair = aNode.mRLnk;
      N1:= aNode;   // Начало обхода в напралении парного ребра
```

```
      Sign:= true;  // true - парное ребро, false - непарное
      repeat
        if Direct then begin
          N2:= N1.mRight;      // По часовой
          Link:= N1.mRLnk;     // Линк связывает N1 <-> N2
        end else begin
          N2:= N1.mLeft;       // Против часовой
          Link:= N2.mRLnk;     // Линк связывает N1 <-> N2
        end;
        Delta:= Link.mValue;  // Вес текущего линка (ребра)
        // Если это парное ребро, то берём с минусом
        if Sign then Delta:= -Delta;
        // Накапливаем результат
        Inc(Result, Delta);
        // Для непарного ребра исследуем
        // соединяемые им вершины N1 <--> N2
        // Если они являются вложенными цветками,
        // то добавляем профиты от перемещения их баз
        if not Sign then begin
          // Здесь Link -- не парное ребро
          if N1 is TBlossom then with N1 as TBlossom do begin
            // Здесь N1 является вложенным цветком
            // Выявляем N -- простую вершину внутри цветка N1
            if IsPresent(Link.mOwner)
              then N:= Link.mOwner else N:= Link.mDest;
            // Вычисляем профит (with N1 as TBlossom)
            Delta:= GetProfit(GetNewBase, N);
            // Накапливаем результат
            Inc(Result, Delta);
          end;

          if N2 is TBlossom then with N2 as TBlossom do begin
            // Здесь N2 является вложенным цветком
            // Выявляем N -- простую вершину внутри цветка N2
            if IsPresent(Link.mOwner)
              then N:= Link.mOwner else N:= Link.mDest;
            // Вычисляем профит (with N2 as TBlossom)
            // в зависимости от того, является ли вершина N2
            // базой, либо промежуточной на пути к базе
            if N2 = Base
              then Delta:= GetProfit(GetBase, N)
              else Delta:= GetProfit(GetNewBase, N);
            // Накапливаем результат
            Inc(Result, Delta);
          end;
        end; // if not Sign
        // Переход к следующему ребру
        Sign:= not Sign;
        N1:= N2;
      until N1 = Base;
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  var Owner, Dest : TNode; // Обобщённые вершины замыкающего линка
      N1, N2  : TNode;     // Пара смежных вершин
      N  : TNode;

  begin  // function BlossomMake

    // Переменная gBlosNum нужна для сквозной нумерации цветков
    Inc(gBlosNum);
    // Создаём новый цветок с текущим номером
```

```
    Result:= TBlossom.Create(gBlosNum, 0, nil);
    // Выявляем обобщённые вершины замыкающего линка
    with aLink do begin
      Owner:= mOwner.GetBlossom;
      Dest:= mDest.GetBlossom;
    end;

    // Расстановка ссылок mLink для вершин замыкающего линка

    Owner.mLink:= aLink;
    Dest.mLink:= aLink;
    Dest.mPred:= Owner;

    // Пометка вершин от концов линка
    // в сторону корня текущего дерева
    // с целью поиска базы цветка.
    // Для пометок вершин используем поле mColor

    N:= Owner; // От начала линка (внешняя вершина)
    repeat
      N.mColor:= gBlosNum;  // Метим уникальным номером цветка
      if GetOut(N) then begin
        // Поскольку это внешняя вершина,
        // то переходим к парной
        N:= N.GetPair;
        if Assigned(N) then N:= N.GetBlossom;
      end else begin
        // Поскольку это внутренняя вершина,
        // то переходим к предшествующей
        N:= N.mPred;
        if Assigned(N) then N:= N.GetBlossom;
      end;
    until not Assigned(N);

    // Здесь достигли корня текущего дерева.
    // Теперь выполняем проход от конца замыкающего линка
    N:= Dest; // от конца линка (внешняя вершина)
    while N.mColor <> gBlosNum do begin
      // Повторяем, пока не достигнем помеченной вершины
      N:= N.GetPair.GetBlossom; // парная вершина
      N:= N.mPred.GetBlossom;   // предыдущая парной вершине
    end;

    // Здесь N указывает на базу цветка
    // которая сама может являться цветком
    Base:= N;

    // Заполняем поля созданного цветка

    with Result do begin
      mOut:= true;           // изначально внешняя вершина
      mBase:= Base;          // первичная база
      mNewBase:= Base;       // текущая (новая) база
      mPair:= Base.mPair;    // парная вершина цветка
      mOwnDest:= Base.mOwnDest; // Положение в паре (Owner|Dest)
      mDist:= Base.mDist;    // профит цветка
    end;

    // Расстановка ссылок mRight, mLeft и mRLnk
    // для обобщённых вершин замыкающего линка

    Owner.mRight:= Dest;  // ссылка вправо
```

```
    Dest.mLeft:= Owner;   // ссылка влево
    Owner.mRLnk:= aLink;  // линк вправо

    // Расстановка в цветке ссылок
    // по часовой N.mRight и против часовой N.mLeft
    // путём двукратного прохода от концов замыкющнго ребра aLink
    // в направлении базы цветка.

    // Функция GetOut не учитывает созданный здесь цветок,
    // поскольку вершины цветка пока не ссылаются на него.

    // От конца замыкающего линка движемся к базе
    N2:= Dest;  // обобщённая вершина
    while N2 <> Base do begin
      if GetOut(N2) then begin
        // N1 -- внутренняя, N2 - внешняя
        N1:= N2.GetPair.GetBlossom;
        N2.mRLnk:= N2.mPair;  // берём парный линк
      end else begin
        // N1 -- внешняя, N2 -- внутренняя
        N1:= N2.mPred.GetBlossom;
        N2.mRLnk:= N2.mLink;  // берём линк на предыдущую вершину
      end;
      N2.mRight:= N1;   N1.mLeft:= N2;
      N2:= N1;
    end;

    // От начала замыкающего линка движемся к базе
    N2:= Owner;  // обобщённая вершина
    while N2 <> Base do begin
      if GetOut(N2) then begin
        // N1 -- внутренняя, N2 - внешняя
        N1:= N2.GetPair.GetBlossom;
        N1.mRLnk:= N2.mPair;  // берём парный линк
      end else begin
        // N1 -- внешняя, N2 -- внутренняя
        N1:= N2.mPred.GetBlossom;
        N1.mRLnk:= N2.mLink;  // берём линк на предыдущую вершину
      end;
      N2.mLeft:= N1;   N1.mRight:= N2;
      N2:= N1;
    end;

    // Установка в расширяющую дерево очередь gQue
    // внутренних вершин цветка.
    // Обходим цветок по часовой стрелке
    N:= Base;
    repeat
      if not GetOut(N) then begin
        // Здесь N -- это внутренняя вершина,
        // ставим в очередь для расширения дерева
        PutInQue(gQue, N);
      end;
      // метим вершину текущим цветком
      // связывая тем самым с созданным цветком
      N.mBlossom:= Result;
      N:= N.mRight;
    until  N= Base;

    // Обход цветка по часовой стрелке
    // для вычисления полей mProfit
    // -- профитов относительно базы внутри цветка
```

```
    N:= Base;
    repeat
      N.mProfit:= CalcProfit(N);
      N:= N.mRight;
    until N= Base;

  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Функция вычисляет дисбалланс линка aLink.
  // Линк сбаллансирован (доступен) если Delta = 0.
  // При поиске паросочетания без учёта веса рёбер (aMode = pMaxN),
  // т.е. для не взвешенного графа, всегда возвращается ноль.

  function Delta(aLink : TLink): integer;
  begin
    if aMode = pMaxN
      then Result:=0  // Если вес рёбер не учитывается
      else with aLink do
        // А иначе берём сумму весов вершин
        // и вычитаем вес ребра
        Result:= mOwner.mValue + mDest.mValue - mValue;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск альтернирующей цепи с корнем в aRoot.
  // Возможные результаты:
  // -- Возвращает белую вершину, если найдена сильная увелич. цепь.
  // -- Возвращает чёрную вершину, если найдена слабая увелич. цепь.
  // -- Возвращает nil, если обнаружено венгерское дерево.

  function Alter(aRoot: TNode): TNode;

    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Умножение графа на 2 (масштабирование)
    // Вызывается при получении дробного значения поправки,
    // расширяющей текущее дерево

    procedure Mul_2;
    var Node: TNode;
        L : TLink;
    begin
      PosPush;  // Сохранить позицию перебора
      Node:= NodeFirst;
      while Assigned(Node) do begin
        with Node do begin
          mValue:= 2*mValue;
          mDist:= 2*mDist;
          mProfit:= 2*mProfit;
          // Обработка исходящих линков
          L:= OutLinkFirst;
          while Assigned(L) do begin
            L.mValue:= 2*L.mValue;
            L:= OutLinkNext;
          end;
        end;
        Node:= NodeNext;
      end;
      PosPop;  // восстановить позицию перебора

      // Обработка цветков, хранящихся в стеке gBlossomStack
      Node:= gBlossomStack.GetFirst as TNode;
      while Assigned(Node) do begin
        with Node as TBlossom do begin
```

```
        mValue:= 2*mValue;     // Текущий вес цветка
        mDist:= 2*mDist;       // Линейный профит
        mProfit:= 2*mProfit;   // Профит в цветке
      end;
      Node:= gBlossomStack.GetNext as TNode;
    end;
  end;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Получение минимального значения весовой поправки
  // путём анализа весов (поправок) внутренних цветков

  function GetMinBlossValue(aRoot : TNode): integer;
  var Bloss: TBlossom;
  begin
    Result:= MaxInt;
    // Перебор всех цветков в стеке
    Bloss:= gBlossomStack.GetFirst as TBlossom;
    while Assigned(Bloss) do begin
      with Bloss do begin
        if (GetRoot(Bloss) = aRoot) // если цветок в текущем дереве
           and not GetOut(Bloss)    // и помечен как внутреняя вершина
           then begin
          // Это внутренняя вершина, выбираем минимальный вес
          if Result > mValue then Result:= mValue
        end;
      end;
      Bloss:= gBlossomStack.GetNext as TBlossom;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вычисление минимального приращения,
  // необходимого для расширения текущего дерева
  // aSuccess -- возвращает признак возможности расширить дерево

  function CalcForExpand(var aSuccess: boolean): integer;
  var Node: TNode;       // текущая вершина
      Blossom : TNode;   // цветок этой вершины
      L : TLink;         // текущий линк
      d : integer;       // баланс ребра
      Res: extended;     // Промежуточный результат (дробный)

  begin
    aSuccess:= false; // Признак возможности расширить дерево
    gAppendix.Clear;  // Очищаем множество добавляемых в очередь вершин

    // Находим минимальную поправку для внутр. цветка
    Result:= GetMinBlossValue(aRoot);
    // Если Result <> MaxInt, то начинаем с этого значения
    if Result <> MaxInt then begin
      Result:= Result div 2;
      aSuccess:= true;  // признак возможности расширить дерево
      if Result = 0
        then Exit;  // будет распущен хотя бы один внутр. цветок
    end;
    // Если поправка для внутр. цветков не нулевая,
    // начинаем с этого её текущего значения
    Res:= Result; // Res : extended;

    // Начало перебора вершин текущего дерева
    Node:= NodeFirst;
    while Assigned(Node) do with Node do begin
```

```
      // Исследуем только внешние вершины текущего дерева
      if (GetRoot(Node)<>aRoot) // Если вершина не в текущем дереве
         or not GetOut(Node)    // или это внутренняя вершина
         then begin             // то пропускаем её
          Node:= NodeNext as TNode;
          Continue;
      end;
      // Определяем цветок, в котором состоит данная вершина
      Blossom:= Node.GetBlossom;
      // Перебор соседей очередной внешней вершины Node (with Node)
      L:= OutLinkFirst;
      while Assigned(L) do with L do begin
        // Смежная вершина принадлежит текущему дереву?
        if GetRoot(mDest) = aRoot then begin
          // Здесь смежная вершина принадлежит текущему дереву
          if (mDest.GetBlossom = Blossom)
            // Если смежная вершина расположена внутри того же цветка...
            or not GetOut(mDest) // или ведёт во внутреннюю вершину дерева
            then begin
            // то пропускаем этот линк и переходим к следующему
            L:= OutLinkNext;
            Continue;
          end;
          // Здесь смежная вершина не принадлежит тому же цветку
          // и является внешней,
          // следовательно ребро L может замкнуть новый цветок
          // Вычисляем баланс ребра (with L)
          d:= mOwner.mValue + mDest.mValue - mValue;
          // Если этот баланс d меньше текущего результата Res
          if d/2 < Res {текущий} then begin
            // то очищаем текущее расширяющее множество
            // Res может быть дробным!
            Res:= d/2;  // обновляем текущий результат
            gAppendix.Clear;  // Очищаем множество очередных вершин
            aSuccess:= true;  // Признак возможности расширить дерево
          end;
          // Если вновь полученный результат
          // лучше или равен текущему...
          if Round(2*Res) {текущий} = d {новый}
            // то добавляем вершину к множеству
             then gAppendix.Insert(Node);
        end else begin
          // Здесь смежная вершина НЕ принадлежит текущему дереву.
          // То есть, линк L ведёт за пределы дерева,
          // он может расширить дерево
          // Вычисляем баланс ребра (with L)
          d:= mOwner.mValue + mDest.mValue - mValue;
          // Если этот баланс d меньше текущего результата Res
          if d < Res {текущий} then begin
            Res:= d;           // обновляем текущий результат
            gAppendix.Clear;   // Очищаем множество очередных вершин
            aSuccess:= true;   // признак возможности расширить дерево
          end;
          // Если вновь полученный результат
          // лучше или равен текущему...
          if Round(Res) = d
            // то добавляем вершину к множеству очередных вершин
            then gAppendix.Insert(Node);
        end;  // if GetRoot(mDest) = aRoot
        L:= OutLinkNext;  // Берём следующее ребро вершины Node
      end; // while Assigned(L)
      // Переход к следующей вершине текущего дерева
```

```
      Node:= NodeNext;
    end; // while Assigned(Node)

    // По окончании перебора вершин проверяем дробность результата
    if (Round(2*Res) mod 2) <>0 then begin
      // Здесь результат (поправка) дробный
      Res:= Round(2*Res); // Умножаем его на 2
      Mul_2; // и масштабируем дерево умножением на 2
    end;
    // Формируем окончательный результат
    // как минимальное возможное приращение
    // Здесь: Result -- целое, Res -- Real
    if Result > Round(Res) then Result:= Round(Res);
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  // Изменение весов вершин текущего дерева на величину поправки aDelta
  // (вычисленного функцией CalcForExpand).
  // Возвращает TRUE, если был распущен хотя бы один цветок

  function ChangeValues(aDelta: integer): boolean;
  var Node: TNode;
      Count: integer;
  begin
    Result:= false;

    // Перебор вершин текущего дерева
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Если вершина в текущем дереве
      if (GetRoot(Node) = aRoot) then begin
        // меняем её вес
        if GetOut(Node)
          then Dec(Node.mValue, aDelta)    // Node внешняя -
          else Inc(Node.mValue, aDelta);   // Node внутренняя +
      end;
      Node:= NodeNext;
    end; // while

    // Обработка цветков, хранящихся в стеке gBlossomStack
    Node:= gBlossomStack.GetFirst as TNode;
    while Assigned(Node) do begin
      with (Node as TBlossom) do begin
        if (GetRoot(Node) = aRoot)  // Если цветок в текущем дереве
          // то меняем его вес
          then if GetOut(Node)
            then Inc(mValue, 2*aDelta)  // внешний + 2*aDelta
            else Dec(mValue, 2*aDelta); // внутренний - 2*aDelta
      end; // with
      Node:= gBlossomStack.GetNext as TNode;
    end; // while

    // Повторная обработка цветков в стеке gBlossomStack
    // Будут распущены внутренние цветки текущего дерева,
    // вес которых составляет ноль.
    Count:= gBlossomStack.GetCount;
    while Count > 0 do begin
      Dec(Count);
      // Берём цветок с вершины стека
      Node:= gBlossomStack.Get as TNode;
      with (Node as TBlossom) do begin
        if GetRoot(Node) = aRoot then begin
```

```
          // Здесь цветок в текущем дереве
          if not GetOut(Node) // Если он внутренний
             and (mValue = 0) {и обнулён} then begin
            // Здесь вершинное число обнулилось,
            // распускаем цветок
            Result:= true;  // Признак роспуска цветка
            BlossExpand(Node as TBlossom);
            Node.Free;      // Удаляем из памяти
            Continue;       // В стек не возвращаем
          end; // if
        end; // if
      end; // with Node as TBlossom
      gBlossomStack.Put(Node);  // Возвражаем в стек (очередь)
    end; // while
  end; // function

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Сброс полей mOut всех вершин в состояние false (внутренние)
// и удаление их из текущего дерева.
// Вызывается перед построением очередного дерева.

procedure LocalReset;
var Node : TNode;
begin
  PosPush;  // сохранить позицию перебора
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mBlossom = Node then with Node do begin
      // Эта вершина не содержится в цветках
      mOut:= false;
      mRoot:= nil;
    end;
    Node.mDist:= 0;
    Node:= NodeNext;
  end;
  PosPop;  // восстановить позицию перебора

  // Перебор цветков
  Node:= gBlossomStack.GetFirst as TNode;
  while Assigned(Node) do begin
    with Node do begin
      mOut:= false;
      mRoot:= nil;
      mDist:= 0;
    end;
    Node:= gBlossomStack.GetNext as TNode;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Обработка очередной внешней вершины aNode
// в ходе построения альтернирующего дерева
// с корнем в вершине aRoot

function NodeHandle(aNode: TNode): TNode;
var Dest : TNode;       // Смежная вершина
    Pair : TNode;       // Пара для Dest
    NodeBloss : TNode; // Цветок для aNode
    DestBloss : TNode; // Цветок для Dest
    PairBloss : TNode; // Цветок для Pair
    Bloss : TNode;      // Вновь созданный цветок
    Base : TNode;       // База цветка NodeBloss
    Node : TNode;       // Простая вершина внутри цветка
```

```
      Link : TLink;       // Очередной линк к смежной вершине
      Profit : integer;   // Профит внутри цветка
      d : integer;        // Баланс ребра Link
  begin
    Result:= nil; // Результат на случай безуспешного поиска
    // Выявляем крайний цветок, внутри которого
    // расположена вершина aNode (если цветок существует)
    NodeBloss:= aNode.GetBlossom;
    // Это в самом деле цветок?
    if NodeBloss is TBlossom then with NodeBloss as TBlossom do begin
      // Здесь NodeBloss -- это цветок
      // Этот цветок имеет пару?
      if Assigned(mPair) then begin
        // Цветок имеет пару, ищем текущую (новую) базу цветка,
        // проверяя концы парного ребра mPair
        if IsPresent(mPair.mOwner)
          then Base:= mPair.mOwner // mOwner в цветка
          else Base:= mPair.mDest; // mDest в цветке
      end else begin
        // Здесь цветок не имеет пары
        Base:= GetNewBase;   // Берём его текущую базу
      end;
      // Вычисляем цветочный профит в связи с возможным переходом
      // базы из вершины Base в вершину aNode
      Profit:= GetProfit(Base, aNode);
      // И формируем линейный профит вершины
      aNode.mDist:= Base.mDist + Profit;
    end; // if NodeBloss is TBlossom

    // Проверяем aNode на слабую увеличивающую цепь
    if (aNode.mDist > 0) and Assigned(NodeBloss.mPair) then begin
      // Здесь найдена слабая увеличивающая цепь
      Result:= aNode; // Возвращаем текущую вершину
      Exit;           // и выходим из функции
    end;

    // Здесь aNode не замыкает слабую увеличивающую цепь,
    // исследуем окрестности вершины aNode,
    // результатом может быть:
    // a) обнаружение непарной (белой) вершины
    // b) обнаружение парных вершин, расширяющих дерево
    // c) обнаружение парных вершин, создающих цветок
    // d) пустой результат

    // Перебор соседних вершин
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      // Вычисляем балланс ребра (доступность ребра)
      d:= Delta(Link);
      if d <> 0 then begin
        // Ребро L недоступно, пропускаем его
        Link:= aNode.OutLinkNext; Continue;
      end;
      // Здесь d = 0, поэтому ребро L доступно
      Dest:= Link.mDest;  // Берём целевую вершину Dest
      // Целевая вершина Link.mDest
      // может принадлежать цветку DestBloss
      DestBloss:= Dest.GetBlossom;
      if DestBloss.GetPair <> aNode then
        // Здесь DestBloss не является парой
        // для текущей вершины aNode
        // Dest принадлежат текущему дереву?
```

```
        if GetRoot(Dest) <> aRoot then begin
          // Здесь вершина Dest не принадлежат текущему дереву
          // Вычисляем линейный профит (mDist) к целевой вершине
          DestBloss.mDist:= aNode.mDist + Link.mValue;
          with DestBloss do begin
            // Заполняем поля целевой вершины (или её цветка)
            mRoot:= aRoot;  // включаем в текущее дерево
            mOut:= false;   // метим как внутреннюю
            mPred:= NodeBloss; // предыдущая вершина в цепи
            mLink:= Link;   // входящий линк вершины или цветка
            // Переходим к паре внутренней вершины
            Pair:= GetPair; // DestBloss.GetPair;
          end;
          // Если пара у DestBloss отсутствует,
          // значит найдена непокрытая вершина,
          // либо цветок, её содержащий
          if not Assigned(Pair) then begin
            // здесь обнаружена либо непокрытая вершина L.mDest,
            // либо цветок, содержащий белую вершину
            Result:= DestBloss;       // возвращаем эту вершину
            Result.mPred:= NodeBloss; // предшествующая вершина
            Result.mLink:= Link;      // фиксация входящего линка
            Break;                    // выход из функции
          end // if not Assigned(Pair)
          else begin
            // Здесь существует пара Pair (DestBloss.mPair)
            // Берём её цветок (или саму простую вершину)
            PairBloss:= Pair.GetBlossom;
            with PairBloss do begin
              // Заполняем поля вершины или цветка
              mRoot:= aRoot;  // включаем в текущее дерево
              mOut:= true;    // метим как внешнюю
              Link:= mPair;   // парный линк Dest <=> Pair
              // Вычисляем линейный профит к парной вершине
              // или базе цветка (вычитаем вес парного линка)
              Pair.mDist:= DestBloss.mDist - Link.mValue;
              mDist:= Pair.mDist; // PairBloss.mDist
            end; // with PairBloss

            // Если смежная внутренняя вершина является цветком...
            if DestBloss is TBlossom then
              with DestBloss as TBlossom do begin
              // Находим в цветке простую парную вершину Node
              // и добавляем к пути Pair.mDist её профит
              // GetProfit(Node, Dest)
              if IsPresent(Link.mOwner)
                then Node:= Link.mOwner
                else Node:= Link.mDest;
              Profit:= GetProfit(Node, Dest);
              Inc(Pair.mDist, Profit);
            end; // if DestBloss is TBlossom
            // Помещаем простые вершины цветка в очередь
            // для расширения текущего дерева
            PutInQue(gQue, PairBloss);
          end // else Assigned(Pair)
        end else begin
          // Здесь GetRoot(Dest) = aRoot,
          // то есть, обнаружена вершина L.mDest,
          // уже принадлежащая текущему дереву
          if aNode.GetBlossom <> Dest.GetBlossom then begin
            // Здесь две вершины не принадлежат одному цветку
            if GetOut(Dest) then begin
```

```
                // Здесь Dest является внешней вершиной,
                // и потому создаём новый цветок
                Bloss:= BlossomMake(Link);// новый цветок
                Bloss.mRoot:= aRoot;      // метим текущим деревом
                Bloss.mOut:= true;        // назначаем внешним
                gBlossomStack.Push(Bloss);// запоминаем в стеке
              end;
            end;  // if aNode.GetBlossom <> Dest.GetBlossom
          end; // else
        Link:= aNode.OutLinkNext;
      end; // while Assigned(L)
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  var Node : TNode;        // Текущая внешняя вершина
      DeltaMin: integer;   // Минимальное дельта для расширения дерева
      Success: boolean;    // Признак возможности расширить дерево

  begin  // function Alter(aRoot: TNode): TNode;

    // Построение альтернирующего дерева с корнем в вершине aRoot
    // с целью поиска сильной или слабой увеличивающей цепи

    Result:= nil;
    // Сброс полей всех вершин в начальное состояние
    LocalReset;

    // Предустановка полей корневой вершины
    with aRoot do begin
      mRoot:= aRoot;  // Метим корнем
      mOut:= true;    // Внешняя вершина
      mPred:= nil;    // Предшествующая вершина в дереве
      mLink:= nil;    // Линк на предшествующую вершину
    end;

    // Инициируем текущую очередь вершин gQue
    // aRoot -- обобщённая вершина (может быть цветком)

    PutInQue(gQue, aRoot);

    repeat
      // gAppendix -- множество добавляемых в очередь вершин.
      // Формируется функцией CalcForExpand
      // после безуспешного поиска альтернирующей цепи.
      // В начальный момент оно пусто.
      // Заносим добавляемые вершины в текущую очередь:
      with gAppendix do begin
        Node:= GetFirst as TNode;
        while Assigned(Node) do begin
          gQue.Put(Node);   // Текущая очередь обрабатываемых вершин
          Node:= GetNext as TNode;
        end;
      end;
      // Обработка текущей очереди вершин
      // в попытке найти увеличивающую цепь
      Node:= gQue.Get as TNode;
      while Assigned(Node) do begin
        Node.mRoot:= aRoot; // Метим корнем дерева
        // Обработка смежных Node вершин
        // с целью поиска увеличивающей цепи.
        // При этом расширяется текущее дерево.
        Result:= NodeHandle(Node);
```

```
        // Выход из цикла при обнаружении
        // увеличивающей цепи
        if Assigned(Result) then Break;
        // Выбор следующей вершины
        Node:= gQue.Get as TNode;
      end; // while Assigned(Node)

      // Если обнаружена сильная или слабая увеличивающая цепь
      // выход из цикла.

      if Assigned(Result) then Break;

      // Здесь не обнаружена никакая увеличивающая цепь.
      // Вычисляем минимальноое изменение веса вершин,
      // необходимое для расширения текущего дерева.
      // Возвращается Success -- признак возможности расширить дерево.

      DeltaMin:= CalcForExpand(Success);

      // Если нельзя расширить текущее дерево, то выход

      if not Success then Break;

      // Меняем веса в вершинах дерева на величину DeltaMin.
      // Функция ChangeValues меняет веса и возвращает TRUE,
      // если в ходе пересчёта весов
      // был распущен хотя бы один внутренний цветок,
      // принадлежащий данному дереву.

      if ChangeValues(DeltaMin) then begin
        // Здесь был распущен по меньшей мере один цветок
        // и потому возвращаем корневую вершину в начало очереди
        // и начинаем построение дерева заново.
        gMainQue.Push(aRoot.GetBlossom);
        Break;  // Выход из цикла
      end;

      // Здесь после пересчёта весов вершин не был распущен
      // ни один внутренний цветок.
      // Продолжаем попытки расширения дерева
      // с поиском увеличивающей цепи.
    until false;
    // Очистка множества и очереди.
    gAppendix.Clear;
    gQue.Clear;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Соединение в пару двух вершин (обычных или цветков).
  // Вызывается в ходе реверса альтернирующей цепи,
  // а также при роспуске цветка.
  // aLink -- ребро, создающее пару.

  procedure PairMake(aLink : TLink);
  var Owner, Dest : TNode;
  begin
    // Выявляем обобщённые вершины, смежные с ребром aLink
    with aLink do begin
      Owner:= mOwner.GetBlossom; // простая или цветок
      Dest:= mDest.GetBlossom;   // простая или цветок
    end;
    // Формируем поля парных вершин:
    // Парные линки:
```

```
    Owner.mPair:= aLink; Dest.mPair:= aLink;
    // Фиксируем положение вершин в паре
    Owner.mOwnDest:= false; // источник линка
    Dest.mOwnDest:= true;   // приёмник линка
    // Если обобщённая вершина является цветком
    // то устанавливаем в цветке новую базу (рекурсивно)
    if Owner is TBlossom then with Owner as TBlossom do begin
      SetNewBase(aLink.mOwner);
    end;
    if Dest is TBlossom then with Dest as TBlossom do begin
      SetNewBase(aLink.mDest);
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Разрыв пары aNode <-> aNode.mPair
  // Вызывается при перестройке альтернирующей цепи,
  // а также при роспуске цветка.
  // Если задан крайний цветок aBloss,
  // значит пара разрывается при его роспуске,
  // а иначе (nil) -- перед реверсом альтернирующей цепи.

  procedure Disconnect(aNode, aBloss : TNode);

    // Локальная процедура восстановления текущей базы mNewBase
    // в начальное состояние (mBase)

    procedure BasesRestore(aNode : TNode);
    var Node: TNode;
    begin
      // Восстанавливаем базы mNewBase,
      // двигаясь от вложенного цветка к крайнему наружному
      // (рекурсивно)
      Node:= aNode.mBlossom;  // непосредственный цветок
      repeat
        // Восстанавливаем текущую базу в исходное состояние
        with Node as TBlossom do mNewBase:= mBase;
        // Выход, если достигнут крайний цветок
        if Node = Node.mBlossom then Break;
        // Переход к цветку следующего уровня
        Node:= Node.mBlossom;
      until false;
    end;

  var Node, NodeBloss : TNode;
  begin
    // Пара разрывается внутри цветка?
    if Assigned(aBloss) then begin
      // Если разрыв пары происходит внутри цветка aBloss
      // (в процедуре его роспуска),
      // то ищем вложенный цветок,
      // предшествующий в иерархии крайнему цветку aBloss
      NodeBloss:= aNode;
      // Рекурсия изнутри наружу:
      while NodeBloss.mBlossom <> aBloss do
        NodeBloss:= NodeBloss.mBlossom;
    end else begin
      // Если пара разрывается вне цветка (в цепи реверса)
      // берём крайний цветок для вершины Node
      NodeBloss:= aNode.GetBlossom;
    end;

    // Движение от крайнего цветка внутрь
```

```
    Node:= NodeBloss;  // крайний цветок
    repeat
      Node.mPair:= nil; // гасим его парный линк
      // Выход, если дошли до корневой базы
      if not (Node is TBlossom) then Break;
      // Берём базу следующего (нижнего) уровня иерархии
      Node:= (Node as TBlossom).mBase;
    until false;
    // Если вершина aNode состояла в цветке NodeBloss,
    // и не задан крайний цветок aBloss,
    // то восстанавливаем текущие базы NodeBloss (рекурсивно)
    if (NodeBloss is TBlossom) and not Assigned(aBloss)
      then BasesRestore(aNode);
  end;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура роспуска цветка
  // Выполняется в двух случаях:
  // a) Досрочно -- при обнаружении внутреннего цветка с нулевым весом
  // b) По окончании обработки всех вершин графа в главной очереди

  procedure BlossExpand(aBloss: TBlossom);

    // Разрыв связей вершин с текущим цветком

    procedure RemoveFromBlossom;
    var N : TNode;
    begin
      // Обходим вершины цветка по часовой стрелке
      N:= aBloss.mBase;
      repeat
        N.mBlossom:= N; // Гасим ссылку на цветок
        N.mRoot:= nil;  // Удаляем из текущего дерева
        N:= N.mRight;
      until N = aBloss.mBase;
    end;

    // Поиск в цветке простой вершины с минимальным весом
    // (рекурсивно)

    function GetMinNode(aBloss : TBlossom): TNode;
    var Node, N: TNode;
        MinVal: integer;
    begin
      MinVal:= MaxInt;
      with aBloss as TBlossom do begin
        Result:= mBase;
        Node:= mBase;
        // Обход цветка по часовой стрелке
        repeat
          // Node - обобщённая вершина
          // N -- простая вершина
          // Для цветка вызываем рекрсивно.
          if Node is TBlossom
            then N:= GetMinNode(Node as TBlossom)
            else N:= Node;
          if N.mValue < MinVal then begin
            MinVal:= N.mValue;
            Result:= N;
          end;
          Node:= Node.mRight;
        until Node = mBase;
```

```
      end;
    end;

  var N1, N2 : TNode; // Пара смежных вершин в цветке
      Next: TNode;    // Внутренняя пара базовой вершины
      Base : TNode;   // Новая (текущая) база цветка
      Pair : TNode;   // Внешняя пара цветка
      Delta: integer; // Общая поправка весов вершин в цветке
      Link: TLink;    // Линк для создания пары
      Direct: boolean;// Направление обхода цветка

  begin  // procedure BlossExpand

    with aBloss do begin
      // Выясняем, имеется ли у цветка пара (mPair)
      if Assigned(mPair) then begin
        // Здесь цветок имеет пару
        // Выявляем простую парную вершину внутри цветка
        if mOwnDest
          then Base:= mPair.mDest   // приёмник линка mPair
          else Base:= mPair.mOwner; // источник линка mPair
        Next:= SetNewBase(Base);    // и делаем её новой базой
        // Здесь Next -- внутренняя пара Base (или nil)
      end else begin
        // Здесь цветок не имеет пары,
        // потому сдвигаем пары в цветке по кругу так,
        // чтобы сумма весов внутри цветка увеличилась.
        // Для этого сдвигаем новую базу вправо или влево
        // от первичной базы.
        // Находим в цветке простую вершину с минимальным весом
        Base:= GetMinNode(aBloss);
        // Берём охватываемый цветок, содержащий эту вершину
        while Base.mBlossom <> aBloss do Base:= Base.mBlossom;
        // Устанавливаем в качестве новой базы
        Next:= SetNewBase(Base);
        // Разрываем пару
        Pair:= Base.GetPair;
        if Assigned(Pair) then begin
          Disconnect(Base, aBloss); // отрываем новую базу
          Disconnect(Pair, aBloss); // отрываем её пару
        end;
      end;
      // Восстановление вершинных чисел вершин цветка
      // (за исключением текущей (новой) базовой вершины)
      if mValue <> 0 then begin
        Delta:= mValue div 2;  // mValue по определению чётно
        N1:= mNewBase.mRight;
        repeat
          if N1 is TBlossom
            then Inc(((N1 as TBlossom).GetNewBase).mValue, Delta)
            else Inc(N1.mValue, Delta);
          N1:= N1.mRight;
        until N1 = mNewBase;
      end;

      // Если база не изменилась,
      // то внутренние пары остаются прежними

      if mNewBase=mBase then begin
        // Здесь база цветка НЕ изменилась
        // Разрыв связей вершин с текущим цветком
        RemoveFromBlossom;
```

```
        // Парой для парной вершины становится база цветка
        if Assigned(mPair) then PairMake(mPair);
        Exit;
      end;

      // Здесь база цветка изменилась (Next <> NIL)
      // Разрываем связи вершин с текущим цветком
      RemoveFromBlossom;

      // Выявляем парную вершину новой базы
      Next:= Next.GetBlossom;

      // Обходим цветок либо по часовой, либо против
      // Выясняем направление обхода
      Direct:= Next = mNewBase.mRight;
      // Обход цветка
      N1:= mNewBase; // Начинаем с новой базы
      repeat
        if Direct then begin
          // Обход цветка по часовой
          N2:= N1.mRight;
          Link:= N2.mRLnk;  // Будущий парный линк
          N1:= N2.mRight;
        end else begin
          // Обход цветка против часовой
          N2:= N1.mLeft;
          N1:= N2.mLeft;
          Link:= N1.mRLnk;  // Будущий парный линк
        end;
        PairMake(Link);// Создаём пару N1 <=> N2
      until N1= mBase; // Завершаем на первичной базе

      // Парой для парной вершины становится новая база цветка
      if Assigned(mPair) then PairMake(mPair);
    end;   // with aBloss
  end;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура реверса альтернирующей цепи.
  // Выполняется после обнаружения
  // либо сильной, либо слабой увеличивающей цепи.
  // aFin -- конечная белая (непарная) вершина цепи

  procedure Reverse(aFin : TNode);
  var N1, N2, Next : TNode;
  begin
    // Начинаем с конечной белой вершины
    // и двигаемся в направлении корня дерева
    Next:= aFin;
    repeat
      N2:= Next.GetBlossom;  // N2 -- внутренняя
      N1:= N2.mLink.mOwner.GetBlossom;  // N1 -- внешняя
      Next:= N1.GetPair;  // Next -- простая вершина
      PairMake(N2.mLink); // создаём пару N1 <=> N2
      // Пока не достигли корня дерева
    until not Assigned(Next);
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск очередной пары из вершины aNode

  procedure Handle(aNode: TNode);
  var FreeNode,      // Конечная вершина цепи
```

```
    FreeBloss,      // Обобщённая конечная вершина
    Start : TNode; // Конечная вершина для реверса цепи
begin
  // Строим альтернирующее дерево с корнем в aNode
  // в попытке найти либо сильную, либо слабую
  // увеличивающую цепь.
  // FreeNode -- конечная вершина цепи

  FreeNode:= Alter(aNode);

  // Если увеличивающая цепь не найдена, то выход
  if not Assigned(FreeNode) then Exit;

  // Здесь обнаружена увеличивающая цепь
  // (слабая либо сильная)

  // Берём обобщённую конечную вершину
  // (цветок или саму эту вершину)
  FreeBloss:= FreeNode.GetBlossom;
  // Имеет ли она пару?
  if not Assigned(FreeBloss.mPair) then begin
    // Пара у конечной вершины не обнаружена,
    // значит это сильная увеличивающая цепь
    Start:= FreeBloss;  // Конечная вершина для реверса цепи
  end else begin
    // Да, пара обнаружена, -- это слабая увеличивающая цепь.
    // Берём пару этой конечной вершины
    // и назначаем конечной вершиной для реверса цепи
    Start:= FreeBloss.GetPair.GetBlossom;
    // Входит ли конечная вершина в цветок?
    if FreeBloss is TBlossom then with FreeBloss as TBlossom do begin
      // Да, входит в цветок цветок
      Disconnect(GetNewBase, nil);  // Отсоединяем базу цветка
      SetNewBase(FreeNode); // Назначаем простую вершину базой цветка
    end else begin
      // Нет, эта простая вершина не входит в цветок
      Disconnect(FreeNode, nil);  // Отсоединяем вершину от пары
    end;
    // Возвращаем конечную вершину в начало главной очереди
    gMainQue.Push(FreeBloss);
  end;
  // Если альтернирующая цепь содержит хотя бы одно ребро,
  // вызываем процедуру реверса этой цепи,
  // начиная с вершины Start.
  if Start <> aNode.GetBlossom then Reverse(Start);
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Раскрытие и удаление цветков по завершении обработки вершин.
// Цветки раскрываются в порядке, обратном их созданию

procedure RemoveBlossoms;
var Bloss: TBlossom;
begin
  while gBlossomStack.GetCount>0 do begin
    Bloss:= gBlossomStack.Pop as TBlossom;
    BlossExpand(Bloss);
    Bloss.Free;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Формирование множества пар.
// Вызывается после обработки всех вершин из главной очереди
```

```
  // и роспуска всех цветков.

  function PairsMake: TCostSet;
  var   Node: TNode;
        Pair: TPair;
           L: TLink;
           S: TSet;
        Cost: integer;
  begin
    Cost:= 0;
    // В множество S будут вставляться создаваемые пары
    S:= CreateSet;
    // Перебираем вершины графа
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Assigned(Node.mPair) then begin
        // Здесь вершина Node имеет пару,
        // создаём объект-пару
        Pair:= TPair.CreateEmpty;
        with Pair do begin
          // Заполняем поля объекта
          mDestroy:= true;
          mSet.Insert(Node);
          mSet.Insert(Node.GetPair);
          L:= Node.GetLink(Node.GetPair);
          // В поле L.mTemp хранится начальное значение L.mValue
          mCost:= L.mTemp;
          Inc(Cost, mCost);
        end;
        // Вставляем пару в множество пар
        S.Insert(Pair);
        // Очистка полей mPair
        Node.GetPair.mPair:= nil;
        Node.mPair:= nil;
      end;
      Node:= NodeNext;
    end;
    // Если вес рёбер не учитывается,
    // то стоимость паросочетания равна количеству пар
    if aMode = pMaxN then Cost:= S.GetCount;
    // Создаём множество с оценкой Cost
    Result:= TCostSet.Create(Cost, S, true);
  end;

var Node: TNode;  // Очередная вершина

begin

  Self.ResetNodes;  // Инициализация вершин (сброс)
  InitGraphValues;  // Установка начальных значений вершинных чисел

  gMainQue:= TBuffer.Create;// Главная очередь обрабатываемых вершин
  gQue:= TBuffer.Create;    // Очередь обрабатываемых вершин
  gBlosNum:= 0;             // Уникальный идентификатор цветка
  gAppendix:= CreateSet;    // Добавляемые к текущему дереву вершины
  gBlossomStack:= TBuffer.Create;  // Стек для хранения цветков

  // Заносим все вершины в главную очередь
  Node:= NodeFirst;
  while Assigned(Node) do begin
    gMainQue.Put(Node);
    Node:= NodeNext;
```

```
  end;

  // Обработка главной очереди
  while gMainQue.GetCount>0 do begin
    // Извлекаем из начала очереди
    Node:= gMainQue.Get as TNode;
    // Это парная вершина?
    if Assigned(Node.mPair) // если вершина состоит в паре
      // или её цветок состоит в паре
      or Assigned(Node.GetBlossom.mPair)
      then Continue;        // то игнорируем её
    // Непарную (белую) вершину обрабатываем
    Handle(Node);
  end;

  // Завершающие действия:
  RemoveBlossoms;     // Удаление нераскрытых цветков
  gBlossomStack.Free; // Стек для хранения цветков
  gAppendix.Free;     // Добавляемые к текущему дереву вершины
  gQue.Free;          // Очередь обрабатываемых вершин
  gMainQue.Free;      // Главная очередь обрабатываемых вершин
  Result:= PairsMake; // Формирование множества пар
  RestoreGraphValues; // Восстановление весов вершин и рёбер
end;

//#######################################################################
//                                                                      #
//               Паросочетания в двудольном графе                       #
//                                                                      #
//#######################################################################

// Построение минимального совершенного паросочетания.
// Алгоритм был предложен Кёнигом, Эгервари и Куном (венгерский алгоритм)
// Кристофидес, стр. 405
// Обрабатываются только активные дуги, для которых mHigh=0
// Возвращает стоимость паросочетания и его мощность aPairs

// Поля объектов используются следующим образом:
// * TNode.mColor -- белая - экспонированная, чёрная - в паре
// * TNode.mDist  -- вершинное число, степень "поднятия мостовой опоры"
// * TNode.mFlow  -- признак посещения вершины (0/1)
// * TNode.mRoot  -- линк, ведущий к корню текущего дерева
// * TNode.mLink  -- чёрный линк паросочетания для этой вершины
//
// * TLink.mColor -- чёрная - признак принадлежности к паросочетанию
// * TLink.mValue -- стоимость (длина, крутизна) дуги
// * TLink.mHigh  -- признак активности дуги, 0 - активна, 1 - отключена
// * TLink.mLow   -- редуцированная (пониженная) стоимость дуги

function TGraph.MarkMinPairsDicoty(var aPairs: integer): integer;

var    Que: TBuffer;  // очередь вершин
    NodesL: TSet;     // множество левых вершин
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Выполняется начальная установка двудольного графа:
  // все вершины и дуги белые
  // Формируется вспомогательное подмножество левых вершин
  // Возвращает максимально возможное количество пар

  function  InitNodes: integer;
  var Node: TNode;
      Link: TLink;
```

```
    L, R: integer; // счётчики левых и правых вершин
begin
  // Обнуление счётчиков вершин
  L:= 0; R:= 0;
  // Перебор всех вершин:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mColor:= CWhite;   // цвет вершины белый
    if Node.IsLeft then begin
      Inc(L);               // подсчёт левых вершин
      Node.mDist:= MaxInt;  // вершинное число ищем вторым перебором
      NodesL.Insert(Node);  // и вставка в множество левых
    end else if Node.IsRight then begin
      Inc(R);               // подсчёт правых вершин
      Node.mDist:=0;        // вершинное число = 0
    end;
    Node:= NodeNext;
  end; // while
  // Количество ожидаемых паросочетаний равно меньшему из двух:
  if L > R then Result:= R else Result:= L;

  // Перебор левых вершин:
  Node:= NodesL.GetFirst as TNode;
  while Assigned(Node) do begin
    // Перебор исходящих линков
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do if mHigh=0 then begin
        // это активный линк
        Link.mColor:= CWhite;
        // запоминаем вес минимальной дуги
        if Node.mDist > Link.mValue  then
          Node.mDist:= Link.mValue; // начальное "приподнятие опоры"
      end;
      Link:= Node.OutLinkNext;
    end; // while
    Node:= NodesL.GetNext as TNode;
  end; // while

end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск аугментальной (улучшающей) цепи:
// aLeft - корневая экспонированная левая вершина дерева (белая)
// Возвращает экспонированную правую вершину (белую),
// при условии, что найдена аугментальная цепь, а иначе NIL.
// Формирует ссылки в полях TNode.mRoot, ведущие к корню дерева.
// Если экспонированная вершина не найдена,
// то параметр aDelta содержит минимальное приращение цены,
// необходимое для покупки ещё хотя бы одного линка.

function FindAugmenting(aLeft : TNode; var aDelta: integer): TNode;
var Node: TNode;
    Link: TLink;
    Cost: integer;
begin
  Result:= nil;        // ссылка на правую вершину
  Que.Clear;           // очистить очередь вершин
  Que.Put(aLeft);      // и поместить туда корневую вершину
  // Пока очередь не пуста и не обнаружена аугментальная цепь
  while (Que.GetCount > 0) and not Assigned(Result) do begin
    Node:= Que.Get as TNode;
    Node.mFlow:= 1;    // отмечаем посещение вершины
```

```
      if Node.IsLeft then begin
        // Это вершина левой доли
        // Ищем белую вершину в правой доле, а чёрные ставим в очередь
        Link:= Node.OutLinkFirst;
        while Assigned(Link) do begin
          // обрабатываем только активные белые линки
          with Link do if (mHigh=0) and    // активный линк
                          (mColor=CWhite)  // и линк белый
            then begin
              // Вычисляем приведенную (пониженную) стоимость дуги
              Cost:= mValue - (mOwner.mDist - mDest.mDist);
              if Cost = 0 then begin
                // Это достижимая правая вершина (Cost=0)
                if  mDest.mColor = CWhite then begin
                  // Если она белая,
                  // то найдена белая вершина и аугментальная цепь
                  Result:= mDest;       // экспонированная белая вершина
                  Result.mRoot:= Link; // ссылка в направлении корня дерева
                  Break;                // выход из цикла и процедуры
                end else if mDest.mFlow = 0 then begin
                  // Найдена чёрная вершина,
                  // она ещё не посещалась, и потому ставим её в очередь
                  mDest.mRoot:= Link; // ссылка в направлении корня дерева
                  Que.Put(mDest);     // ставим в очередь
                end; // else
              end else begin
                // Это недостижимая правая вершина (Cost > 0),
                // через aDelta возвращаем минимальное приращение цены
                if aDelta > Cost then aDelta:= Cost;
              end;
          end; // with Link
          Link:= Node.OutLinkNext;
        end; // while - конец перебора линков
      end else begin
        // Это внутренняя чёрная вершина в правой доле (Node.mIsLeft = FALSE)
        Link:= Node.mLink;         // извлекаем из неё линк паросочетания
        Link.mOwner.mRoot:= Link; // это ссылка в направлении корня дерева
        // Ставим в очередь найденную внешнюю чёрную вершину в левой доле
        Que.Put(Link.mOwner);
      end;
    end;
  end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Инверсия аугментальной (увеличивающей) цепи.
// Вызывается при условии, что такая цепь обнаружена
// в процедуре FindAugmenting.
// aLeft, aRight -- левая и правая экспонированные (белые) вершины.
// Использует поля Node.mRoot, сформированные процедурой FindAugmenting,
// и формирует из них линки паросочетаний Node.mLink.

procedure Inverse(aLeft, aRight : TNode);
var Node: TNode;
    Link: TLink;
    Flag: boolean;
begin
  // Проходим аугментальную цепь в обратном порядке
  // инвертируя цвета дуг
  Flag:= true;      // инвертируемый флаг определяет цвет ребра (дуги)
  Node:= aRight;
  repeat
    Link:= Node.mRoot as TLink;
    if Flag then begin
```

```
        // Переход справа налево, окраска дуги чёрным
        // Красим белым старый линк паросочетания
        with Node do if Assigned(mLink) then mLink.mColor:= CWhite;
        Node.mLink:= Link;       // запоминаем новый линк паросочетания
        Link.mColor:= CBlack;    // и красим его чёрным
        Node:= Link.mOwner;      // переход в сторону корня дерева
        Node.mLink:= Link;       // и здесь запоминаем новый линк
паросочетания
      end else begin
        // Переход слева направо, окраска дуги белым
        Link.mColor:= CWhite;
        Node:= Link.mDest;     // переход в сторону корня дерева
      end;
      Flag:= not Flag; // инвертируемый флаг определяет цвет ребра (дуги)
    until Node = aLeft;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка у всех вершин полей TNode.mFlow -- признаков посещения.
  // Вызывается перед обходом свободных (белых) левых вершин

  procedure ClearFlags;
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= 0;   // Это признак посещения при постройке деревьев
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обход свободных (белых, экспонированных) левых вершин
  // с целью построения улучшающих аугментальных цепей.
  // Возвращает количество обнаруженных цепей (и соответственно пар).
  // Если ни одна такая цепь не будет обнаружена,
  // то через параметр aDelta возвращает минимальное "поднятие опоры",
  // необходимое для выпрямления хотя бы одного моста

  function FindPairs(var aDelta: integer): integer;
  var LNode, RNode : TNode;
  begin
    Result:= 0;        // счётчик обнаруженных пар
    aDelta:= MaxInt;   // минимальное приращение цены
    ClearFlags;        // очистка признаков посещения вершин TNode.mFlow
    // Перебор левых вершин
    LNode:= NodesL.GetFirst as TNode;
    while Assigned(LNode) do begin
      if LNode.mColor = CWhite then begin
        // Это открытая (экспонированная) левая вершина
        // Ищем открытую вершину справа и формируем aDelta
        RNode:= FindAugmenting(LNode, aDelta);
        if Assigned(RNode) then begin
          // Аугментальная цепь найдена
          // Инвертируем цвета дуг от RNode к LNode
          Inverse(LNode, RNode);
          // Две крайние вершины цепи метим чёрным
          // как вошедшие в паросочетание
          LNode.mColor:= CBlack;
          RNode.mColor:= CBlack;
          Inc(Result);  // и наращиваем счётчик пар
        end;
      end;
      LNode:= NodesL.GetNext as TNode;
```

```
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура корректирует вершинные числа Node.mDist на величину aDelta
  // (приподнимает платформу)
  // Вызывается после обхода экспонированных (белых) вершин
  // после того, как ни одна новая пара не обнаружена.

  procedure Correct(aDelta: integer);
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Если вершина посещалась, то нарастить вершинное число
      if Node.mFlow <> 0 then begin
        Inc(Node.mDist, aDelta);
      end;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт стоимости паросочетания по сумме чёрных линков
  // и формирование пониженной стоимости (крутизны) дуг TLink.mLow
  // с учётом текущих значений вершинных чисел TNode.mDist.
  // (поля TLink.mLow используются для поиска цикла Гамильтона)

  function CalcCost: integer;
  var Node: TNode;
      Link: TLink;
  begin
    Result:= 0;
    // Перебор вершин левой доли
    Node:= NodesL.GetFirst as TNode;
    while Assigned(Node) do begin
      // Node.mLink - это чёрный линк паросочетания
      Inc(Result, Node.mLink.mValue);
      // Перебор исходящих линков
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // если это активный линк, вычисляем редуцированную стоимость
        with Link do
          if mHigh=0 then mLow:= mValue - (mOwner.mDist - mDest.mDist);
        Link:= Node.OutLinkNext;
      end;
      Node:= NodesL.GetNext as TNode;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
   P: integer;      // количество новых пар на очередной итерации
   Delta: integer; // приращение, необходимое для приподнятия платформы
   Count: integer; // счётчик осташихся не найденных пар

begin  { TGraph.SetPairsDicoty }

  Que:= TBuffer.Create; // очередь вершин
  NodesL:= CreateSet;   // множество левых вершин
  Count:= InitNodes;    // возвращает максимально возможное число пар
  aPairs:= Count;

  repeat     // повторять до формирования всех паросочетаний
    repeat   // повторять, пока возможно приращение паросочетания
```

```
      P:= FindPairs(Delta); // P - количество обнаруженных новых пар
                            // Delta - миним. приращение для покупки дуги
      Dec(Count, P);        // Count - счётчик оставшихся пар
    until (P=0) or (Count=0);
    // Если паросочетание не полно, то скорректировать вершинные числа
    // на величину Delta (приподнять платформу),
    // "выровняв" тем очередной мост или несколько мостов
    if Count > 0 then Correct(Delta);
  until Count=0;
  // Здесь максимальное паросочетание достигнуто,
  // подсчитать стоимость паросочетания и остатки стоимостей дуг
  Result:= CalcCost;
  // Очистка памяти:
  NodesL.Free;  // множество левых вершин
  Que.Free;     // очередь вершин
end;


// Формирует множество паросочетания после того, как помечены его линки
// методом MarkMinPairsDicoty

function TGraph.GenMinPairsDicoty: TCostSet;
var Cost: integer;   // стоимость
    Pairs: integer;  // количество найденных пар
    Node: TNode;
    S : TSet;
begin
  Cost:= MarkMinPairsDicoty(Pairs);
  S:= CreateSet;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft
      then S.Insert(Node.mLink);
    Node:= NodeNext;
  end;
  Result:= TCostSet.Create(Cost, S, true);
end;


// Построение наибольшего паросочетания минимальной стоимости
// через вычисление ПОТОКА минимальной стоимости

function TGraph.GenMinPairsDicotyFlow: TCostSet;
var  S, T : TNode;
     Node : TNode;
     Link : TLink;
     Cost : integer;
      Cnt : integer;
      Res : TSet;

begin

  // Всем дугам графа назначаем единичную пропускную способность
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do begin
        mTemp:= mHigh;  // временно сохраняем
        mHigh:= 1;      // и устанавливаем единичную пропускную способность
      end;
      Link:= Node.OutLinkNext;
```

```
    end;
    Node:= NodeNext;
  end;

  // Создаём искусственные исток S и сток R:
  S:= TNode.Create(0, Self);
  T:= TNode.Create(0, Self);

  // Перебираем вершины, соединяя левую долю с истоком, а правую со стоком
  // дугами со стоимостью =0, пропускной способностью =1
  // Одновременно подсчитываем вершины в левой доле
  Cnt:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft then begin
      // Дуга из истока в левую долю,
      Link:= S.MakeLink(Node, 0);
      Link.mHigh:= 1;
      Inc(Cnt);
    end  else begin
      // Дуга из правой доли в сток,
      Link:= Node.MakeLink(T, 0);
      Link.mHigh:= 1;
    end;
    Node:= NodeNext;
  end;  // while

  // Из количества вершин в левой и правой долях выбираем меньшее:
  if Cnt > mNodes.GetCount div 2 then Cnt:= mNodes.GetCount - Cnt;

  // Вставляем в граф:
  InsertNode(S);   // исток
  InsertNode(T);   // сток

  // Находим поток минимальной стоимости:
  Cost:= CalcMinCostFlow(S, T, Cnt);

  // Удаляем и уничтожаем искусственные исток и сток:

  RemoveNode(T);
  RemoveNode(S);
  T.Free; S.Free;

  Res:= CreateSet;
  if Cost>=0 then begin
    // Здесь паросочетание существует
    // В паросочетание включаем дуги с ненулевым потоком (Link.mFlow <> 0)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do begin
          if mFlow <> 0 then Res.Insert(Link);
          mHigh:= mTemp; // восстанавливаем исходные значения
        end;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
  end; // if
  // Формируем множество со стоимостью:
  Result:= TCostSet.Create(Cost, Res, true);
```

```
end;


//#######################################################################
//                                                                      #
//              Цикл Эйлера и Задача почтальона                         #
//                                                                      #
//#######################################################################

// В зависимости от того, ориентирован граф, или нет,
// вызывает метод для орграфа или неориентированного графа

function TGraph.GenPostPath(var aCost: integer): TBuffer;
begin
  if mDirect
    then Result:= GenPostPathDir(aCost)
    else Result:= GenPostPathUndir(aCost)
end;

// Генерация цикла (цепочки рёбер) в неориентированном графе

function TGraph.GenPostPathUndir(var aCost: integer): TBuffer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Изменение степени ребра (двух полеё TLink.mLimit)

  procedure ModifyLimit(aLink: TLink; aDelta: integer);
  var L: TLink; // встречный линк
  begin
    Inc(aLink.mLimit, aDelta);              // прямой линк
    L:= aLink.mDest.GetLink(aLink.mOwner);  // встречный линк
    Inc(L.mLimit, aDelta);
  end;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества вершин с нечётной степенью
  // и предварительная подготовка полей TNode.mLimit

  function GenOddNodes: TSet;
  var Node: TNode;
  begin
    Result:= nil;
    // Перебор всех вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Сброс:  mColor= 0;  mPred= mLink= nil;  mDist= mFlow= MaxInt;
      Node.ResetNode;
      // В поле mLimit сохраним количество линков вершины (степень):
      Node.mLimit:= Node.OutGetCnt;  //Node.mLnkOut.GetCount;
      // Если количество рёбер нечётное, заносим в множество результата:
      if Odd(Node.mLimit) then begin
        if not Assigned(Result) then Result:= CreateSet;
        Result.Insert(Node);
      end;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Установка степеней рёбер в полях TLink.mLimit
  // и степеней вершин в полях TNode.mLimit

  procedure LimitsPrepare;
```

```
var OddNodes: TSet;      // множество вершин с нечётной степенью
      Pairs : TCostSet; // паросочетание минимального веса
       Pair : TPair;    // пара вершин
         Gr : TGraph;   // вспомогательный граф
    Ni, Nj  : TNode;    // вершины исходного графа
    N2i,N2j : TNode;    // копии вершин исходного графа во вспомогательном
       NFin : TNode;    // конечная вершина в кратчайшем пути
       Dist : integer;  // кратчайшее расстояние между вершинами
       Link : TLink;
       i, j : integer;
begin
  // Устанавливаем степени всех рёбер в единицу:
  SetLinksLimit(1);
  // Собираем множество вершин с нечётными степенями:
  OddNodes:= GenOddNodes;
  // Если таких вершин нет, то выход:
  if not Assigned(OddNodes) then Exit;

  // Создаём вспомогательный граф:
  Gr:= TGraph.Create('Odd Nodes:', // имя графа произвольно
                     false,        // граф не ориентирован
                     false,        // вершины не нагружены
                     true          // рёбра нагружены
                    );
  // И вставляем в него копии вершин с нечётными степенями:
  Ni:= OddNodes.GetFirst as TNode;
  while Assigned(Ni) do begin
    Gr.InsertNode(Ni.Copy as TNode);
    Ni:= OddNodes.GetNext as TNode;
  end;
  // Для определения взаимных расстояний строим карту дальних указателей
  // в исходном графе:
  Self.InitMap_Floyd;
  // Строим рёбра между всеми вершинами вспомогательного графа
  // (создаём полный граф).
  // Длина рёбер устанавливается равной длине кратчайшего пути
  // между вершинами исходного графа
  for i:= 1 to OddNodes.GetCount-1 do begin
    Ni:= OddNodes.GetItem(i) as TNode;     // вершина из множества нечётных
    N2i:= Gr.mNodes.GetItem(i) as TNode;   // её копия в графе
    // Перебор последующих после Ni вершин:
    for j:= i+1 to OddNodes.GetCount do begin
      Nj:= OddNodes.GetItem(j) as TNode;   // вершина из множества нечётных
      N2j:= Gr.mNodes.GetItem(j) as TNode; // её копия во вспомогат. графе
      Dist:= Ni.GetFarLink(Nj).mDist; // кратчайшее расстояние в исх. графе
      Gr.SetLink(N2i, N2j, Dist);     // назначим длине ребра во вспомогат.
    end;
  end;
  // Во вспомогательном графе
  // находим паросочетание с минимальным весом:
  Pairs:= Gr.GenPairs_Edmonds(pMinW);

  // Для каждой из полученных пар увеличиваем степени mLimit
  // вдоль соответствующих кратчайших путей:
  Pair:= Pairs.mSet.GetFirst as TPair;
  while Assigned(Pair) do begin
    // Pair.mSet содержит ровно две вершины пары.
    // Стартовая вершина цепи в исходном графе:
    Ni:= mNodes.GetObject(Pair.mSet.GetFirst as TNode) as TNode;
    // Конечная вершина цепи в исходном графе:
    NFin:= mNodes.GetObject(Pair.mSet.GetNext as TNode) as TNode;
    repeat
```

```
        Inc(Ni.mLimit);          // степень+1 в начальной (промежут.) вершине
        Nj:= Ni.GetNear(NFin);   // следующая вершина на кратчайшем пути
        Link:= Ni.GetLink(Nj);   // линк на промежуточную вершину
        ModifyLimit(Link, +1);   // степень+1 линка
        Ni:= Nj;                 // продвижение к следующей вершине
      until Ni = NFin;           // пока не достигнута конечная
      Inc(Ni.mLimit);            // степень+1 в конечной вершине
      Pair:= Pairs.mSet.GetNext as TPair;  // следующая пара
    end;
    // Очистка памяти:
    Pairs.ClrAndDestroy;     // пары
    Pairs.Free;              // множество пар
    DoneMap;                 // освобождаем карту
    Gr.Free;                 // вспомогательный граф
    OddNodes.Free;           // множество вершин с нечётной степенью
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление кольцевых маршрутов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк
      // (степень которого Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл и путь:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (степень больше нуля):
      Result.Put(Link);         // помещаем линк в буфер результата
      Dec(aNode.mLimit);        // уменьшаем степень текущей вершины
      aNode:= Link.mDest;       // продвигаемся к следующей вершине
      Dec(aNode.mLimit);        // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue); // накопление стоимости (длины пути)
      ModifyLimit(Link, -1);    // уменьшаем степень ребра
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;
    Link: TLink;
    Count: integer; // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathUndir }

  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= 0;                // накопитель стоимости
  LimitsPrepare;            // подготовка полей mLimit (пределов посещ.)
  Node:= NodeFirst;         // построение начинаем с первой вершины

  // Строим кольцевые маршруты,
  // пока не исчерпаны степени всех вершин и рёбер
  repeat
    // Строим кольцевой маршрут из вершины Node и добавляем в результат:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;        // счётчик для прокрутки буфера Result
```

```
    while Count>0 do begin
      Link:= Result.Get as TLink;   // берём линк из начала буфера
      Result.Put(Link);             // и помещаем в конец
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то стоп.
      // Построение очередного цикла начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
    // Выход из цикла, когда степени всех вершин и рёбер исчерпаны:
  until not Assigned(Node);

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);        // да, возвращаем назад в буфер
      Break;                    // и прекращаем цикл
    end;
    Result.Put(Link);           // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;

//#######################################################################
//
//   Задача почтальона для сильно связанного орграфа
//
//#######################################################################

function TGraph.GenPostPathDir(var aCost: integer): TBuffer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подготовка полей mLimit в дугах и вершинах графа.
  // Возвращает стоимость дополнительного потока

  function LimitsPrepare: integer;
  var Node : TNode;     // текущая вершина
      Link : TLink;     // текущая дуга
      Flag : boolean;   // признак наличия асимметричных вершин
      S, T : TNode;     // искусственные исток и сток
      Flow: integer;    // величина корректирующего потока
  begin
    Result:= 0;
    // Устанавливаем пределы посещения всех дуг в единицу:
    SetLinksLimit(1);
    // Для всех вершин определяем разности полустепеней входа и выхода,
    // а также степени вершин TNode.mLimit:
    Flag:= false;         // признак асимметрии графа
    // Перебор вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        mLimit:= InGetCnt + OutGetCnt;  // степень вершины
        mDist:= InGetCnt - OutGetCnt;   // асимметрия
        Flag:= Flag or (mDist<>0); // признак наличия асимметричных вершин
      end;
```

```
    Node:= NodeNext;
  end;
  // Если все вершины симметричны (все полустепени совпадают),
  // то выход из процедуры:

  if not Flag then Exit;

  // Для несимметричного графа строим поток минимальной стоимости
  // с величиной, равной суммарной ассиметрии
  // Создаём:
  S:= TNode.Create(0, Self); // вспомогательный исток
  T:= TNode.Create(0, Self); // вспомогательный сток

  Flow:= 0; // здесь подсчитаем суммарный поток
  // Обработка всех вершин, кроме S и T:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // Во всех дугах формируем данные для вычисления потока:
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      Link.mLow:= 0;            // минимальный поток = 0
      Link.mHigh:= MaxInt;      // максимальный не ограничен
      Link:= Node.OutLinkNext;
    end;
    // Если вершина несимметрична (mDist <> 0),
    // соединяем её с искусственным истоком либо стоком:
    if Node.mDist < 0 then begin
      // создаём дугу из вершины во вспомогательный сток:
      Link:= Node.MakeLink(T, 0); // создаём линк
      Link.mHigh:= -Node.mDist;   // пропускная способность дуги
    end else if Node.mDist > 0 then begin
      // Накапливаем требуемый поток:
      Inc(Flow, Node.mDist);      // асимметрия вершины
      // создаём дугу из вспомогательного истока в вершину:
      Link:= S.MakeLink(Node, 0); // создаём линк
      Link.mHigh:= Node.mDist;    // пропускная способность дуги
    end;
    Node:= NodeNext;
  end; // while
  // Вставляем вспомогательные вершины в граф:
  mNodes.Insert(S);          // исток
  mNodes.Insert(T);          // сток

  // Распределяем поток Flow по дугам и подсчитываем его стоимость:
  Result:= CalcMinCostFlow(S, T, Flow);

  // Удаляем из графа и освобождаем:
  RemoveNode(S);  S.Free;     // вспомогательный исток
  RemoveNode(T);  T.Free;     // вспомогательный сток

  // Добавляем полученные потоки в дугах к степеням дуг и вершин
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      Flow:= Link.mFlow;  // поток в дуге
      // При наличии потока
      // добавляем его к степеням дуги и двух смежных вершин:
      if Flow > 0 then begin
        Inc(Link.mLimit, Flow);          // + к дуге
        Inc(Link.mOwner.mLimit, Flow);   // + к источнику дуги
        Inc(Link.mDest.mLimit, Flow);    // + к приёмнику дуги
```

```
        end;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление частичных циклов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк (Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (лимит больше нуля):
      Result.Put(Link);         // помещаем линк в буфер результата
      Dec(aNode.mLimit);        // уменьшаем степень текущей вершины
      aNode:= Link.mDest;       // продвигаемся к следующей вершине
      Dec(aNode.mLimit);        // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue); // накопление стоимости (длины)
      Dec(Link.mLimit);         // уменьшаем степень дуги
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;      // текущая вершина
    Link: TLink;      // текущая дуга
    Count: integer;   // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathDir }

  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= -1;     // Стоимость цепи на случай отсутствия решения

  // Если граф не сильно связан, то решения нет:

  if not IsLinked then Exit;

  aCost:= 0;           // накопитель стоимости
  LimitsPrepare;       // подготовка степеней mLimit вершин и рёбер
  Node:= NodeFirst;    // построение начинаем с любой вершины

  // Цикл добавления циклов к буферу Result:
  repeat
    // Строим циклы из текущей вершины Node
    // и добавляем в буфер Result:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;         // счётчик для прокрутки буфера
    while Count>0 do begin
      Link:= Result.Get as TLink;    // берём линк из начала буфера
      Result.Put(Link);              // и помещаем в конец
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то
```

```
      // построение очередного цикла начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
    // Выход, если степени всех вершин исчерпаны
  until not Assigned(Node);

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);        // да, возвращаем назад в буфер
      Break;                    // и прекращаем цикл
    end;
    Result.Put(Link);           // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;

//#######################################################################
//                                                                      #
//            Задача коммивояжёра и цикл Гамильтона                    #
//                   Майника, гл. 7, стр. 251                          #
//                                                                      #
//#######################################################################

//  Поиск цикла Гамильтона полным перебором в глубину

function TGraph.GenHamilton_Full(var aCost: integer): TBuffer;
var
    Cost: integer;    // текущая стоимость цикла
    Count: integer;   // счётчик окрашенных вершин
    First: TNode;     // стартовая вершина
    Node: TNode;      // текущая вершина
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура сохранения текущего решения.
  // Вызывается после окраски всех вершин.
  // aNode - последняя окрашенная вершина
  procedure SaveResult(aNode: TNode);
  begin
    aCost:= Cost;          // сохраняем стоимость цикла
    First.mPred:= aNode;   // замыкаем цикл
    Result.Clear;          // очищаем буфер результата
    Node:= aNode;          // заполняем буфер с последней вершины
    repeat
      Result.Push(Node);   // заносим в буфер в обратном порядке
      Node:= Node.mPred;   // предыдущая вершина
    until Node=First;      // пока не достигнем исходной вершины
    Result.Push(Node);     // заносим исходную вершину
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура поиска в глубину

  procedure Local(aNode: TNode);
  var L : TLink;  // текущий исходящий линк
  begin
    aNode.mColor:= CBlack;  // красим данную вершину
```

```
    Inc(Count);              // и наращиваем счётчик окрашенных
    // Все вершины окрашены?
    if Count < mNodes.GetCount then begin
      // Нет, ищем соседнюю неокрашенную перебором исходящих линков
      L:= aNode.OutLinkFirst;
      while Assigned(L) do begin
        if L.mDest.mColor = CWhite then begin
          // Нашли неокрашенную:
          Inc(Cost, L.mValue);         // накапливаем стоимость
          if Cost < aCost then begin   // если стоимость ниже минимальной
            L.mDest.mPred:= aNode;     // то метим предыдущей вершиной
            Local(L.mDest);            // и рекурсивно вызываем эту же проц.
          end;
          Dec(Cost, L.mValue);         // восстанавливаем стоимость
        end;
        L:= aNode.OutLinkNext;         // следующий исходящий линк
      end;
    end else begin
      // Здесь Count = mNodes.GetCount -- все вершины окрашены
      // Ищем замыкающий линк на стратовую вершину First
      L:= aNode.OutLinkFirst;
      while Assigned(L) and (L.mDest<>First)
        do L:= aNode.OutLinkNext;
      if Assigned(L) then begin
        // Замыкающий линк найден:
        Inc(Cost, L.mValue);       // накапливаем стоимость
        if Cost < aCost            // если она меньше минимальной
          then SaveResult(aNode); // сохраняем цикл и его стоимость
        Dec(Cost, L.mValue);       // восстанавливаем стоимость
      end;
    end;
    // При выходе восстанавливаем счётчик вершин и цвет
    Dec(Count);
    aNode.mColor:= CWhite;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Temp: TBuffer; // Для оценки начальной стоимости жадным алгоритмом

begin { TGraph.GenHamilton_Full }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                  // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;

  // Оценка начальной стоимости aCost жадным алгоритмом
  Temp:= GenHamilton_Greed(aCost, true);
  Temp.Free;

  ResetNodes;            // очистка вспомогательных полей
  Cost:= 0;              // накопленная стоимость = 0
  Count:= 0;             // счётчик окрашенных вершин = 0
  First:= NodeFirst;     // стартовая вершина
  Local(First);          // вызов рекурсивной процедуры
  // Если цикл не обнаружен, возвращаем минус 1
  if aCost = MaxInt then aCost:=-1;
end;

//#######################################################################
//  Упрощённый жадный алгоритм
//  поиска минимального Гамильтонова цикла
```

```
//#######################################################################

function TGraph.GenHamilton_Greed(var aCost: integer;
                                  aGreed: boolean
                                  ): TBuffer;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение первого встретившегося белого линка
  function GetAnyLink(aNode: TNode): TLink;
  begin
    Result:= aNode.OutLinkFirst;
    while Assigned(Result) and
          (Result.mDest.mColor <> CWhite)
      do Result:= aNode.OutLinkNext;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение ближайшего белого линка
  function GetBestLink(aNode: TNode): TLink;
  var  Link: TLink;
       BestCost: integer;
  begin
    Result:= nil;
    BestCost:= MaxInt;
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      if (Link.mDest.mColor = CWhite) and
         (Link.mValue < BestCost) then begin
        BestCost:= Link.mValue;
        Result:= Link;
      end;
      Link:= aNode.OutLinkNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    Cost: integer;    // текущая стоимость цепи
    Count: integer;   // счётчик неокрашенных вершин
    Start: TNode;     // начальная вершина
    Node: TNode;      // текущая вершина
    L : TLink;        // текущий линк

begin { GenHamilton_Greed }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                  // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  ResetNodes;            // очистка вспомогательных полей
  Start:= NodeFirst;     // начинаем с первой вершины
  Node:= Start;
  Cost:= 0;              // накопленная стоимость = 0
  Count:= mNodes.GetCount-1; // счётчик неокрашенных вершин
  while Count > 0 do begin
    Node.mColor:= CBlack;     // красим очередную вершину
    Result.Put(Node);
    // перебираем соседние неокрашенные вершины
    if aGreed
      then L:= GetBestLink(Node) // ищем ближайшую неокрашенную
      else L:= GetAnyLink(Node); // ищем любую неокрашенную
    // Если неокрашенных нет, прервать цикл
    if not Assigned(L) then break;
    // Нашли неокрашенную:
    Inc(Cost, L.mValue);     // накапливаем стоимость
```

```
    Dec(Count);              // счётчик - 1
    Node:= L.mDest;          // следующая вершина
  end; // while

  if Count=0 then begin
    // Здесь пройдены все вершины
    Result.Put(Node);         // заносим в результат предпоследнюю
    L:= Node.GetLink(Start); // линк из последней в первую
    if Assigned(L) then begin
      Inc(Cost, L.mValue);  // накапливаем стоимость
      aCost:= Cost;         // возвращаем стоимость
    end else begin
      Result.Clear;
    end;
  end;
end;

//#######################################################################
//         Поиск минимального Гамильтонова цикла
// Усовершенствованный алгоритм на базе двудольного графа
//#######################################################################

type  // Специальная вершина для вспомогательного двудольного графа

      THamNode = class (TNode)
        mLeft : boolean;     // признак левой вершины (TRUE)
        mNode : TNode;       // родовая вершина
        mBack : THamNode;    // ссылка справа налево и слева направо
        constructor Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
        function  Compare(arg: TItem): TCompare; override;
        function  GetName: string;  override;
        function  NextCircNode: THamNode;
        function  GenFactorsLinks: TSet;
      end;

constructor THamNode.Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
begin
  inherited Create(0, aOwner);
  mNode:= aNode;     // родовая вершина
  mLeft:= aLeft;     // сторона: TRUE - Left, FALSE - Right
end;

function  THamNode.GetName: string;
begin
  // Сторона + имя родовой вершины
  if mLeft then Result:= 'L.' else Result:= 'R.';
  Result:= Result + mNode.GetName;
end;

// Генерирует множество неотключенных дуг,
// ведущих внутрь текущего фактора (цикла)

function THamNode.GenFactorsLinks: TSet;
var Link: TLink;
begin
  Result:= CreateSet;
  Link:= OutLinkFirst;
  while Assigned(Link) do begin
    // Если дуга ведёт внутрь текущего цикла (фактора)
    // и она не отключена (mHigh = 0), то сохраняем в результате
    with Link do
    if ((mDest as THamNode).mBack.mValue = Self.mValue) and (mHigh = 0)
```

```
      then Result.Insert(Link);  // запомнить в множестве отключенных
    Link:= OutLinkNext;
  end;
end;

// Возвращает следующую левую вершину текущего цикла
// в соответствии с текущим потоком

function THamNode.NextCircNode: THamNode;
begin
  Result:= (mLink.mDest as THamNode).mBack;
end;

// При вставке вершины сортируются по алфавиту

function  THamNode.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  if GetName < (arg as THamNode).GetName
    then Result:= cmpLess
  else if GetName > (arg as THamNode).GetName
    then Result:= cmpGreate
end;

// Специальный двудольный граф для поиска гамильтоновых циклов
// Kristofides - 404

type
  THamGraph = class (TGraph)
    mGraph: TGraph;   // исходный (родовой) граф
    constructor Create(aGraph : TGraph);
  protected
    function Check: boolean;
    function CalcFactors(var aStart: THamNode): integer;
    procedure SaveBest(aBuf: TBuffer);
    function CalcCostAndFactors(var aFactors: integer;//кол-во факторов
                                var aStart: THamNode  // стартовая вершина
                                ): integer;           // стоимость потока
    function Shrink(aFact: integer): THugeGraph;
  end;


constructor THamGraph.Create(aGraph : TGraph);
var Node: TNode;
    HamL, HamR: THamNode;
    Link : TLink;
begin
  inherited Create('Hamilton',  // имя
                   true,        // дуги ориентированы
                   false,       // вершины не нагружены
                   true);       // дуги нагружены

  mGraph:= aGraph;            // ссылка на исходный граф

  // На базе вершин исходного графа создаём пары специальных вершин:
  Node:= aGraph.NodeFirst;
  while Assigned(Node) do begin
    // Создаём пару вершин
    HamL:= THamNode.Create(Self, Node, True);  // исток
    HamR:= THamNode.Create(Self, Node, False); // сток
    // Формируем взаимные ссылки для ускорения доступа:
```

```
    HamL.mBack:= HamR;
    HamR.mBack:= HamL;
    // и вставляем вершины в этот граф
    mNodes.Insert(HamL);
    mNodes.Insert(HamR);
    Node:= aGraph.NodeNext;
  end;

  // Формируем дуги, ведущие из левой доли в правую:
  HamL:= NodeFirst as THamNode;
  while Assigned(HamL) do begin
    // Это левая вершина?
    if HamL.mLeft then  begin
      // Да, обрабатываем её
      PosPush;  // сохраним позицию перебора вершин
      // Перебираем исходящие дуги в вершине исходного графа
      Link:= HamL.mNode.OutLinkFirst;
      while Assigned(Link) do begin
        Node:= Link.mDest;  // целевая вершина исходного графа
        // Ищем соответствующую ей вершину в правой доле:
        HamR:= NodeFirst as THamNode;
        while Assigned(HamR) do begin
          with HamR do if not mLeft and (mNode = Node) then Break;
          HamR:= NodeNext  as THamNode;
        end;
        // Здесь HamR соответствует целевой вершине исходного графа
        // Создаём дугу из левой доли в правую (той же стоимости)
        HamL.MakeLink(HamR, Link.mValue);
        // Переход к следующей дуге исходного графа
        Link:= HamL.mNode.OutLinkNext;
      end; // while
      PosPop;  // восстановим позицию перебора вершин
    end; // if
    HamL:= NodeNext as THamNode;
  end; // while
end;

// Поиск и пометка в поле mValue всех факторов (циклов)
// Возвращает количество факторов.
// Принадлежность вершины к фактору определяется полем mValue

function THamGraph.CalcFactors(var aStart: THamNode): integer;
var NodesCnt: integer;  // счётчик оставшихся необработанных вершин
    Start, Left : THamNode; // стартовая и текущая вершины
    CircLen: integer;   // длина очередного фактора
    MinLen : integer;   // длина кратчайшего фактора
    Link : TLink;
begin
  Result:= 0;          // количество факторов
  MinLen:= MaxInt;     // длина кратчайшего фактора
  // количество вершин исходного графа:
  NodesCnt:= mGraph.mNodes.GetCount;
  // очистка полей mValue двудольного графа:
  Start:= NodeFirst as THamNode;
  while Assigned(Start) do begin
    Start.mValue:= 0;
    Start:= NodeNext as THamNode;
  end;
  // Пока не обработаны все вершины графа (исходного)
  while NodesCnt>0 do begin
    Inc(Result);  // Result= 1, 2, .. - цвет очередного цикла
    CircLen:=0;   // длина текущего фактора (цикла)
```

```
    // Ищем первую неокрашенную вершину в левой доле:
    Start:= NodeFirst as THamNode;
    while Assigned(Start) do begin
      with Start do if mLeft and (mValue = 0) then Break;
      Start:= NodeNext as THamNode;
    end;
    // Начиная с вершины Start обходим двудольный граф
    // отмечая цветом Result очередной замкнутый цикл:
    // => вправо - по направлению текущего потока;
    // <= влево  - по обратным ссылкам mBack
    // пока не вернёмся к исходной вершине
    Left:= Start;
    repeat
      Dec(NodesCnt);         // счётчик оставшихся вершин
      Inc(CircLen);          // длина этого фактора (цикла)
      Left.mValue:= Result; // метим цикл текущим цветом
      // Left.mLink -- дуга из вершины Left в вершину Right
      // через которую течёт ненулевой поток (чёрный линк паросочетания)
      Link:= Left.mLink;
      // переход в левую долю вдоль потока:
      Left:= (Link.mDest as THamNode).mBack;
    until Left = Start;
    // После обхода цикла запоминаем кратчайший:
    if CircLen < MinLen then begin
      MinLen:= CircLen;    // длина фактора (цикла)
      aStart:= Start;      // стартовая вершина фактора (любая из)
    end;
  end; // while NodesCnt >0
end;

// Проверка возможности пропуска через двудольный граф
// насыщенного (максимального) потока
// Возвращает:
// FALSE -- если хотя бы одна из вершин не пропускает ни одной единицы потока
// TRUE  -- если через все вершины проходит хоть одна единица потока
// (Link.mHigh <> 0) -- признак закрытой дуги (заблокированной)

function THamGraph.Check: boolean;
var Node: THamNode;
    Link: TLink;
begin
  Result:= false;
  // Перебор вершин
  Node:= NodeFirst as THamNode;
  while Assigned(Node) do begin
    if Node.mLeft then begin
      // В левой доле исследуем исходящие линки
      Link:= Node.OutLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.OutLinkNext;
      if not Assigned(Link) then Exit;
    end else begin
      // В правой доле исследуем входящие линки
      Link:= Node.InLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.InLinkNext;
      if not Assigned(Link) then Exit;
    end;
    Node:= NodeNext as THamNode;
  end;
  Result:= true;  // насыщенный поток возможен
end;
```

```
// Сохранение текущего потока в качестве наилучшего
// Вызывается при обнаружении очередного лучшего цикла

procedure THamGraph.SaveBest(aBuf: TBuffer);
var Start, Node : THamNode;
begin
  aBuf.Clear; // очистка буфера результата
  // Начинаем с первой вершины в левой доле
  Start:= NodeFirst as THamNode;
  while not Start.mLeft do Start:= NodeNext as THamNode;
  // Обходим цикл от Start до Start
  Node:= Start;
  repeat
    aBuf.Put(Node.mNode);
    Node:= Node.NextCircNode;
  until Node = Start;
end;

// Стягивание факторов в новый граф для вычисления прогноза стоимости
// aFact - количество факторов
// Вызывается после построения потока и факторов в последовательности:
// MarkMinPairsDicoty -> CalcFactors -> Shrink

function THamGraph.Shrink(aFact: integer): THugeGraph;
var i: integer;
    Link: TLink;
    N1, N2: TNodeInt;   // вершины стянутого графа помечены числами
    NH1, NH2: THamNode; // вершины данного двудольного графа
    Cost: integer;      // остаточная стоимость дуги
begin
  // Создаём пустой граф:
  Result:= THugeGraph.Create('',  // имя
                         true,    // граф направленный
                         false,   // вершины не нагружены
                         true     // дуги нагружены
                        );
  // Создаём и вставляем вершины, помеченные числами от 1 до aFact

  for i:= 1 to aFact do Result.InsertNode(TNodeInt.Create(i, 0, Result));

  // Соединяем дугами бесконечной длины все вершины созданного графа

  with Result do begin
    N1:= NodeFirst as TNodeInt;
    while Assigned(N1) do begin
      PosPush;
      N2:= NodeFirst as TNodeInt;
      while Assigned(N2) do begin
        if N1 <> N2 then SetLink(N1, N2, MaxInt);
        N2:= NodeNext as TNodeInt;
      end;
      PosPop;
      N1:= NodeNext as TNodeInt;
    end;
  end;

  // Находим минимальные относительные стоимости дуг нового графа
  // Остаточные стоимости дуг в полях TLink.mLow
  // определены при поиске минимального паросочетания (потока)
  // процедурой TGraph.MarkMinPairsDicoty
```

```
  NH1:= NodeFirst as THamNode;
  while Assigned(NH1) do begin
    // Обработка вершин левой доли:
    if NH1.mLeft then begin
      // Перебираем дуги вершины:
      Link:= NH1.OutLinkFirst;
      while Assigned(Link) do begin
        NH2:= (Link.mDest as THamNode).mBack;
        // Если линк ведёт в другой фактор (цикл), то обрабатываем
        if NH1.mValue <> NH2.mValue then begin
          // Остаточная стоимость дуги после редуцирования
          // (определена процедурой TGraph.MarkMinPairsDicoty)
          Cost:= Link.mLow;
          // Находим линк между вершинами (факторами) в стянутом графе
          // Здесь поля NH1.mValue, NH2.mValue соответствуют факторам
          // и вершинам стянутого графа
          Link:= Result.GetLinkByNumber(NH1.mValue, NH2.mValue);
          // Назначаем этому линку пониженую (редуцированную) цену
          if Link.mValue > Cost then Link.mValue:= Cost;
        end;
        Link:= NH1.OutLinkNext;
      end;
    end;
    NH1:= NodeNext as THamNode;
  end;
  // Выполняем компрессию дуг, заменяя их длину кратчайшими расстояниями
  Result.Compress;
end;


// Оценка прогнозируемой стоимости и подсчёт количества факторов
// при текущем состоянии клапанов (открытых и закрытых дуг)
// Возвращает:
// Result   -- нижний предел стоимости для данного состояния клапанов mHigh
// aFactors -- количество факторов
// aStart   -- стартовую вершину кратчайшего фактора

function THamGraph.CalcCostAndFactors(
                            var aFactors: integer; //кол-во факторов
                            var aStart: THamNode   // стартовая вершина
                            ): integer;            // стоимость потока

var Level: integer; // уровень рекурсии

  // Рекурсивная процедура стягивания факторов

  procedure Local(aGraph: THamGraph);
  var Shrinked: TGraph; // очередной граф, образованный стягиванием факторов
      HamGr: THamGraph; // и соответствующий ему двудольный граф
      Cost: integer;    // стоимость потока
      Factors: integer; // количество факторов
      Dummy: integer;   // количество обнаруженных пар (не используем)
      Node: THamNode;   // любая вершина кратчайшего фактора
  begin

    Inc(Level); // отслеживаем уровень рекурсии

    if Level=1 then begin
      // На первом уровне проверим возможность достижения насыщенного потока
      if not aGraph.Check then begin
        // Насыщенный поток невозможен
        Result:= -1;   Exit;
```

```
      end;
    end;

    // Вычисляем стоимость полного потока на двудольном графе
    Cost:= aGraph.MarkMinPairsDicoty(Dummy);

    // Накапливаем результат (прогнозируемую стоимость)
    Inc(Result, Cost);

    // Находим факторы (циклы)
    Factors:= aGraph.CalcFactors(Node);

    // На первом уровне рекурсии запоминаем количество факторов
    // и любую вершину кратчайшего фактора

    if Level=1 then begin
      aFactors:= Factors;
      aStart:= Node;
    end;

    // Если фактор единственный, то выход с возвращением цены
    if Factors = 1 then Exit;

    // При наличии нескольких факторов (не связанных циклов)
    // стягиваем факторы в вершины, создаём новый полный граф
    // и компрессируем его дуги
    Shrinked:= aGraph.Shrink(Factors);

    // Создаём двудольный граф типа THamGraph
    // и рекурсивно повторяем предыдущие шаги

    HamGr:= THamGraph.Create(Shrinked);
    Local(HamGr);

    // Удаляем вспомогательные графы
    HamGr.Free;
    Shrinked.Free;
    Dec(Level); // уровень рекурсии -1
  end;

begin   { THamGraph.CalcCost }
  Result:=0;   aFactors:= 0; aStart:= nil;
  Level:= 0;
  Local(Self);
end;

// TClosed - элемент очереди, содержит:
// -- стоимость потока,
// -- уникальный идентификатор,
// -- множество закрытых дуг (клапанов),
// -- множество множеств отключаемых дуг кратчайшего фактора

type  TClosed = class (TItem)
        mCost: integer;  // стоимость потока
        mId: integer;    // уникальный идентификатор
        mClosed: TSet;   // множество закрытых дуг (клапанов)
        mFactors: TSet;  // множества отключаемых дуг кратчайшего фактора
        constructor Create(aCost: Integer; aClosed: TSet);
        destructor Destroy;  override;
        function Compare(arg: TItem): TCompare; override;
        procedure InsertLinksSet(aSet: TSet);
        procedure GatesOn;
```

```
        procedure GatesOff;
        procedure Print(var aFile: TextFile); override;
      end;

var ClosedId : integer; // уникальный идентификатор элемента очереди
                        // наращивается в конструкторе объекта

constructor TClosed.Create(aCost: Integer; aClosed: TSet);
begin
  inherited Create;
  Inc(ClosedId);     // наращиваем уникальный идентификатор
  mId:= ClosedId;    // и заносим в поле объекта
  mCost:= aCost;     // стоимость потока на момент создания объекта
  // Создаём и копируем множество закрытых дуг из параметра
  mClosed:= CreateSet;
  if Assigned(aClosed) then mClosed.CopyItems(aClosed);
  // Заготавливаем множество множеств закрываемых дуг
  // (по одному множеству для каждой вершины фактора).
  // В это множество будут вставляться множества закрываемых дуг
  // процедурой TClosed.InsertLinksSet
  mFactors:= CreateSet;
end;

destructor TClosed.Destroy;
begin
  mFactors.ClrAndDestroy;
  mFactors.Free;
  mClosed.Free;
  inherited;
end;

// Добавляет очередное множество отключаемых дуг aSet

procedure TClosed.InsertLinksSet(aSet: TSet);
begin
  mFactors.Insert(aSet)
end;

// Метод сравнения выстраивает объекты в приоритетной очереди
// в порядке не убывания стоимости,
// а при равной стоимости - в порядке вставки
// путём сравнения уникальных идентификаторов

function TClosed.Compare(arg: TItem): TCompare;
begin
  if mCost < (arg as TClosed).mCost
    then Result:= cmpLess
  else if mCost > (arg as TClosed).mCost
    then Result:= cmpGreate
  else { при равенстве стоимостей сравниваем уникальные идентификаторы }
    if mId < (arg as TClosed).mId
      then Result:= cmpLess
    else if mId > (arg as TClosed).mId
      then Result:= cmpGreate
    else Result:= cmpEq
end;

// Отключает дуги, содержащиеся в mClosed

procedure TClosed.GatesOff;
var Link: TLink;
begin
```

```
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // дуга отключается единицей
      Link:= GetNext as TLink;
    end;
  end;
end;

// Включает дуги, содержащиеся в mClosed

procedure TClosed.GatesOn;
var Link: TLink;
begin
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 0; // дуга подключается нулём
      Link:= GetNext as TLink;
    end;
  end;
end;

// Метод используется при отладке

procedure TClosed.Print(var aFile: TextFile);
var L : TLink;
begin
  Writeln(aFile);  Writeln(aFile, '=== C L O S E D === ', mCost);
  L:= mClosed.GetFirst as TLink;;
  while Assigned(L) do begin
    Write(aFile, (L.mOwner as THamNode).mNode.GetName);
    Write(aFile, ' -> ');
    Writeln(aFile, (L.mDest as THamNode).mNode.GetName);
    L:= mClosed.GetNext as TLink;;
  end;
end;

//#######################################################################
//             Метод TGraph.GenHamilton
// Ускоренный приближённый метод отыскания Гамильтоновых циклов
//#######################################################################

function TGraph.GenHamilton(var aCost: integer): TBuffer;

var   HamGraph: THamGraph; // вспомогательный двудольный граф
(Dicotyledonous)
      Que : TSet;          // Приоритетная очередь клапанов (по неубыванию)
      BestCost: integer;   // лучшая цена на текущий момент
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Отключение дуг, переданных в множестве aLinks

  procedure LinksOff(aLinks: TSet);
  var Link: TLink;
  begin
    Link:= aLinks.GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // отключить дугу
      Link:= aLinks.GetNext as TLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
// Включение дуг, переданных в множестве aLinks

procedure LinksOn(aLinks: TSet);
var Link: TLink;
begin
  Link:= aLinks.GetFirst as TLink;
  while Assigned(Link) do begin
    Link.mHigh:= 0;  // включить дугу
    Link:= aLinks.GetNext as TLink;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Обработка объекта с клапанами aClosed
// (множеством отключенных и множествами отключаемых дуг)

procedure Handle(aClosed: TClosed);
var Cost: integer;     // прогнозируемая или точная стоимость решения
    Factors: integer; // количество факторов в очередном решении
    Start: THamNode;  // стартовая вершина кратчайшего фактора
    Closed: TClosed;  // новый объект с клапанами
    Links: TSet;      // множество откюченных дуг в текущий момент
    Node: THamNode;
begin
  // Отключить блокированные дуги объекта
  aClosed.GatesOff;

  // Цикл: поочерёдно отключатем дуги, ведущие внутрь фактора,
  // оцениваем стоимость соответствующих потоков,
  // и ставим в приоритетную очередь элементы TClosed
  // для частичных решений

  // Перебор множеств отключаемых дуг:
  Links:= aClosed.mFactors.GetFirst as TSet;
  while Assigned(Links) do begin
    // Отключаем дуги одной из вершин фактора, ведущие внутрь этого фактора
    LinksOff(Links);
    // При текущем состоянии клапанов определить стоимость потока
    Cost:= HamGraph.CalcCostAndFactors(Factors, Start);
    // Вновь подключаем дуги вершины, ведущие внутрь фактора
    LinksOn(Links);
    // Если поток существует, и он лучше текущего...
    if (Cost>=0) and (Cost< BestCost) then begin
      // Здесь поток существует и его стоимость ниже текущего минимума.
      // Если количество факторов=1, то запомнить лучший результат.
      if Factors=1 then begin
        BestCost:= Cost;            // лучшая цена
        aCost:= Cost;               // она же как результат
        // сохраняем последовательность вершин цикла в буфере результата
        HamGraph.SaveBest(Result);
      end else begin
        // Здесь количество факторов Factors > 1 (частичное решение),
        // то создаём и ставим в очередь множество закрытых клапанов
        Links.Add(aClosed.mClosed); // объединяем Links и mClosed
        Closed:= TClosed.Create(Cost, Links);
        // Обходим кратчайший фактор
        // и сохраняем в объекте Closed отключаемые дуги
        Node:= Start; // Начальная (любая) вершина фактора
        repeat
          // Функция THamNode.GenFactorsLinks создаёт множество дуг
          // ведущих внутрь текущего фактора
          Closed.InsertLinksSet(Node.GenFactorsLinks);
          Node:= Node.NextCircNode; // следующая вершина фактора
```

```
          until Node = Start;
          Que.Insert(Closed); // Вставляем объект в приоритетную очередь
        end;
      end;
      // Следующее множество отключаемых дуг:
      Links:= aClosed.mFactors.GetNext as TSet
    end;
    // Вернуть исходное состояние клапанов
    aClosed.GatesOn;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var  Closed: TClosed; // очередное множество закрытых клапанов

begin  { TGraph.GenHamilton }

  Result:= TBuffer.Create;  // создаём пустой буфер для хранения решения
  aCost:= -1;               // стоимость на случай отсутствия решения

  // Создаём вспомогательный двудольный граф HamGraph (Dicotyledonous)
  // с целью определения минимальной стоимости полного потока,
  // а также очередь клапанов Que (элементов типа TGate)

  HamGraph:= THamGraph.Create(Self);
  Que:= CreateSet;  // Приоритетная очередь клапанов по неубыванию стоимости
  BestCost:= MaxInt;// Минимальная стоимость на текущий момент

  // Создаём и вставляем в приоритетную очередь фиктивный элемент,
  // в котором множества закрытых и закрываемых линков пусты
  Closed:= TClosed.Create(0, nil);
  Closed.InsertLinksSet(CreateSet);
  Que.Insert(Closed);

  // Цикл обработки приоритетной очереди:
  while Que.GetCount > 0 do begin
    // Выбрать из очереди объект-множество клапанов
    Closed:= Que.GetFirst as TClosed;
    // Если его прогнозируемая цена меньше лучшей на данный момент,
    // то обработать это состояние клапанов
    if Closed.mCost < BestCost then Handle(Closed);
    // Удалить объект-множество клапанов
    Que.Delete(Closed);
    Closed.Free;
  end;

  // Очистка памяти:
  Que.Free;       // приоритетная очередь клапанов по неубыванию стоимости
  HamGraph.Free;  // вспомогательный двудольный граф (Dicotyledonous)
end;


//#######################################################################
// Быстрый приближённый алгоритм поиска гамильтонова цикла
// на основе поиска кратчайшей гамильтоновой цепи
// в неориентированном графе
//#######################################################################

function TGraph.GenHamilton_Quick(var aCost: integer): TBuffer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка очередной вершины:
  // возвращаются две ближайшие вершины (aY, aZ)
```

```
  // и сумма расстояний

  function FindX(aX : TNode; var aY, aZ : TNode): integer;
  var L: TLink;
      YVal, ZVal : integer; // расстояния к соседям
  begin
    aY:= nil; aZ:= nil;
    YVal:= MaxInt; ZVal:= MaxInt;
    // Перебор исходящих линков:
    L:= aX.OutLinkFirst;
    // Всегда YVal <= ZVal
    while Assigned(L) do begin
      if L.mValue <= YVal then begin
        ZVal:= YVal;
        YVal:= L.mValue;
        aZ:= aY; aY:= L.mDest;
      end else if L.mValue < ZVal then begin
        ZVal:= L.mValue;
        aZ:= L.mDest;
      end;
      L:= aX.OutLinkNext;
    end;
    // Возвращаем сумму расстояний к ближайшим соседям
    Result:= YVal + ZVal;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск вершины с минимальной суммой расстояний
  // к двум ближайшим соседям

  function FindNode(var aX, aY, aZ : TNode): integer;
  var X, Y, Z : TNode;
      Cost, BestCost: integer;
  begin
    aX:= nil; aY:= nil; aZ:= nil;
    BestCost:= MaxInt;
    X:= NodeFirst;
    while Assigned(X) do begin
      Cost:= FindX(X, Y, Z);
      if Cost < BestCost then begin
        BestCost:= Cost;
        aX:= X; aY:= Y; aZ:= Z;
      end;
      X:= NodeNext;
    end;
    Result:= BestCost;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var NodeX, NodeY, NodeZ : TNode;
    Cost, CostX : integer;

begin  { GenHamilton_Quick }

  // Для орграфа используем основной метод:
  if mDirect then begin
    Result:= GenHamilton(aCost);
    Exit;
  end;

  // Находим вершину X и двух её ближайших соседей Y, Z таких,
  // что сумма расстояний от X к соседям Y и Z минимальна
```

```
  CostX:= FindNode(NodeX, NodeY, NodeZ);

  // Строим кратчайшую цепь от вершины Y к вершине Z
  // с исключением вершины X

  Result:= GenHamPathStrip(NodeY, NodeZ, NodeX, Cost);
  if Cost < 0 then Exit;

  aCost:= Cost + CostX;  // общая стоимость цикла

  // Вставляем вершину в разрез между Z и Y, замыкая тем самым цикл
  Result.Push(NodeX);

  // Вращаем буфер, пока первая вершина графа не сместится в его начало
  NodeX:= NodeFirst;
  repeat
    NodeY:= Result.Get as TNode;
    if NodeY <> NodeX then Result.Put(NodeY);
  until NodeY = NodeX;
  Result.Push(NodeY);
end;


//#######################################################################
//                                                                      #
//                           Цепь Гамильтона                            #
// Кристофидес, стр 273,                                                #
// метод штрафования вершин модифицированый                             #
//                                                                      #
//#######################################################################

// Вспомогательная функция для вычисления стоимости линка
// с учётом штрафов инцидентных вершин

function CalcValue(aLink: TLink): integer;
begin
  with aLink do Result:= mValue + mOwner.mDist + mDest.mDist;
end;


type  // THamLink -- вспомогательный линк для построения остова

  THamLink = class (TItem)
    mValue: integer; // стоимость с учётом штрафов
    mLink: TLink;    // ссылка на исходный линк
    constructor Create(aLink: TLink);
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;


constructor THamLink.Create(aLink: TLink);
begin
  inherited Create;
  mLink:= aLink;
  // Условная стоимось формируется с учётом штрафов
  mValue:= CalcValue(aLink);
end;


function THamLink.Compare(arg: TItem): TCompare;
begin
```

```
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  with mLink do begin
    // Для неориентированного графа сравниваем исходную и конечную вершины
    // на предмет встречных линков
    if not mOwner.mOwner.mDirect and  // если не орграф
      (mOwner=(arg as THamLink).mLink.mDest) and
      (mDest=(arg as THamLink).mLink.mOwner)
      then begin
        // Здесь линки направлены встречно, отвергаем дубликат:
        Result:= cmpEq;    Exit;
      end;
  end;
  // Если линки не совпадают, то сортируем по неубыванию стоимости
  Result:= cmpLess;
  if mValue > (arg as THamLink).mValue then Result:= cmpGreate;
end;

// В орграфе выводим по направлению стрелок,
// а в графе - по алфавиту

procedure THamLink.Print(var aFile: TextFile);
var N1, N2 : TNode;
begin
  with mLink do begin
    N1:=mOwner;  N2:=mDest;
    if not mOwner.mOwner.mDirect and
       (mOwner.Compare(mDest) = cmpGreate) then begin
      N1:=mDest;   N2:=mOwner;
    end;
    Write(aFile, ' ' + N1.GetName + N2.GetName);
    Write(aFile, '=', mValue:3, '':1);
  end;
end;


//#######################################################################
// Поиск гамильтоновой цепи из вершины aStart в вершину aFin
//#######################################################################

function TGraph.GenHamPath(aStart,       // начальная вершины
                           aFin: TNode;  // конечная вершины
                           var aCost: integer  // длина цепи
                           ): TBuffer;
begin
  if mDirect
    // Для орграфа используется основной метод поиска гамильтонова цикла
    then Result:= GenHamPathDir(aStart, aFin, aCost)
    // Для неориентированного графа - метод штрафования верши
    // Кристофидес, стр 273,
    // метод штрафования верши модифицированый
    else Result:= GenHamPathStrip(aStart, aFin, nil, aCost);
end;


//#######################################################################
// Поиск гамильтоновой цепи из вершины aStart в вершину aFin
// в ориентированном графе
//#######################################################################

function TGraph.GenHamPathDir(aStart, aFin: TNode; var aCost: integer):
TBuffer;
```

```
var Closer,       // Замыкающая вершина
    Node: TNode;
begin
  Closer:= TNode.Create(0, Self);  // создаём вспомогат. замыкающую вершину
  InsertNode(Closer);              // вставляем её в граф
  Closer.MakeLink(aStart, 0);      // создаём связь Closer -> aStart
  aFin.MakeLink(Closer, 0);        // создаём связь aFin -> Closer
  Result:= GenHamilton(aCost);     // находим цикл Гамильтона
  // Удаляем из буфера результата замыкающую вершину
  repeat
    Node:= Result.Get as TNode;
    if Node = Closer then Break;
    Result.Put(Node);
  until false;
  aFin.RemoveLink(Closer);     // удаляем связь aFin -> Closer
  Closer.RemoveLink(aStart);   // удаляем связь Closer -> aStart
  RemoveNode(Closer);          // удаляем замыкающую вершину из графа
  Closer.Free;                 // уничтожаем вершину
end;


//#######################################################################
// Поиск гамильтоновой цепи из вершины aStart в вершину aFin
// с исключением вершины aStrip
// (если aStrip=nil, то без исключения)
// Кристофидес, стр 273,
// метод штрафования верши модифицированый
//#######################################################################

function TGraph.GenHamPathStrip(aStart,        // начальная вершина
                                aFin,          // конечная вершина
                                aStrip: TNode; // исключаемая вершина
                                var aCost: integer  // длина цепи
                                ): TBuffer;

// Поля вершин TNode используются так:
// mDist    -- текущий штраф
// mPower   -- текущая степень вершины в остовном дереве
// mColor   -- текущая окраска
// mRoot    -- принадлежность поддереву
// mFlow    -- для временного хранения приращения штрафа
// mLimit   -- обратный счётчик штрафования (до нуля)

var   Que : TSet; // очередь рёбер по неубыванию стоимости с учётом штрафов
     Tree : TSet; // множество чёрных рёбер покрывающего дерева
   Marked : TBuffer;  // очередь вершин для маркировки поддеревьев
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка штрафных полей mDist (вызывается единожды)

  procedure ClearFines;
  var Node: TNode;      // текущая вершина
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        mDist:= 0;     // штраф
        mLimit:= 1;    // ненулевой предел на штрафование вершины
        mRoot:= nil;   // корневая метка
      end;
      Node:= NodeNext;
    end;
  end;
```

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Возобновление счётчиков штрафования Node.mLimit
// Вызывается после уменьшения невязки или стоимости

procedure SetLimits(aLimit: integer);
var Node: TNode;
begin
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mLimit:= aLimit; // предел на штрафование вершины (счётчик)
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Очистка цвета вершин и рёбер, а также степеней вершин
// (вызывается на каждой итерации перед построением дерева)
// mColor = CWhite   -- цвет белый
// mPower = 0        -- степень вершины в дереве

procedure ResetNodesAndLinks;
var Node : TNode;
    Link : TLink;
begin
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mColor:= CWhite; // цвет
    Node.mPower:= 0;      // степень вершины в покрывающем дереве
    // Сбросить цвета рёбер
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      Link.mColor:= CWhite;
      Link:= Node.OutLinkNext;
    end;
    Node:= NodeNext;
  end;
  // Исключаемую вершину пометить красным:
  if Assigned(aStrip) then aStrip.mColor:= CRed;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Локальная процедура построения минимального покрывающего дерева
// Формирует множество чёрных рёбер дерева и помещает в множество Tree

function GenCover: boolean;

var Node : TNode;     // текущая вершина
    Link : TLink;     // для пометки встречного линка
      HL : THamLink;  // ребро для дерева
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление в очередь Que линков к соседним белым вершинам.
  // Линки HL сортируются в Que по неубыванию штрафной длины

  procedure AddTreeLinks(aNode: TNode);
  var  Link : TLink;
         HL : THamLink;
  begin
    // Обработка исходящих связей
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      // Вставить только линки, ведущие к белым вершинам
      if Link.mDest.mColor = CWhite then begin
        // Сконструировать вспомогательный линк:
        HL:= THamLink.Create(Link);
```

```
          // И вставить в очередь:
          if not Que.Insert(HL) then HL.Free;
        end;
        Link:= aNode.OutLinkNext;
      end;
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  var N : integer;  // количество присоединяемых вершин

  begin  { GenCover }
    Result:= true;
    // Предварительная очистка:
    Tree.ClrAndDestroy;    // дерево результата
    ResetNodesAndLinks;    // очистка вершин и рёбер: mColor = CWhite
    Node:= aStart;         // начинаем с исходной вершины
    Node.mColor:= CBlack; // красим чёрным
    // Присоединить все вершины графа кроме красной
    // (линков будет на единицу меньше)
    N:= mNodes.GetCount-1;
    if Assigned(aStrip) then Dec(N); // красную обойти стороной
    while Assigned(Node) and (Tree.GetCount < N) do  begin
      // Добавить в очередь Que линки к ближайшим белым вершинам:
      AddTreeLinks(Node);
      // Перебрать линки Que в поиске ближайшей белой вершины
      Node:= nil;                      // искомая вершина пока не найдена
      HL:= Que.GetFirst as THamLink;   // первый линк -- кратчайший
      while Assigned(HL) do begin
        Que.Delete(HL);                // удалить линк из очереди
        // Если линк ведёт к белой вершине, то ближайшая найдена
        with HL.mLink do if mDest.mColor = CWhite
            then Node:= mDest;    // приёмник связи
        if Assigned(Node) then begin
          // Здесь ближайшая белая вершина найдена:
          Node.mColor:= CBlack;      // присоединить её к множеству чёрных
          Tree.Insert(HL);           // вставить линк в результат (дерево)
          HL.mLink.mColor:= CBlack; // и отметить чёрным
          // Отметить чёрным и встречный линк:
          Link:= HL.mLink.GetReverse;
          if Assigned(Link) then Link.mColor:= CBlack;
          Break; // выход из while
        end else begin
          // Этот линк не ведёт к белой вершине, уничтожить его
          HL.Free;
          HL:= Que.GetNext as THamLink; // и взять следующий
        end;
      end; // while
    end; // while
    // Если остов не построен, то граф не связан, и решения не существует
    if Tree.GetCount < N then begin
      Tree.ClrAndDestroy;
      Result:= false;
    end; // if
    Que.ClrAndDestroy;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт степеней всех вершин, помещение их в поля TNode.mPower
  // и подсчёт невязки через количество листьев

  function CalcPowerNodes: integer;
  var HL : THamLink;    // текущий линк дерева Tree
      Node: TNode;
  begin
```

```
    // Предварительно очистить поля mPower
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mPower:= 0;  // степень вершины в покрывающем дереве
      Node:= NodeNext;
    end;
    // Перебор линков в дереве Tree:
    HL:= Tree.GetFirst as THamLink;
    while Assigned(HL) do begin
      // наращиваем степени вершин дерева :
      Inc(HL.mLink.mOwner.mPower);    // в источнике дуги
      Inc(HL.mLink.mDest.mPower);     // в приёмнике дуги
      HL:= Tree.GetNext as THamLink;
    end;
    // Подсчитать невязку как сумму листьев минус 2
    Result:=-2;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mPower=1 then Inc(Result);
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Маркировка поддерева из данной вершины (метим поле mRoot).
  // Дерево обходим в ширину по чёрным линкам.
  // Маркированные аершины красим серым.

  procedure Mark(aNode: TNode);
  var   Link : TLink;
        Node : TNode;
  begin
    Marked.Clear;          // очередь вершин для маркировки поддеревьев
    Marked.Put(aNode);     // занести в очередь корневую вершину
    aNode.mColor:= CGray;  // и окрасить серым
    // Пока очередь не пуста:
    while Marked.GetCount > 0 do begin
      Node:= Marked.Get as TNode;   // выбрать вершину из очереди
      Node.mRoot:= aNode;           // и отметить её в поле mRoot
      Link:= Node.OutLinkFirst;
      // Поместить в очередь соседние вершины, связанные чёрными линками:
      while Assigned(Link) do begin
        // Если линк чёрный
        if (Link.mColor = CBlack) and
           // и цвета инцидентных вершин не совпадают
           (Node.mColor <> Link.mDest.mColor) then begin
          // то присоединить соседа:
          Marked.Put(Link.mDest);     // в очередь
          Link.mDest.mColor:= CGray;  // и окрасить серым
        end;
        Link:= Node.OutLinkNext;
      end;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вычисление минимального штрафа в результате разрыва ребра aLink
  // Вызывается из CalcNodeFine
  // Возвращает штраф и признак успеха aStatus

  function CalcLinkFine(aLink: TLink; var aStatus: boolean): integer;
  var Link : TLink;
      Node : TNode;
      Fine : integer;
```

```
begin
  Result:= MaxInt;  aStatus:= false;
  // В текущий момент все вершины окрашены чёрным, а линки так:
  // принадлежащие дереву - чёрным, разорваный - серым, прочие - белым
  // Пометить вершины двух поддеревьев (поля mRoot)
  Mark(aLink.mOwner);  // поддерево mOwner
  Mark(aLink.mDest);   // поддерево mDest
  // Найти ребро минимального веса, соединяющее два поддерева
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // Восстановить цвет после маркировки (кроме красной вершины):
    if Node <> aStrip then Node.mColor:= CBlack;
    // Если это вершина поддерева mOwner и она не корневая
    if (Node.mRoot = aLink.mOwner) and (Node <> aLink.mOwner) then begin
      // то обрабатать её, перебирая исходящие линки
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Если ребро ведёт в другое поддерево,
        // то запомнить его минимальный вес с учётом штрафа:
        // Link.mDest.mRoot -- метка вершины
        // aLink.mDest -- корень оторванной части
        if Link.mDest.mRoot = aLink.mDest then begin
          Fine:= CalcValue(Link);
          if Result > Fine then begin
            Result:= Fine;  // предварительный результат
            aStatus:= true; // признак наличия результата
          end;
        end;
        Link:= Node.OutLinkNext;
      end;
    end;
    Node:= NodeNext;
  end;
  // Окончательный результат (приращение штрафа)
  // вычислить как разность
  if aStatus then Result:= Result - CalcValue(aLink);
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Подсчёт минимального приращения штрафа для данной вершины
// вызывается из CalcFines
// Возвращает штраф и признак успеха aStatus

function CalcNodeFine(aNode: TNode; var aStatus: boolean): integer;
var Link, Reverse: TLink;  // прямой и обратный линки
    Fine: integer;         // вычисленный штраф
    OK: boolean;           // признак успеха
begin
  Result:= MaxInt;  aStatus:= false;
  // Поочерёдно "разрываем" линки вершины, входящие в состав дерева
  // (чёрные линки) и строим обходные пути,
  // соединяющие разорванные части дерева
  Link:= aNode.OutLinkFirst;
  while Assigned(Link) and (Result > 0) do begin
    if Link.mColor = CBlack then begin
      // Этот линк принадлежит дереву
      // Временно "разорвать" его и встречный линк (красить серым)
      Link.mColor:= CGray;
      Reverse:= Link.GetReverse;
      if Assigned(Reverse) then Reverse.mColor:= CGray;
      // Вычислить штраф:
      aNode.OutPosPush;   // сохранить позицию перебора
      Fine:= CalcLinkFine(Link, OK); // вычислить штраф
```

```
        aNode.OutPosPop;     // восстановить позицию перебора
        // Вновь восстанавить цвета прямого и встречного линков
        Link.mColor:= CBlack;
        if Assigned(Reverse) then Reverse.mColor:= CBlack;
        // Запоминаем минимальное приращение штрафа
        if OK and (Result > Fine) then begin
          Result:= Fine;
          aStatus:= true;
        end;
      end;
      Link:= aNode.OutLinkNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт и установка штрафов для всех вершин
  // Возвращает количество оштрафованных вершин

  function CalcFines: integer;
  var Node: TNode;
      Fine: integer;
      OK: boolean;
  begin
    Result:=0; // для подсчёта оштрафованных вершин
    // Сканировать вершины, степень которых превышает 2,
    // вычислять приращения штрафов и временно помещать в поля mFlow
    // Node.mPower -- степень вершины в дереве
    // Node.mFlow -- сюда временно помещаем приращение штрафа
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= 0;          // mFlow -- приращение штрафа
      if (Node.mPower > 2)     // если это узел
         and (Node.mLimit > 0)// и лимит штрафования не исчерпан
        then begin
        Dec(Node.mLimit); // уменьшить лимит
        Inc(Result);      // нарастить счётчик оштрафованных
        // Вычисляить минимальный штраф, уменьшающий её степень:
        PosPush; // сохранить позицию перебора вершин
        Fine:= CalcNodeFine(Node, OK);
        PosPop;  // восстановить позицию перебора вершин
        if OK then begin
          if Fine = 0 then Fine:=1;  // реальный штраф больше нуля
          Node.mFlow:= Fine;         // mFlow -- приращение штрафа
        end;
      end; // if
      Node:= NodeNext;
    end; // while
    // Нарастить штрафы вершин mDist:= mDist + Node.mFlow
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Inc(Node.mDist, Node.mFlow);
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вычисление минимального приращения цены
  // в результате разрыва ребра aLink
  // Вызывается из CalcDeltaNode
  // Возвращает кратчайший белый линк, соединяющий листья

  function CalcDeltaLink(aLink: TLink; var aRes: TLink): integer;
  var Link : TLink;
      Node : TNode;
```

```
begin
  Result:= MaxInt;
  aRes:= nil;
  // В текущий момент все вершины окрашены чёрным,
  // линки дерева - чёрным, разорваный линк - серым, остальные - белым
  // Пометить вершины двух поддеревьев (поля mRoot)
  Mark(aLink.mOwner);  // поддерево mOwner
  Mark(aLink.mDest);   // поддерево mDest
  // Найти линк минимального веса, соединяющий два поддерева
  // через вершины со степенями 1 (листья)
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // Восстановить цвет после маркировки (кроме красной вершины)
    if Node <> aStrip then Node.mColor:= CBlack;
    // Если это лист в дереве mOwner
    if (Node.mRoot = aLink.mOwner) and (Node.mPower < 2)
      then begin
      // то обработать его
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Запомнить кратчайший линк к листу другого поддерева
        if (Link.mDest.mRoot = aLink.mDest)  // в другом поддереве
          and (Link.mDest.mPower < 2)        // и степень меньше 2-х
          and (Result > CalcValue(Link))     // и штрафная длина меньше
        then begin
            Result:= CalcValue(Link);  // промежуточный результат
            aRes:= Link;               // белый линк
        end; // if
        Link:= Node.OutLinkNext;
      end; // while
    end; // if
    Node:= NodeNext;
  end; // while
  // Окончательный результат вычислить как разность
  if Assigned(aRes) then Result:= Result - CalcValue(aLink);
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск подходящей пары линков для снижения степени узла aNode
// Вызывается из ForceCoverTree
// Возвращает два линка: aBlack -- разрываемый, aWhite -- замещающий

procedure FindBlackAndWhite(aNode: TNode; var aBlack, aWhite: TLink);
var Link : TLink;   // прямой линк
    White: TLink;   // белый линк
    Delta: integer; // текущее приращение
    Minim: integer; // минимальное приращение
begin
  aBlack:= nil;    aWhite:= nil;
  Minim:= MaxInt;
  // Поочерёдно "разрывать" линки вершины, входящие в состав дерева
  // (эти линки помечены чёрным)
  // и отыскивать обходные пути,
  // соединяющие разорванные части дерева через листья
  Link:= aNode.OutLinkFirst;
  while Assigned(Link) do begin
    if Link.mColor = CBlack then begin
      // Этот линк принадлежит дереву, разорвать его (красим серым)
      Link.mColor:= CGray;
      // При разрыве уменьшаются степени инцидентных вершин:
      with Link do begin
        Dec(mOwner.mPower);
        Dec(mDest.mPower);
```

```
        end;
        aNode.OutPosPush; // сохр. позицию перебора
        // Вычислить приращение цены
        Delta:= CalcDeltaLink(Link, White);
        aNode.OutPosPop;  // восст. позицию перебора
        // Восстанавить цвет линка
        Link.mColor:= CBlack;
        // При воссоединении чёрного линка
        // восст. степени инцидентных вершин:
        with Link do begin
          Inc(mOwner.mPower);
          Inc(mDest.mPower);
        end;
        // Запомнить минимальное приращение
        if Assigned(White) and (Delta < Minim) then begin
          Minim:= Delta;   // приращение
          aBlack:= Link;   // чёрный линк (в дереве)
          aWhite:= White;  // белый линк (вне дерева)
        end;
      end;
      Link:= aNode.OutLinkNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Удаление из дерева вспомогательных линков

  procedure Tree_Delete(aLink: TLink);
  var HL : THamLink; // линк в дереве Tree
      Rev: TLink;    // встречный линк
  begin
    // Уменьшить степени инцидентных вершин чёрного ребра aLink:
    Dec(aLink.mOwner.mPower);
    Dec(aLink.mDest.mPower);
    // Прямой и встречный линк метить белым
    aLink.mColor:= CWhite;
    Rev:= aLink.GetReverse;  // встречный линк
    if Assigned(Rev) then Rev.mColor:= CWhite;
    // Найти в дереве соответствующий вспомогательный линк
    HL:= Tree.GetFirst as THamLink;
    while Assigned(HL) do begin
      if (HL.mLink = aLink) or (HL.mLink = Rev) then Break;
      HL:= Tree.GetNext as THamLink;
    end;
    // Удалить его из дерева и ликвидировать
    if Assigned(HL) then begin
      Tree.Delete(HL);
      HL.Free;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вставка в дерево вспомогательных линков

  procedure Tree_Insert(aLink: TLink);
  var HL : THamLink;
      Rev: TLink;
  begin
    // Увеличить степени инцидентных вершин:
    Inc(aLink.mOwner.mPower);
    Inc(aLink.mDest.mPower);
    // Прямой и встречный линк метить чёрным
    aLink.mColor:= CBlack;
    Rev:= aLink.GetReverse;
```

```
    if Assigned(Rev) then Rev.mColor:= CBlack;
    // Сконструировать вспомогательный линк:
    HL:= THamLink.Create(aLink);
    // И вставить его в дерево:
    if not Tree.Insert(HL) then HL.Free;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Принудительное уменьшение степеней узлов остова

  procedure ForceCoverTree;
  var Node: TNode;
      Black, White : TLink;
  begin
    // Обработать узлы (вершины со степенями более 2)
    // Node.mPower -- степень вершины в дереве
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mPower > 2 then begin
        // Вычислить минимальный штраф, уменьшающий её степень
        PosPush; // сохранить позицию перебора вершин
        FindBlackAndWhite(Node, Black, White);
        PosPop;  // восстановить позицию перебора вершин
        // Если пара рёбер обнаружена
        if Assigned(Black) and Assigned(White) then begin
          // Видоизменить дерево
          Tree_Delete(Black);   // удалить из дерева разрываемый линк
          Tree_Insert(White);   // вставить в дерево соединяющий линк
        end;
      end;
      Node:= NodeNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Форсированное преобразование дерева в цепочку
  // с вычислением стоимости цепочки

  function Force(var aCost: integer): TBuffer;

    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Локальная функция для переноса цепи
    // из буфера aBuf в результат Result
    // с одновременным подсчётом стоимости цепи

    function MoveResult(aRes: TBuffer): integer;
    var HL : THamLink; // текущий линк дерева из Tree
    begin
      Result:= 0;
      aRes.Clear;
      // Перебор линков в покрывающем дереве:
      HL:= Tree.GetFirst as THamLink;
      while Assigned(HL) do begin
        aRes.Put(HL.mLink);
        Inc(Result, HL.mLink.mValue); // накопление стоимости
        HL:= Tree.GetNext as THamLink;
      end;
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  var  Delta : integer;  // невязка, вычисляемая по числу листьев

  begin { Force }
    // Повторять, пока невязка больше нуля
```

```
    repeat
      // Форсировано снизить степени "узлов"
      ForceCoverTree;
      // Подсчитать степени вершин и невязку через количество листьев
      Delta:= CalcPowerNodes;
    until Delta=0;
    // Перенести цепь из буфера Buf в результат Result
    // и подсчитать стоимость цепи aCost
    Result:= TBuffer.Create;
    aCost:= MoveResult(Result);  //
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Преобразование буфера результата:
  // линки заменяются цепочкой вершин от Start до Fin

  function ConvertLinksToNodes(aLinks: TBuffer): TBuffer;

    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Поиск в буфере aLinks следующей за aNode вершины

    function FindNext(aNode: TNode): TNode;
    var Link: TLink;
        i: integer;
    begin
      Result:= nil;
      // Круговое вращение буфера в поиске нужного линка
      for i:= 1 to aLinks.GetCount do begin
        Link:= aLinks.Get as TLink;
        // Если линк найден, то цикл прекращается
        with Link do begin
          if aNode = mOwner then begin
            Result:= mDest;  Break;
          end;
          if aNode = mDest then begin
            Result:= mOwner; Break;
          end;
        end;
        // Если не тот линк, то вернуть его в буфер
        aLinks.Put(Link);
      end;
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  var  Node: TNode;

  begin { ConvertLinksToNodes }

    Result:= TBuffer.Create;
    // Цепочка начинается с вершины aStart
    Node:= aStart;
    repeat
      Result.Put(Node);        // поместить в буфер
      Node:= FindNext(Node);   // и найти следующую
    until (Node = aFin) or not Assigned(Node);
    // Цепочка завершается вершиной aFin
    if Assigned(Node) then Result.Put(Node);
    aLinks.ClrAndDestroy;
    aLinks.Free;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    OK : boolean;        // признак связности графа
```

```
    Changed: boolean;   // признак снижения стоимости или невязки
    Delta : integer;    // невязка, вычисляемая по степеням вершин
    MinDelta : integer; // текущая минимальная невязка
    Nodes: integer;     // количество оштрафованных вершин
    Cost: integer;      // текущая стоимость
    BestCost: integer;  // стоимость лучшего промежуточного решения
    Res: TBuffer;       // очередное решение (цепь)
    FineLimit: integer; // предел штрафования вершины (счётчик)

begin  { TGraph.GenHamPath }

  aCost:= -1; // цена на случай отсутствия решения

  // Создать буфер для линков гамильтоновой цепи:
  Result:= TBuffer.Create;
  // Если это орграф, то выход:
  if mDirect then Exit;

  // Назначить начальное значение для счётчиков штрафования вершин
  FineLimit:= mNodes.GetCount;

  // Создать:
  Que:= CreateSet;          // множество для очереди линков
  Tree:= CreateSet;         // буфер для линков дерева
  Marked:= TBuffer.Create;  // очередь вершин для маркировки

  BestCost:= MaxInt;  // начальная лучшая стоимость

  // Очистить штрафные поля mFines:
  ClearFines;

  // Назначить максимальные штрафы (mDist) для крайних вершин:
  aStart.mDist:= MaxInt div 4;
  aFin.mDist:= aStart.mDist;

  MinDelta:= MaxInt;       // минимальная невязка
  Nodes:= 0;               // количество оштрафованных вершин

  // Повторять, пока существуют оштрафованные вершины
  repeat
    // Построить кратчайшее покрывающее дерево с учётом штрафов:
    OK:= GenCover;
    // Выход, если граф не связан:
    if not OK then Break;
    // Подсчитать невязку Delta как сумму листьев - 2
    Delta:= CalcPowerNodes;
    // Подсчитать новые штрафов
    // на основе текущих степеней вершин (полей mPower)
    // вернуть количество оштрафованных вершин Nodes
    Nodes:= CalcFines;
    Changed:= false;   // признак снижения стоимости или невязки
    // Искать форсированное решение
    Res:= Force(Cost);
    // Если оно лучше текущего, то запомнить, а иначе удалить
    if Cost < BestCost then begin
      Changed:= true;   // признак снижения стоимости или невязки
      BestCost:= Cost;  // запомнить лучшую стоимость
      aCost:= Cost;     // и текущую стоимость
      Result.Free;      // прежний результат удалить
      Result:= Res;     // а новый сохранить
    end else Res.Free;  // а иначе удалить новый результат
    // При уменьшении невязки
```

```
    if Delta < MinDelta then begin
      Changed:= true;        // признак снижения стоимости или невязки
      MinDelta:= Delta;      // запомнить минимальную невязку
    end;
    // Если уменьшилась стоимость или невязка
    if Changed then begin
      SetLimits(FineLimit); // возобновить количество попыток штрафования
      FineLimit:= 2 + 3*FineLimit div 4; // пересчитать лимит штрафования
    end;
  until Nodes = 0;  // прекратить, если штрафование не выполнялось
  // По окончании преобразовать буфер результата:
  // линки --> в вершины от Start ... до Fin
  if OK then Result:= ConvertLinksToNodes(Result);
  // Очистить память:
  Marked.Free;         // очередь вершин для маркировки
  Tree.ClrAndDestroy; // остовное дерево
  Tree.Free;
  Que.Free;            // очередь линков
end;

//#######################################################################
// GenHamPath_Full -- Поиск гамильтоновой цепочки полным перебором
//#######################################################################

function TGraph.GenHamPath_Full(aStart, aFin: TNode;
                              var aCost: integer): TBuffer;
var
    Cost: integer;    // текущая стоимость цикла
    Count: integer;   // счётчик окрашенных вершин
    Node: TNode;      // текущая вершина
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура сохранения текущего решения.
  // Вызывается после окраски всех вершин.
  // aFin - последняя окрашенная вершина
  procedure SaveResult;
  begin
    aCost:= Cost;          // сохраняем стоимость цикла
    Result.Clear;          // очищаем буфер результата
    Node:= aFin;           // заполняем буфер с последней вершины
    repeat
      Result.Push(Node);   // заносим в буфер в обратном порядке
      Node:= Node.mPred;   // предыдущая вершина
    until Node=aStart;     // пока не достигнем начальной вершины
    Result.Push(Node);     // заносим конечную вершину
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура поиска в глубину

  procedure Local(aNode: TNode);

  var L : TLink;  // текущий исходящий линк
  begin { Local }
    aNode.mColor:= CBlack;  // красим данную вершину
    Inc(Count);             // и наращиваем счётчик окрашенных
    // Все вершины (кроме последней) окрашены?
    if Count < mNodes.GetCount-1 then begin
      // Нет, ищем соседнюю неокрашенную перебором исходящих линков
      L:= aNode.OutLinkFirst;
      while Assigned(L) do begin
        if (L.mDest <> aFin) and (L.mDest.mColor = CWhite) then begin
          // Нашли неокрашенную и не последнюю:
          Inc(Cost, L.mValue);        // накапливаем стоимость
```

```
          if Cost < aCost then begin  // если стоимость ниже минимальной
            L.mDest.mPred:= aNode;    // то метим предыдущей вершиной
            Local(L.mDest);           // и рекурсивно вызываем эту же проц.
          end;
          Dec(Cost, L.mValue);        // восстанавливаем стоимость
        end;
        L:= aNode.OutLinkNext;        // следующий исходящий линк
      end;
    end else begin
      // Здесь Count = mNodes.GetCount-1 -- все вершины, кроме aFin, окрашены
      // Ищем замыкающий линк на вершину aFin
      L:= aNode.OutLinkFirst;
      while Assigned(L) and (L.mDest<>aFin)
        do L:= aNode.OutLinkNext;
      if Assigned(L) then begin
        // Замыкающий линк найден:
        Inc(Cost, L.mValue);      // накапливаем стоимость
        L.mDest.mPred:= aNode;    // метим предыдущей вершиной
        if Cost < aCost           // если она меньше минимальной
          then SaveResult;        // сохраняем цикл и его стоимость
        Dec(Cost, L.mValue);      // восстанавливаем стоимость
      end;
    end;
    // При выходе восстанавливаем счётчик вершин и цвет
    Dec(Count);
    aNode.mColor:= CWhite;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Temp: TBuffer;  // Для оценки начальной стоимости жадным алгоритмом

begin { TGraph.GenHamPath_Full }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;               // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;

  // В первом приближении начальную стоимость aCost
  // оцениваем жадным алгоритмом
  Temp:= GenHamPath_Greed(aStart, aFin, aCost, true);
  Temp.Free;

  ResetNodes;           // очистка вспомогательных полей
  Cost:= 0;             // накопленная стоимость = 0
  Count:= 0;            // счётчик окрашенных вершин = 0
  Local(aStart);        // вызов рекурсивной процедуры
  // Если путь не обнаружен, возвращаем минус 1
  if aCost = MaxInt then aCost:=-1;
end;


//#######################################################################
// GenHamPath_Greed -- Поиск гамильтоновой цепочки
// либо жадным методом (aGreed=true)
// либо случайным перебором (aGreed=false);
//#######################################################################

function TGraph.GenHamPath_Greed(aStart, aFin: TNode;
                                 var aCost: integer;
                                 aGreed: boolean): TBuffer;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение первого встретившегося белого линка
  function GetAnyLink(aNode: TNode): TLink;
  begin
    Result:= aNode.OutLinkFirst;
    while Assigned(Result) and
          (Result.mDest.mColor <> CWhite)
      do Result:= aNode.OutLinkNext;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение ближайшего белого линка
  function GetBestLink(aNode: TNode): TLink;
  var  Link: TLink;
       BestCost: integer;
  begin
    Result:= nil;
    BestCost:= MaxInt;
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      if (Link.mDest.mColor = CWhite) and
         (Link.mValue < BestCost) then begin
        BestCost:= Link.mValue;
        Result:= Link;
      end;
      Link:= aNode.OutLinkNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    Cost: integer;    // текущая стоимость цепи
    Count: integer;   // счётчик неокрашенных вершин
    Node: TNode;      // текущая вершина
    L : TLink;        // текущий линк

begin { GenHamPath_Greed }
  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                  // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  ResetNodes;            // очистка вспомогательных полей
  aFin.mColor:= CBlack; // красим конечную вершину
  Cost:= 0;              // накопленная стоимость = 0

  Count:= mNodes.GetCount-2; // счётчик неокрашенных вершин
  Node:= aStart;
  while Count > 0 do begin
    Node.mColor:= CBlack;
    Result.Put(Node);
    // перебираем соседние неокрашенные вершины
    if aGreed
      then L:= GetBestLink(Node) // ищем ближайшую неокрашенную
      else L:= GetAnyLink(Node); // ищем любую неокрашенную
    // Если неокрашенных нет, прервать цикл
    if not Assigned(L) then break;
    // Нашли неокрашенную:
    Inc(Cost, L.mValue);     // накапливаем стоимость
    Dec(Count);              // счётчик - 1
    Node:= L.mDest;          // следующая вершина
  end; // while

  if Count=0 then begin
    // Здесь пройдены все вершины, кроме последней
```

```
    Result.Put(Node);        // заносим в результат предпоследнюю
    L:= Node.GetLink(aFin); // линк из предпоследней в последнюю
    if Assigned(L) then begin
      Result.Put(aFin);
      Inc(Cost, L.mValue);  // накапливаем стоимость
      aCost:= Cost;         // возвращаем стоимость
    end else Result.Clear;  // если нет завершающего линка
  end;
end;

//#######################################################################
//                                                                      #
// Замкнутая задача коммивояжёра на ненасыщенном графе (цикл)     #
// TGraph.GenTravelCycle(var aCost: integer): TBuffer;                  #
//                                                                      #
// Разомкнутая задача коммивояжёра на ненасыщенном графе (путь)         #
// TGraph.GenTravelPath(aStart, aFin: TNode; var aCost: integer): TBuffer#
//                                                                      #
//#######################################################################

// Метод, дополняющий граф до насыщенного.
// Вновь созданным линкам назначается вес aValue (бесконечный)

procedure TGraph.MakeFull(aValue: integer);
var Node: TNode;
    N : TNode;
    L: TLink;
begin
  // Дополняем граф линками до насыщенного
  Node:= NodeFirst as TNode;
  // Перебор вершин
  while Assigned(Node) do begin
    Self.PosPush; // Сохр. позицию перебора
    N:= NodeFirst as TNode;
    while Assigned(N) do begin
      if N <> Node then begin
        // Получить линк на вершину Node --> N
        L:= Node.GetLink(N);
        // Если линк отсутствует, то создать его
        if not Assigned(L) then SetLink(Node, N, aValue);
      end;
      N:= NodeNext as TNode;
    end;
    Self.PosPop;  // Восст. позицию перебора
    Node:= NodeNext as TNode;
  end;
end;

// Процедура замены длинных линков
// короткими путями через другие вершины

procedure Decompress(aBuf: TBuffer);
var i: integer;
    Node, Next : TNode;
    Link: TFarLink;
begin
  Node:= aBuf.Get as TNode;
  for i:=1 to aBuf.GetCount do begin
    Next:= aBuf.Get as TNode;
    aBuf.Put(Node);
    repeat
      Link:= Node.GetFarLink(Next);
```

```
      if Link.mNodeNear = Next then Break;
      Node:= Link.mNodeNear;
      aBuf.Put(Node);
    until false;
    Node:= Next;
  end;
  aBuf.Put(Next);
end;

// Замена копий оригиналами.
// Вершины графа-копии, содержащиеся в aBuf,
// заменяются оригиналами из графа aGr

procedure ReplaceNodes(aGr: TGraph; aBuf: TBuffer);
var Node, N: TNode;
    i: integer;
begin
  for i:=1 to aBuf.GetCount do begin
    // Выборка исходной вершины Node
    Node:= aBuf.Get as TNode;
    // Получение копии N
    N:= aGr.GetNode(Node);
    // и засылка в тот же буфер
    aBuf.Put(N);
  end;
end;

// Замкнутая задача коммивояжёра на ненасыщенном графе (цикл)

function TGraph.GenTravelCycle(var aCost: integer): TBuffer;
var FullGraph : TGraph;
begin
  // Создаём копию данного графа
  FullGraph:= Self.Copy as TGraph;
  // Дополняем его до насыщенного,
  // причём добавочным линкам назначаем бесконечный вес
  FullGraph.MakeFull(MaxInt div 2);
  // Создаём карту маршрутов и сжимаем линки
  // заменяя длины дуг кратчайшими расстояниями
  FullGraph.Compress;
  // В насыщенном графе находим Гамильтонов цикл
  Result:= FullGraph.GenHamilton(aCost);
  // Заменяем в этом цикле длинные дуги кратчайшими путями
  Decompress(Result);
  // Заменяем копии вершин их оригиналами из Self
  ReplaceNodes(Self, Result);
  // Удаляем копию графа
  FullGraph.Free;
end;

// Разомкнутая задача коммивояжёра на ненасыщенном графе (путь)

function TGraph.GenTravelPath(aStart, aFin: TNode; var aCost: integer):
TBuffer;
var FullGraph : TGraph;
    Start, Fin : TNode;
begin
  // Создаём копию данного графа
  FullGraph:= Self.Copy as TGraph;
  // Дополняем его до насыщенного,
  // причём добавочным линкам назначаем бесконечный вес
  FullGraph.MakeFull(MaxInt div 2);
```

```
  // Создаём карту маршрутов и сжимаем линки
  // заменяя длины дуг кратчайшими расстояниями
  FullGraph.Compress;
  // В копии графа находим вершины начала и конца пути
  Start:= FullGraph.GetNode(aStart);
  Fin:= FullGraph.GetNode(aFin);
  // В насыщенном графе находим Гамильтонов путь
  Result:= FullGraph.GenHamPath(Start, Fin, aCost);
  // Заменяем в этом цикле длинные дуги кратчайшими путями
  Decompress(Result);
  // Заменяем копии вершин их оригиналами из Self
  ReplaceNodes(Self, Result);
  // Удаляем копию графа
  FullGraph.Free;
end;

//#######################################################################
//                                                                      #
// THugeGraph -- Огромный граф (узлы содержат числа типа Integer)       #
// Применяется как вспомогательный в других методах и при тестировании  #
//                                                                      #
//#######################################################################


// Конструктор случайного графа

constructor THugeGraph.GenRandom(aDir: boolean; // направленность
                      aLoadNodes, aLoadLinks,   // нагруженность
                      aNodes,                   // количество вершин
                      aLinks                    // плотность связей,%
                      : integer);
const CArr : array[boolean] of char = ('F','T');

var    nm: string;
        i: integer;
    Links: integer;
    nS,nD: integer;
     Link: TLink;

begin
  // Формируем имя графа:
  nm:='Huge: '+ CArr[aDir] +' : '+
         IntToStr(aLoadNodes)+ ' : '+
         IntToStr(aLoadLinks)+ ' : '+
         IntToStr(aNodes)+' : '+
         IntToStr(aLinks);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 1 to aNodes do begin
    InsertNode(TNodeInt.Create(i, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи:
  Links:= aNodes*(aNodes-1);  // предельное количество связей
  if aLinks < 100 {%}
    then Links:= Round(Links * aLinks / 100);

  while Links > 0 do begin
    nS:= 1+Random(aNodes);
```

```
    nD:= 1+Random(aNodes);
    if nS = nD then Continue;
    Link:= GetLinkByNumber(nS, nD);
    if not Assigned(Link) then begin
      MakeLink(nS, nD, 1+Random(aLoadLinks));
      Dec(Links);
      if not mDirect then Dec(Links);
    end;
  end;
end;


// Конструктор полного двудольного графа

constructor THugeGraph.GenDicoty(aLoadLinks, aPairs: integer);
var nm : string; // имя графа
     n : integer;
    N1, N2 : TNode;

begin
  // Формируем имя графа:
  nm:='Huge : Dicoty '+ IntToStr(aLoadLinks)+ ' : '+ IntToStr(2*aPairs);
  // Создаём пустой граф:
  inherited Create (nm,      // имя
                    true,    // орграф
                    true,    // вершины нагружены
                    true);   // дуги нагружены
  // создаём и вставляем вершины с номером и значением n
  for n:= 1 to 2*aPairs do begin
    mNodes.Insert(TNodeInt.Create(n, n, Self));
  end;
  // Создаём дуги из всех нечётных (левых) вершин во все чётные (правые)
  N1:= NodeFirst;
  while Assigned(N1) do begin
    if (N1.mValue mod 2) = 1 then begin
      PosPush;
      N2:= NodeFirst;
      while Assigned(N2) do begin
        if (N2.mValue mod 2) = 0
          then SetLink(N1, N2, 1+Random(aLoadLinks));
        N2:= NodeNext;
      end;
      PosPop;
    end;
    N1:= NodeNext;
  end;
end;


// Конструктор полного графа

constructor THugeGraph.GenFull(aDir: boolean; // направленность
                    aLoadNodes, aLoadLinks,   // нагруженность
                    aNodes                    // количество вершин
                    : integer);

const CArr : array[boolean] of char = ('F','T');
var i : integer;
    N1, N2 : TNode;
    nm : string;
begin
  // Формируем имя графа:
```

```
  nm:='Huge Full: '+ CArr[aDir] +' : '+
         IntToStr(aLoadNodes)+ ' : '+
         IntToStr(aLoadLinks)+ ' : '+
         IntToStr(aNodes);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 1 to aNodes do begin
    mNodes.Insert(TNodeInt.Create(i, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи между всеми вершинами графа
  N1:= NodeFirst;
  while Assigned(N1) do begin
    PosPush;
    N2:= NodeFirst;
    while Assigned(N2) do begin
      if N1 <> N2 then SetLink(N1, N2, 1+Random(aLoadLinks));
      N2:= NodeNext;
    end;
    PosPop;
    N1:= NodeNext;
  end;
end;


// Получение указателя на связь по номерам исходящей и входящей вершин

function THugeGraph.GetLinkByNumber(aSource, aDest: integer): TLink;
var S: TNodeInt;
begin
  Result:= nil;
  S:= GetNode(aSource);
  if not Assigned(S) then Exit;
  Result:= S.GetLinkByNumber(aDest);
end;


// Получение указателя на вершину по её номеру

function THugeGraph.GetNode(aNumber: integer): TNodeInt;
begin
  mNodes.PositionPush;
  Result:= TNodeInt(mNodes.GetFirst);
  while Assigned(Result) do begin
    if Result.mNumber = aNumber then Break;
    Result:= mNodes.GetNext as TNodeInt;
  end;
  mNodes.PositionPop;
end;


// Установка связи по номерам исходящей и входящей вершин

procedure THugeGraph.MakeLink(aSource, aDest, aVal: integer);
var S, D: TNode;
begin
  if GetLinkByNumber(aSource, aDest) <> nil then Exit;
  S:= GetNode(aSource);
  D:= GetNode(aDest);
```

```
  if Assigned(S) and Assigned(D)
    then SetLink(S, D, aVal)
end;


// Функция для создания огромного графа с заданными параметрами

function GenHugeGraph (aDir: boolean;          // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                 // количество вершин
                       aLinks                  // плотность связей,%
                       : integer): TGraph;
begin
  if aLinks >= 100 {%}
    then Result:= THugeGraph.GenFull(aDir,          // направленность
                             aLoadNodes, aLoadLinks, // нагруженность
                             aNodes                  // количество вершин
                             )
    else Result:= THugeGraph.GenRandom(aDir,        // направленность
                             aLoadNodes, aLoadLinks, // нагруженность
                             aNodes,                 // количество вершин
                             aLinks                  // плотность связей,%
                             );
end;


end.
```

Приложение H
Модуль GrChars

```
unit GrChars;
{$I Common.inc}
//*********************************************************************
//    Модуль генрации графов, вершины которого обозначены символами      *
//                                                                       *
// Содержит объекты:                                                     *
// TNodeChar -- вершина-символ                                           *
// TGraphChars  -- Граф, сотавленный из вершин-символов                  *
//*********************************************************************

interface

uses Root, Graph;

type

  // TNodeChar -- вершина-символ
  TNodeChar = class (TNode)
    private
      function GetLinkByName(aName : char): TLink;
    public
      mName  : char;  // хранимый символ
      constructor Create(aName: char; aVal: integer; aOwner: TGraph);
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;

  // TGraphChars  -- Граф, сотавленный из вершин-символов
  TGraphChars = class (TGraph)
    private
      function  GetLinkByName(aSource, aDest : char): TLink;
      procedure MakeLink(aSource, aDest : char; aVal : integer);
    public
      constructor Load(const aName: String);
      constructor GenRandom(aDir: boolean;
                            aLoadNodes, aLoadLinks,
                            aNodes, aLinks : integer);
      constructor GenFull(aDir: boolean;
                          aLoadNodes, aLoadLinks, aNodes : integer);
      procedure LoadFlowData(const aName: String);
      procedure Save(const aName: String); override;
      function  GetNode(aName : char): TNodeChar;
  end;

// Генерация либо полного, либо неполного случайных графов

function GenCharsGraph(aDir: boolean;          // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                 // количество вершин
                       aLinks                  // плотность связей,%
                       : integer): TGraph;

//////////////////////////////////////////////////////////////////////////
implementation
uses SysUtils;
//////////////////////////////////////////////////////////////////////////
```

```
// Генерация либо полного, либо неполного случайных графов

function GenCharsGraph(aDir: boolean;          // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                 // количество вершин
                       aLinks                  // плотность связей,%
                       : integer): TGraph;
begin
  if aLinks >= 100 {%}
    then Result:= TGraphChars.GenFull(aDir,          // направленность
                            aLoadNodes, aLoadLinks, // нагруженность
                            aNodes                  // количество вершин
                            )
    else Result:= TGraphChars.GenRandom(aDir,        // направленность
                            aLoadNodes, aLoadLinks, // нагруженность
                            aNodes,                 // количество вершин
                            aLinks                  // плотность связей,%
                            );
end;

////////////////////////////////////////////////////////////////////////////
// TNodeChar -- вершина-символ
////////////////////////////////////////////////////////////////////////////

constructor TNodeChar.Create(aName: char; aVal: integer; aOwner: TGraph);
begin
  inherited Create(aVal, aOwner);
  mName:= aName;
end;


function TNodeChar.Copy: TItem;
begin
  Result:= TNodeChar.Create(mName, mValue, mOwner);
end;

// Поиск линка по имени смежной вершины

function TNodeChar.GetLinkByName(aName: char): TLink;
begin
  OutPosPush;
  Result:= OutLinkFirst;
  while Assigned(Result) do begin
    if (Result.mDest as TNodeChar).mName = aName then Break;
    Result:= OutLinkNext;
  end;
  OutPosPop;
end;

function TNodeChar.GetName: string;
begin
  Result:= mName;  // возвращаем символ - название вершины
end;

////////////////////////////////////////////////////////////////////////////
// TGraphChars -- граф с вершинами-символами
////////////////////////////////////////////////////////////////////////////

// Конструктор полного графа

constructor TGraphChars.GenFull(aDir: boolean;
                                aLoadNodes, aLoadLinks, aNodes: integer);
```

```
const CArr : array[boolean] of char = ('F','T');
var i : integer;
    N1, N2 : TNode;
    name: Char;
    nm : string;
begin
  // Формируем имя графа:
  nm:='Full: '+ CArr[aDir] +' : '+
          IntToStr(aLoadNodes)+ ' : '+
          IntToStr(aLoadLinks)+ ' : '+
          IntToStr(aNodes);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 0 to aNodes-1 do begin
    name:= Char(Ord('A')+i);
    InsertNode(TNodeChar.Create(name, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи между всеми вершинами графа
  N1:= NodeFirst;
  while Assigned(N1) do begin
    PosPush;
    N2:= NodeFirst;
    while Assigned(N2) do begin
      if N1 <> N2 then SetLink(N1, N2, 1+Random(aLoadLinks));
      N2:= NodeNext;
    end;
    PosPop;
    N1:= NodeNext;
  end;
end;

// Конструктор случайного графа
// Если aNodes < 0, то число вершин точное, а иначе случайное
// Если aLinks < 0, то плотность графа (%) точная, а иначе случайная

constructor TGraphChars.GenRandom(
                            aDir: boolean;  // признак орграфа
                            aLoadNodes,     // нагруженность вершин
                            aLoadLinks,     // нагруженность связей
                            aNodes,         // предельное число вершин
                            aLinks:         // предельное число связей, %
                            integer);

const CArr : array[boolean] of char = ('F','T');

var Nodes, Links : integer;
    i, Limit : integer;
    Link: TLink;
    name, nS, nD: Char;
    nm : string;

begin
  if aNodes > 0
    then  // кол-во вершин случайное
      Nodes:= 2 + aNodes div 2 + Random(aNodes div 2)
    else  // кол-во вершин точное
      Nodes:= Abs(aNodes);
  Limit:= Nodes*(Nodes-1) div 2;  // предельное количество связей
```

```
  // кол-во связей
  if aLinks > 0 then begin
    // Количество связей случайно
    Links:= aLinks;
    if Links > 100 then Links:= 100;
    Links:= Limit div 4 + Random((aLinks*Limit) div 100);
  end else begin
    // Количество связей точно
    Links:= Abs(aLinks);
    if Links > 100 then Links:= 100;
    Links:= (Links*Limit) div 100;
  end;

  // Формируем имя графа:
  nm:='Random: '+ CArr[aDir] +' : '+
         IntToStr(aLoadNodes)+ ' : '+
         IntToStr(aLoadLinks)+ ' : '+
         IntToStr(Nodes)+' : '+
         IntToStr(Links);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 0 to Nodes-1 do begin
    name:= Char(Ord('A')+i);
    InsertNode(TNodeChar.Create(name, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи
  repeat
    nS:= Char(Ord('A')+Random(Nodes));
    nD:= Char(Ord('A')+Random(Nodes));
    if nS = nD then Continue;
    Link:= GetLinkByName(nS, nD);
    if not Assigned(Link) then begin
      MakeLink(nS, nD, 1+Random(aLoadLinks));
      Dec(Links);
    end;
  until Links <= 0;
end;

{                 Структура взвешенного графа
Gragh - 2
0 - граф(0), орграф(1)
1 - нагруженность вершин
1 - нагруженность рёбер (дуг)
3 - количество вершин
A=1 B=3 C=2
A -> B=5 C=7
B -> A=5 C=6
C -> A=7 B=6
}

constructor TGraphChars.Load(const aName: String);
const CharsSet = ['a'..'z', 'A'..'Z'];
var F: TextFile;
    nodes, n, val : integer;
    nm : string;        // имя графа
    name : char;        // имя узла = 'A'..'Z'
    Dir, LN, LL : boolean;
    S, D : char;        // имя источника и приёмника связи
```

```
    node : TNodeChar;  // очередной узел
    err : boolean;     // признак ошибки
begin
  if not SysUtils.FileExists(aName) then begin
    Fail;
    Exit;
  end;
  err:= false; n:= 0;
  AssignFile(F, aName); Reset(F);

  if not Eof(F) then Readln(F, nm) else err:= true;
  if not Eof(F) then Readln(F, n) else err:= true;
  Dir:= n<>0;   // признак орграфа
  if not Eof(F) then Readln(F, n) else err:= true;
  LN:= n<>0;    // признак нагруженных вершин
  if not Eof(F) then Readln(F, n) else err:= true;
  LL:= n<>0;    // признак нагруженных связей

  inherited Create(nm, Dir, LN, LL);  // создание пустого графа

  // Количество вершин:
  if not Eof(F) then Readln(F, n) else err:= true;
  nodes:= n;

  // Чтение вершин
  while not err and not Eof(F) and (n>0) do begin
    name:= '0'; val:= 1;
    Dec(n);
    // чтение имени (a..z, A..Z)
    while not Eoln(F) do begin
      Read(F, name);
      if name in CharsSet then Break;
    end;
    // чтение веса вершины
    if mLoadNodes then begin
      Read(F, S);      // пропускаем знак =
      Read(F, val);    // чтение веса
    end;
    node:= TNodeChar.Create(name, val, Self);
    if not InsertNode(node)
      then node.Free;  // если пытались повторно, удаляем дубликат
  end;
  Readln(F);

  if n>0 then err:= true;  // не все узлы прочитаны

  // Чтение связей
  while not err and not Eof(F) do begin
    val:= 1;
    S:='0'; D:='0';
    // чтение вершины-источника
    while not Eoln(F) do begin
      Read(F, S);
      if S in CharsSet then Break;
    end;
    // чтение вершин-приёмников
    while not Eoln(F) do begin
      // чтение имени
      while not Eoln(F) do begin
        Read(F, name);
        D:= name;
        if D in CharsSet then Break;
```

```
      end;
      if not (D in CharsSet) then Break;
      // Для нагруженных связей:
      if mLoadLinks then begin
         Read(F, name);    // пропускаем знак =
         Read(F, val);     // чтение веса
      end;
      MakeLink(S, D, val);
    end;  // while
    Readln(F);
    Dec(nodes);
    if nodes=0 then Break;  // если все вершины
  end;  // while
  Close(F);

  if err then begin
    if Assigned(mNodes) then begin
      mNodes.ClrAndDestroy;
      mNodes.Free;
    end;
    Fail;
  end;
end;

// Чтение данных к задачам на потоках
// Формат входных данных:
{
FLOW: -- стартовый маркер
XY = mL mH  -- дуга, минимальный поток, максимальный поток
...
}
procedure TGraphChars.LoadFlowData(const aName: String);
var F: TextFile;
    S: string;
    C1, C2: char;
    Low, High : integer;
    L: TLink;
begin
  AssignFile(F, aName); Reset(F);
  // Поиск стартового маркера
  while not Eof(F) do begin
    Readln(F,S);
    if Pos('FLOW',S)<>0 then Break;
  end;
  while not Eof(F) do begin
    // C1, C2 -- имена источника и приёмника дуги
    C1:='-'; C2:='-';
    if not Eoln(F) then Read(F, C1) else begin Readln(F); Continue end;
    if not Eoln(F) then Read(F, C2) else begin Readln(F); Continue end;
    L:= GetLinkByName(C1, C2);
    if Assigned(L) then begin
      while not Eoln(F) do begin
        Read(F,C1);
        if C1='=' then Break;
      end;
      if not Eoln(F) then Read(F, Low) else begin Readln(F); Continue end;
      if not Eoln(F) then Read(F, High) else begin Readln(F); Continue end;
      // Сохраняем данные о потоке
      L.mLow:= Low;
      L.mHigh:= High;
    end;
    Readln(F);
```

```
  end;
  Close(F);
end;

// Сохранение графа в текстовом файле

procedure TGraphChars.Save(const aName: String);
var F: TextFile;
    i, j: integer;
    N: TNodeChar;
    L: TLink;

begin
  AssignFile(F, aName);
  if FileExists(aName) then Append(F) else Rewrite(F);
  Writeln(F, mName);
  Writeln(F, Ord(mDirect), ' - тип графа (1 = орграф)');
  Writeln(F, Ord(mLoadNodes), ' - вершины (1 = нагруженные)');
  Writeln(F, Ord(mLoadLinks), ' - дуги (1 = нагруженные)');
  Writeln(F, mNodes.GetCount, ' - количество вершин');

  for i:= 1 to mNodes.GetCount do begin
    N:= mNodes.GetItem(i) as TNodeChar;
    if mLoadNodes
      then Write(F, N.mName+'=', N.mValue, ' ')
      else Write(F, N.mName, ' ');
  end;
  Writeln(F);

  for i:= 1 to mNodes.GetCount do begin
    N:= TNodeChar(mNodes.GetItem(i));
    Write(F, N.mName,' -> ');
    for j:= 1 to N.OutGetCnt do begin
      L:= N.OutGetLink(j);
      if N.mOwner.mLoadLinks
        then Write(F, (L.mDest as TNodeChar).mName, '=', L.mValue:2,' ')
        else Write(F, (L.mDest as TNodeChar).mName,' ');
    end;
    Writeln(F);
  end;
  Close(F);
end;

// Получение указателя на связь по именам исходящей и входящей вершин

function TGraphChars.GetLinkByName(aSource, aDest: char): TLink;
var S: TNodeChar;
begin
  Result:= nil;
  S:= GetNode(aSource);
  if not Assigned(S) then Exit;
  Result:= S.GetLinkByName(aDest);
end;

// Получение указателя на вершину по её имени

function TGraphChars.GetNode(aName: char): TNodeChar;
begin
  mNodes.PositionPush;
  Result:= TNodeChar(mNodes.GetFirst);
  while Assigned(Result) do begin
    if Result.mName = aName then Break;
```

```
    Result:= mNodes.GetNext as TNodeChar;
  end;
  mNodes.PositionPop;
end;

// Установка связи по именам исходящей и входящей вершин

procedure TGraphChars.MakeLink(aSource, aDest : char; aVal: integer);
var S, D: TNode;
begin
  if GetLinkByName(aSource, aDest) <> nil then Exit;
  S:= GetNode(aSource);
  D:= GetNode(aDest);
  if Assigned(S) and Assigned(D)
    then SetLink(S, D, aVal)
end;


end.
```