БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Теория алгоритмов

Допущено
Министерством образования Республики Беларусь
в качестве учебного пособия для магистрантов
учреждений высшего образования по специальностям
«Прикладная математика и информатика»,
«Теоретические основы информатики»

МИНСК
БГУ
2013

УДК 519.712(075.8)
ББК 22.18я73-1
Т11

А в т о р ы:
**П. А. Иржавский, В. М. Котов, А. Ю. Лобанов,
Ю. Л. Орлович, Е. П. Соболевская**

Р е ц е н з е н т ы:
доктор педагогических наук *В. В. Казачёнок*;
доктор педагогических наук *О. И. Мельников*

Т11 **Теория** алгоритмов : учеб. пособие / П. А. Иржавский [и др.]. — Минск : БГУ, 2013. — 159 с.
ISBN 978-985-518-857-6.

В учебном пособии изложены современные методы построения и анализа алгоритмов с использованием эффективных способов хранения, представления и преобразования информации.

Для магистрантов учреждений высшего образования, обучающихся по специальностям «Прикладная математика и информатика» и «Теоретические основы информатики».

УДК 519.712(075.8)
ББК 22.18я73-1

ISBN 978-985-518-857-6

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Быстрая обработка огромных объёмов информации, возникающих во многих прикладных областях (вычислительная биология и биоинформатика, вычислительная геометрия, автоматизация проектирования сложных технических систем и др.), на современном уровне развития компьютерной техники невозможна без применения эффективных алгоритмов, в основе которых лежит использование специальных структур данных. От правильного выбора структуры данных во многом зависит трудоёмкость решения задачи. Целью данного пособия является изложение современных подходов к разработке и анализу алгоритмов с использованием различных структур данных, а его актуальность обусловлена большой потребностью в подготовке специалистов, обладающих знаниями о современных методах, которые уже нашли отражение в учебных пособиях ведущих университетов мира.

Предполагается, что читатели получили начальные навыки, необходимые для проектирования и анализа алгоритмов [1, 2]. В частности, они должны быть знакомы с основными понятиями, используемыми при разработке алгоритмов (размерность задачи, трудоёмкость алгоритма, полиномиальный и экспоненциальный алгоритм), базовыми алгоритмами внутренней сортировки, простейшими структурами данных (список, стек, очередь, приоритетная очередь, множества, сбалансированные поисковые деревья) и базовыми алгоритмами на графах.

В первых двух главах пособия представлены такие современные структуры данных, как дерево интервалов, бор и суффиксный массив. Несмотря на то, что они малоизвестны, их применение при разработке эффективных алгоритмов нередко становится незаменимым решением. Структура данных дерево интервалов успешно применяется в вычислительной геометрии (обработка элементов непрерывной подпоследовательности одинаковым образом, быстрое вычисление различных функций от набора элементов последовательности), при разработке баз

данных, а также в различных алгоритмах на графах, которые, как и алгоритмы вычислительной геометрии, широко используются для решения прикладных задач. Структуры данных бор и суффиксный массив актуальны, в частности, при обработке огромных массивов данных, которые возникают, например, при решении задач поиска заданной строки или множества строк в тексте.

В третьей главе рассматриваются графовые модели, получившие широкое применение в разнообразных областях: экономике, физике, химии, молекулярной биологии, программировании, логистике, сетевом и календарном планировании, проектировании интегральных схем и сетей оптико-волоконной связи, распределении ресурсов и многих других. Язык теории графов позволяет построить математическую модель достаточно сложных задач и предоставляет эффективный инструмент для их решения.

Особенностью пособия является наличие доказательств корректности и эффективности предлагаемых подходов. Издание содержит достаточное число примеров и рисунков, облегчающих самостоятельное изучение изложенного материала. Каждая глава завершается списком задач для самостоятельного решения.

В пособии знаком $\square$ отмечено окончание доказательства теоремы или леммы. В приводимых псевдокодах запись $\boldsymbol{A \leftarrow B}$ имеет смысл присвоения A значения B, запись $\boldsymbol{A \leftrightarrows B}$ — обмена значениями A и B, т. е. одновременное присвоение A значения B, а B — прежнего значения A, символ $\varnothing$ наравне с общепринятым обозначением пустого множества также заменяет несуществующий объект (вершина, дуга, путь и др.). Тело цикла — последовательность действий между командой цикла и первой командой, расположенной не правее неё. В цикле **Пока** $\boldsymbol{C}$ тело исполняется, пока условие C не нарушится (возможно, ни разу). В цикле **Для всех** $\boldsymbol{x}$ **от** $\boldsymbol{a}$ **до** $\boldsymbol{b}$ тело исполняется первый раз для $x = a$, а каждый следующий — для x со значением на единицу ближе к b, чем предыдущее. Величина b может быть больше, меньше или равна величине a, при равенстве тело цикла исполняется ровно один раз. В цикле **Для всех** $\boldsymbol{x}$ **из** $\boldsymbol{S}$ тело исполняется при $x = y$ для всех различных $y \in S$, порядок не имеет значения. Выражение **Начать**

следующую итерацию цикла означает переход к началу наиболее внутреннего из активных циклов со следующим значением переменной цикла. Аналогично телу цикла определяются ветви условного оператора **Если** C. Ветвь, следующая непосредственно за условным оператором, выполняется, если и только если соблюдается условие C. После неё может располагаться команда **Иначе**, после которой идёт альтернативная ветвь, исполняемая, если и только если условие C нарушено. Все условия представляют собой логические выражения с использованием булевых переменных, функций и логических операторов **и** и **или**. При этом сначала вычисляются все выражения между логическими операторами, затем применяются операторы **и**, после чего — операторы **или**. Порядок действий может меняться при наличии скобок. Команда **Вернуть** x означает, что вычисление функции прекращается, а результатом полагается значение x. После символов // до конца строки следует комментарий. Все объекты являются персистентными (постоянными), т. е. изменения аргументов внутри функции никак не влияют на значения переданных объектов в точке вызова, несмотря на то, что объект не копируется. В псевдокоде отождествляются функция натурального аргумента $a(i)$ и последовательность $(a_i)_{i=1}^n$.

Авторы учебного пособия профессор Котов Владимир Михайлович и доцент Соболевская Елена Павловна в 2012 г. стали лауреатами премии имени А. Н. Севченко в номинации «Образование» за цикл пособий по дискретной математике, проектированию и анализу алгоритмов. Доцент Орлович Юрий Леонидович является автором более 60 научных работ по теории графов и дискретной оптимизации. Ассистенты Иржавский Павел Александрович и Лобанов Алексей Юрьевич имеют большой опыт участия в соревнованиях по программированию разного уровня, включая международные (первый награжден бронзовой медалью на Международном соревновании по программированию среди студентов (ACM ICPC), второй — бронзовой медалью Международной олимпиады по информатике среди школьников (IOI)), неоднократно были награждены премией специального фонда Президента Республики Беларусь по социальной поддержке одарённых учащихся и студентов с вручением нагрудного знака лауреата.

Пособие предназначено для магистрантов, а также может быть полезно студентам высших учебных заведений, программы которых предусматривают изучение дисциплин по теории алгоритмов и программированию. Пособие несомненно будет интересно всем, кто стремится к углублению своих знаний в области алгоритмики.

Глава 1. ДЕРЕВО ИНТЕРВАЛОВ

Во многих задачах вычислительной геометрии возникает необходимость обработки элементов непрерывной подпоследовательности одинаковым образом, а также быстрого вычисления различных функций от набора элементов последовательности (см. [3, гл. 8]). В борьбе за скорость в теории и на практике структура данных *дерево интервалов* нередко становится незаменимым решением. Помимо этого, дерево интервалов успешно применяется при разработке баз данных, а также в различных алгоритмах на графах, которые, как и алгоритмы вычислительной геометрии, широко используются для решения прикладных задач.

В этой главе структура данных дерево интервалов будет рассмотрена на примере подсчёта суммы и минимума на интервале с возможностью увеличения всех чисел в интервале на одну и ту же величину и установления всем числам в интервале одинакового значения. Также будут приведены комментарии о возможностях применения дерева интервалов в других случаях.

1.1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Пусть задана последовательность чисел $(a_i)_{i=0}^{N-1}$.

Определение 1.1. *Интервальный запрос на* $[i, j]$ — операция, производимая с элементами $a_i, a_{i+1}, \ldots, a_j$.

При работе с целочисленными индексами непустой интервал (i, j) совпадает с полуинтервалами $[i + 1, j)$ и $(i, j - 1]$, а также отрезком $[i + 1, j - 1]$, поэтому в дальнейшем, говоря об интервалах индексов, будем использовать наиболее удобное из этих обозначений.

Здесь и далее речь будет идти о большом числе запросов, когда непосредственное исполнение каждого из них становится непозволительно медленным.

Определение 1.2. *Интервальный подсчёт на* $[i, j]$ — интервальный запрос на $[i, j]$, в результате которого вычисляется значение некоторой функции от значений элементов $a_i, a_{i+1}, \ldots, a_j$.

Напомним, что *мультимножество* является обобщением множества, допускающим повторение равных элементов. Иначе говоря, мультимножество — упорядоченная пара (A, m), где A — множество, а $m\colon A \to \mathbb{N}$ — функция, ставящая в соответствие каждому элементу x множества A некоторое положительное целое число $m(x)$, или *кратность* элемента x. Для всех элементов, не принадлежащих A, кратность полагается равной нулю. Пара мультимножеств A и B называется *разбиением* мультимножества C (обозначается $C = A \uplus B$), если кратность каждого элемента C равна сумме его кратностей в A и B. Например, $\{1, 1, 1, 2, 2, 3\} = \{1, 1, 2, 2\} \uplus \{1, 3\}$.

Искомой функцией может быть, например, сумма значений элементов либо их минимум (максимум). В общем случае будем полагать, что значение функции от элементов мультимножества M может быть эффективно вычислено на основе значений функции от элементов мультимножеств M_1 и M_2, являющихся разбиением мультимножества M. В ряде случаев этого можно добиться расширением функции за счёт дополнительных значений. Далее будем полагать, что описанная *операция перевычисления* значения функции выполняется за константное время. Искомую функцию будем обозначать f, а функцию перевычисления — F. Тогда $f(M) = F\big(f(M_1), f(M_2)\big)$, если $M = M_1 \uplus M_2$. Так, если f — сумма/минимум элементов мультимножества, то F — сумма/минимум двух чисел.

Отметим, что значение искомой функции также может зависеть от порядка элементов последовательности. В таком случае дерево интервалов по-прежнему применимо, однако некоторые из приведённых далее алгоритмов потребуют незначительных изменений, которые мы оставим читателю в качестве упражнения.

Определение 1.3. *Одиночное изменение* — изменение одного из элементов последовательности.

Если все запросы заведомо являются запросами интервального подсчёта, можно выполнять каждый из них за $O(\log N)$ операций с пред-

вычислением значения функции на $O(N \log N)$ интервалах (с использованием такого же количества памяти и операций) с длинами, равными степеням двойки. В случае подсчёта как суммы, так и минимума можно уменьшить время обработки запроса до $O(1)$, а время предобработки и размер используемой памяти — до $O(N)$. Тем не менее даже одиночное изменение последовательности приведёт к необходимости обновлять $\Theta(N)$ предпросчитанных значений, что делает такой подход малоэффективным.

Определение 1.4. *Интервальное изменение на* $[i, j]$ — интервальный запрос на $[i, j]$, в результате которого определённым образом изменяются значения элементов этого интервала.

Простыми примерами интервальных изменений являются увеличение значения всех элементов на одну и ту же константу и установление всем элементам одного значения.

Определение 1.5. *Дерево интервалов* для последовательности из N элементов — двоичное дерево, удовлетворяющее следующему набору условий:

- каждая вершина соответствует некоторой непрерывной подпоследовательности (иными словами, интервалу индексов элементов последовательности);
- корень дерева соответствует всей последовательности (или полуинтервалу $[0, N)$);
- листья соответствуют элементам последовательности (или интервалам вида $[i, i]$);
- интервалы, которым соответствуют сыновья вершины, не пересекаются, а их объединение равно интервалу, соответствующему самой вершине;
- высота дерева минимально возможная (равна $\lceil \log_2 N \rceil$).

Соответствие вершины элементам последовательности подразумевает, что значение ключа вершины равно искомой функции от этих элементов.

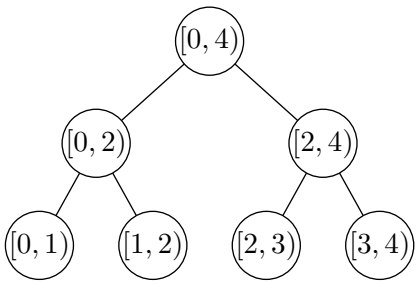


Рис. 1.1. Соответствие вершин индексам

На рис. 1.1 показано соответствие вершин полуинтервалам индексов элементов

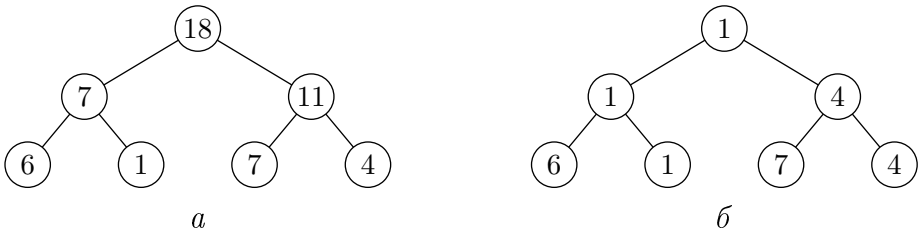


Рис. 1.2. Ключи вершин:
а — для вычисления суммы; *б* — для вычисления минимума

последовательности, а на рис. 1.2 приведены примеры возможных ключей вершин.

1.2. ХРАНЕНИЕ ДЕРЕВА ИНТЕРВАЛОВ В ПАМЯТИ

Очевидно, что хранить дерево интервалов можно как обычное двоичное дерево — для каждой вершины помимо ключа запоминаются указатели на левого и правого сыновей. При этом используется линейное от числа элементов исходной последовательности количество памяти. Такой подход не ухудшает асимптотических оценок времени выполнения, однако таит в себе большую константу, скрытую в этих оценках. Поэтому рассмотрим более эффективный подход.

Для простоты будем считать, что у нас есть полное двоичное дерево, листья которого соответствуют элементам последовательности. Если же N не является степенью двойки, добавим необходимое количество элементов, при этом положим их равными *нейтральному элементу* относительно искомой функции, т. е. значению, добавление которого в мультимножество-аргумент не изменяет значения функции. Например, для суммы элементов мультимножества нейтральным является 0, а для минимума — $+\infty$. При необходимости нейтральный элемент может быть введён в множество, из которого берутся элементы последовательности, искусственно. Новая последовательность будет содержать $N' < 2N$ элементов.

Хранить такое дерево можно по принципу двоичной кучи — занумеровать все вершины начиная с корня, уровень за уровнем от 1 до $2N' - 1$, чтобы родителем вершины v_i была вершина $v_{\lfloor i/2 \rfloor}$ (рис. 1.3). Вершина v_i будет листом тогда и только тогда, когда $i \geqslant N'$. Кроме

того, будем считать, что она соответствует элементу $a_{i-N'}$ исходной последовательности. Если же v_i не лист, то вершины v_{2i} и v_{2i+1} будут её левым и правым сыновьями соответственно. Значит, хранить указатели на сыновей не требуется, достаточно одного массива, содержащего ключи вершин. Таким образом, для всей структуры дерева интервалов потребуется не более $4N-3$ чисел из того же множества, что и элементы последовательности (будем считать, что все значения искомой функции также принадлежат этому множеству), поскольку дерево содержит $2N'-1$ вершин и $N' \leqslant 2N-1$.

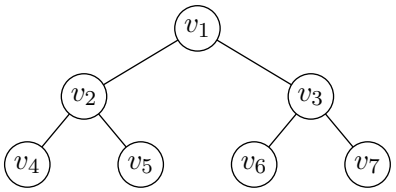


Рис. 1.3. Нумерация вершин при хранении по принципу двоичной кучи

Для построения дерева интервалов сначала зададим значения ключей листьев (элементы последовательности и нейтральный элемент), а затем, поднимаясь каждый раз по дереву на один уровень, будем находить значения ключей всех вершин уровня, пока не дойдём до корня. Тогда при использовании этого алгоритма время построения дерева интервалов будет линейным от длины последовательности, поскольку ключ каждой вершины будет единожды записан и не более чем дважды прочитан, а число вершин дерева составит $\mathrm{O}(N)$.

Построение дерева интервалов

Функция Построить: **последовательность** $(a_i)_{i=0}^{N-1} \to$ $\to$ **дерево интервалов**

$N' \leftarrow 1$.
Пока $N' < N$
    $N' \leftarrow 2N'$.
Для всех i **от** 0 **до** $N-1$
    Ключ $v_{i+N'} \leftarrow f(\langle a_i \rangle)$.
Для всех i **от** N **до** $N'-1$
    Ключ $v_{i+N'} \leftarrow$ нейтральный элемент.
Для всех i **от** $N'-1$ **до** 1
    Ключ $v_i \leftarrow F(\text{ключ } v_{2i}, \text{ ключ } v_{2i+1})$.
$N \leftarrow N'$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

Тем не менее можно понизить оценку на число вершин до $2N - 1$, а также убрать необходимость в нейтральном элементе, немного изменив структуру дерева. Для этого оставим в построенном дереве $2N - 1$ вершин, убрав $2N' - 2N$ вершин на нижнем уровне полного двоичного дерева (рис. 1.4). Пусть теперь элементу a_i соответствует вершина $v_{N'+i}$, если $i < 2N - N'$, или вершина $v_{i-N+N'}$, если $i \geqslant 2N - N'$. Идея построения за линейное время остаётся той же — сначала инициализировать все листья, а потом вычислять ключи вершин, у которых ключи обоих сыновей уже известны.

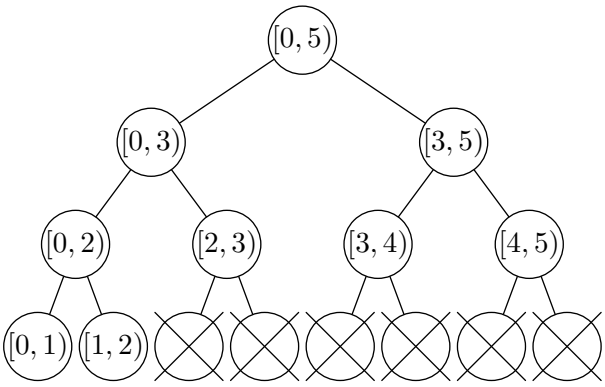


Рис. 1.4. Уменьшение числа вершин дерева

Последний вариант хранения будет полезен в тех случаях, когда число элементов последовательности много больше числа запросов или использование $\Omega(N)$ памяти недопустимо вовсе. При этом предполагается, что начальные значения элементов последовательности подчиняются достаточно простому закону, позволяющему аналитически вычислять ключи вершин (не только листьев). Например, если все элементы последовательности изначально равны одному и тому же числу x, то минимум среди некоторой (непустой) части из них равен x, а сумма — kx, где k — количество слагаемых.

Будем хранить вершины так же, как в первом рассмотренном варианте (с указателями на левого и правого сыновей), с той лишь разницей, что изначально у нас будет только корень дерева, а все остальные вершины будут достраиваться по мере необходимости. Если востребованная вершина не была построена ранее, можно достроить её и придать её ключу такое значение, каким оно было бы в самом начале. При

обработке каждого запроса мы будем достраивать $O(\log N)$ вершин, поскольку в каждом из рассматриваемых далее алгоритмов запрашиваются или изменяются значения ключей не более чем четырёх вершин каждого уровня. Значит, общее количество используемой памяти составит $O(Q \log N)$, где Q — число запросов.

В дальнейшем будем полагать, что хранятся только ключи вершин и дерево является полным двоичным. При использовании другого метода хранения алгоритмы нужно будет модифицировать, тем не менее идеи и асимптотические оценки сохранятся.

1.3. БАЗОВОЕ ИСПОЛЬЗОВАНИЕ

1.3.1. Интервальный подсчёт

Сначала разберём интервальный подсчёт на $[0, i)$. На нижнем уровне нас интересуют все вершины левее v_{i+N}. Чтобы быстро перейти на уровень выше, сохранив свойство: всем нерассмотренным элементам последовательности соответствуют вершины текущего уровня левее некоторой из них, достаточно исключить из рассмотрения не более одной вершины. Если вершин чётное число, то все они разбиваются на пары братьев (*братом* называется сын родителя вершины, не совпадающий с самой вершиной), тогда вместо каждой такой пары вершин можно рассматривать их родителя. Если вершин нечётное число, то только самая правая из них будет беспарной. Исключая вершину из рассмотрения, будем пересчитывать ответ, добавляя к нему соответствующие ей элементы последовательности. При этом каждый элемент будет учтён в ответе ровно один раз.

Переходя таким образом от уровня к уровню, мы в какой-то момент получим пустое множество нерассмотренных вершин. При этом операций перевычисления потребуется не больше, чем переходов на уровень выше, а всего уровней в дереве $\lceil \log_2 N \rceil + 1$, поэтому время выполнения одного подсчёта составит $O(\log N)$.

Интервальный подсчёт на $[0, i)$

Функция Подсчитать: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i) $\to$ **число**

$i \leftarrow i + N$. // *Получаем индекс листа.*
Ответ $\leftarrow$ нейтральный элемент.
Пока $i > 1$
    Если i нечётное
        Ответ $\leftarrow F($ответ, ключ $v_{i-1})$.
    $i \leftarrow \lfloor i/2 \rfloor$.
Вернуть ответ.

В случае подсчёта на интервале (i, j) для произвольных $i < j$ воспользуемся похожим методом. Рассмотрим множество вершин текущего уровня, соответствующих элементам последовательности, полностью включённым в запрос. Вершины того же уровня, которые не будут рассматриваться, но будут соседними с рассматриваемыми, назовём *вершинами – границами запроса.*

Вначале рассмотрим листья с индексами из $(i + N, j + N)$. Вместо того чтобы пересчитывать результат, используя ключи двух вершин с общим отцом, можно воспользоваться ключом их отца для пересчёта. Значит, результат нужно пересчитать, используя ключ самой левой из рассматриваемых вершин, если она является правым сыном своего отца, а также ключ самой правой из рассматриваемых, если она является левым сыном своего отца. Заметим, что дважды одну и ту же вершину мы не учтём, потому что она не может быть одновременно левым и правым сыном своего отца. После этого мы можем перейти к рассмотрению вершин следующего уровня. При этом новые вершины – границы запроса будут родителями текущих. Это действие выполняется, пока множество рассматриваемых на текущем уровне вершин не станет пустым, но в любом случае не более чем $O(\log N)$ раз, поскольку каждый уровень обрабатывается за $O(1)$, а всего уровней $\lceil \log_2 N \rceil$. На рис. 1.5 проиллюстрирован пример интервального подсчёта. Серым обозначены вершины – границы запроса, жирным выделены вершины, ключи которых используются при подсчёте.

Интервальный подсчёт на (i, j)

Функция Подсчитать: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i, **целое** j) $\rightarrow$ **число**
    $i \leftarrow i + N$. // *Получаем индексы листьев.*

$j \leftarrow j + N$.
Ответ $\leftarrow$ нейтральный элемент.
Пока $j - i > 1$
    Если i чётное // v_{i+1} *является правым сыном.*
        Ответ $\leftarrow F($ответ, ключ $v_{i+1})$.
    Если j нечётное // v_{j-1} *является левым сыном.*
        Ответ $\leftarrow F($ответ, ключ $v_{j-1})$.
    $i \leftarrow \lfloor i/2 \rfloor$.
    $j \leftarrow \lfloor j/2 \rfloor$.
Вернуть ответ.

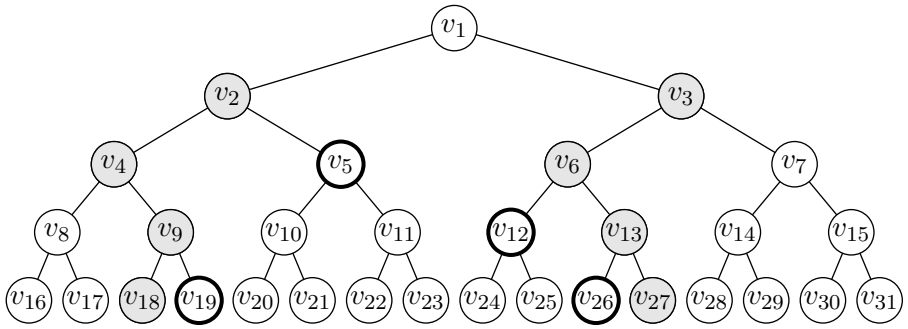


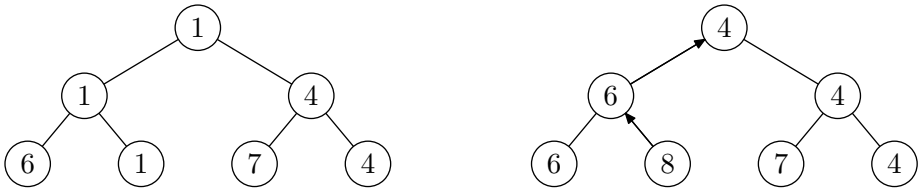
Рис. 1.5. Интервальный подсчёт на интервале $(18, 27)$

Внимательный читатель уже мог озадачиться вопросом, что произойдёт, если $j = N$ или $i = -1$. Заметим, что проверку, является ли вершина левым (правым) сыном своего родителя, мы можем осуществлять по чётности её индекса, а переход к родителю — делением индекса на 2 нацело, поэтому проверка выполнима, даже если у вершины нет родителя или сама вершина не существует. Пусть $i = -1$. Тогда левая вершина–граница запроса фактически оказывается на уровень выше рассматриваемых. Но она всегда будет правым сыном своего родителя (кроме возможного случая, когда она окажется корнем, но проверка по чётности индекса также обозначит корень правым сыном), поэтому ключ другого сына её родителя не будет оказывать влияния на результат. Случай $j = N$ аналогичен с той лишь разницей, что здесь правая вершина–граница запроса будет находиться на уровень ниже рассматриваемых, но всегда будет левым сыном своего родителя (включая начальную несуществующую вершину v_{2N}). Если же $i = -1$ и $j = N$

одновременно, то первым и единственным ключом, учтённым в ответе, будет ключ корня. Это произойдёт, когда левая вершина–граница запроса будет иметь индекс 0, а правая — 2, поскольку в этом случае индекс левой вершины–границы запроса впервые станет чётным. При переходе на следующий уровень получится пустое множество рассматриваемых вершин. Таким образом, случаи $i = -1$ и $j = N$ отдельно рассматривать не требуется.

1.3.2. Одиночное изменение

Теперь рассмотрим одиночное изменение. Когда меняется значение одного элемента, меняются ключи всех вершин, которые ему соответствуют, т. е. ключи некоторого листа и всех его предков. Эти ключи и требуется пересчитать при одиночном изменении, при этом двигаться следует от листа к корню, чтобы у каждой рассматриваемой вершины на момент пересчёта её ключа все потомки имели актуальные ключи (рис. 1.6).



а *б*

Рис. 1.6. Установление значения при подсчёте минимума:
а — исходное состояние; *б* — установление a_1 значения 8

Одиночное изменение: установление значения a_i в x

Функция Установить: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i, **число** x) $\to$ **дерево интервалов**
$i \leftarrow i \ + N$.
Ключ $v_i \leftarrow f(\langle x \rangle)$.
Пока $i > 1$
$i \leftarrow \lfloor i/2 \rfloor$.
Ключ $v_i \leftarrow F(\text{ключ } v_{2i}, \text{ ключ } v_{2i+1})$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

Для получения значения элемента a_i достаточно воспользоваться ключом соответствующего ему листа v_{i+N}, а чтобы прибавить к значе-

нию этого элемента некоторое число d, можно установить ему значение $a_i + d$. С другой стороны, при подсчёте суммы можно сразу увеличить на d значения ключей соответствующего листа и всех его предков.

Одиночное изменение:
увеличение значения a_i на d при подсчёте суммы

Функция Прибавить: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i, **число** d) $\to$ **дерево интервалов**
  $i \leftarrow i + N$.
  Пока $i > 1$
    $i \leftarrow \lfloor i/2 \rfloor$.
    Ключ $v_i \leftarrow$ ключ $v_i + d$.
  Вернуть $(v_i)_{i=1}^{2N-1}$.

1.4. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

1.4.1. Интервальное изменение

Наиболее часто востребованная функциональность дерева интервалов уже разобрана, однако его уникальной возможностью является именно интервальное изменение. Очевидно, что при интервальном изменении невозможно достаточно эффективно поддерживать актуальными значения ключей всех вершин, поскольку интервальное изменение на (i, j) затрагивает значения ключей $\Omega(j - i)$ вершин. Для достижения желаемой оценки $O(\log N)$ потребуется отказаться от поддержания актуальности ключей всех вершин, а также наложить на изменение ограничение, что композиция двух изменений является изменением из того же класса. (Здесь идёт речь об изменениях, применённых к одним и тем же элементам последовательности.) Например, добавление к элементам последовательности сначала x, а затем y эквивалентно добавлению к этим элементам $x + y$, в то время как установление значения элементов сначала в x, а затем в y эквивалентно установлению их значения в y. В некоторых случаях можно расширить класс изменений, чтобы он приобрёл такое свойство.

Итак, значения ключей вершин могут не быть актуальными, но для возможности интервального подсчёта будем поддерживать следующее

свойство: каждое интервальное изменение, коснувшееся элемента a_i, отражено хотя бы в одном предке v_{i+N}. Тогда, спускаясь из корня к листу и применяя накопившиеся у родителя изменения к сыновьям, мы сможем обновить значения ключей всех его предков (включая сам лист) и их братьев. Заметим, что при интервальном подсчёте на (i, j) используются только значения ключей братьев предков v_i и v_j (включая братьев самих v_i и v_j), поэтому в подсчёт нужно добавить только две операции обновления.

Интервальный подсчёт на (i, j)
при возможности интервального изменения

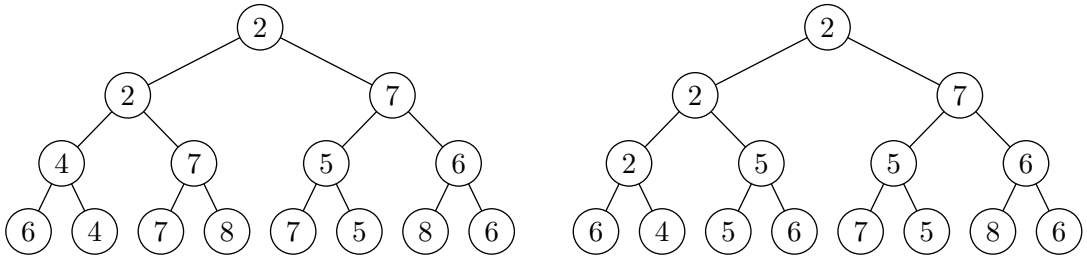
Функция Подсчитать: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i, **целое** j) $\to$ **число**

$i \leftarrow i + N$.
$j \leftarrow j + N$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, i)$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, j)$.
Ответ $\leftarrow$ нейтральный элемент.
Пока $j - i > 1$
  Если i чётное
    Ответ $\leftarrow F($ответ, ключ $v_{i+1})$.
  Если j нечётное
    Ответ $\leftarrow F($ответ, ключ $v_{j-1})$.
  $i \leftarrow \lfloor i/2 \rfloor$.
  $j \leftarrow \lfloor j/2 \rfloor$.
Вернуть ответ.

Обновлять ключи вершин – границ запроса, не попадающих на тот же уровень, что и рассматриваемые вершины, не требуется, так как ключи братьев их предков не будут востребованы. Интервальное изменение похоже на интервальный подсчёт. Точно так же будем подниматься от листьев, накапливая изменения в непарных вершинах текущего уровня и переходя к родителям парных вершин. При этом будем пересчитывать значение ключа вершин – границ запроса, поскольку изменение касается только части потомков этих вершин. Кроме того, когда вершины – границы запроса окажутся соседними, следует не прек-

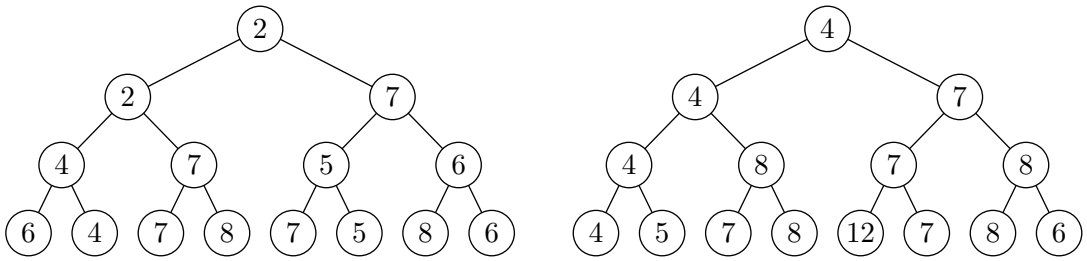
ращать двигаться вверх к их предкам, а обновить значения ключей этих предков, не забывая при этом, что изменение уже не может коснуться сразу всех потомков какой-либо вершины текущего уровня. Помимо этого, если изменения не коммутативны, потребуется сначала обновить значения ключей предков вершин – границ запроса и их братьев во избежание конфликтных ситуаций, когда новое изменение, касающееся всех потомков текущей вершины, затирается впоследствии при обновлении более старым изменением, касающимся всех потомков её предка.

Разберём подробнее на примере *интервального увеличения*, т. е. увеличения всех элементов запроса на одну и ту же величину (возможно, отрицательную), и подсчёта минимума. В этом случае отследить изменение, применённое к вершине, но не применённое к её сыновьям (неактуальность ключей сыновей), можно по ключам этих вершин.



а *б*

Рис. 1.7. Обновление при подсчёте минимума:
а — до обновления; *б* — после обновления v_2



а *б*

Рис. 1.8. Интервальное изменение при подсчёте минимума:
а — до изменения; *б* — после увеличения $a_1, \ldots, a_4$ на 3

Чтобы минимум из ключей сыновей стал равен ключу их родителя, нам нужно добавить к ключам сыновей одну и ту же величину (возможно, 0). Значит, эта величина равна разности ключа родителя

и минимума из ключей сыновей. При самом интервальном изменении будем изменять ключи непарных вершин текущего уровня. Если добавлена одна и та же величина ко всем элементам интервала вершины, то и значение ключа вершины изменится в точности на ту же величину. Примеры обновления и интервального изменения приведены на рис. 1.7 и 1.8 соответственно.

Интервальное увеличение
при интервальном подсчёте минимума

```
// Обновляет ключи предков вершины v_i и их братьев.
Функция Обновить: (дерево интервалов (v_i)_{i=1}^{2N-1},
                   целое i) → дерево интервалов
    Если i = N − 1 или i = 2N
        Вернуть (v_i)_{i=1}^{2N-1}.
    Для всех k от log_2 N до 1
        j ← ⌊i/2^k⌋. // Индекс предка v_i на уровне k.
        x ← ключ v_j − min{ключ v_{2j}, ключ v_{2j+1}}.
        Ключ v_{2j} ← ключ v_{2j} + x.
        Ключ v_{2j+1} ← ключ v_{2j+1} + x.
    Вернуть (v_i)_{i=1}^{2N-1}.

// Прибавляет к элементам последовательности
// с индексами из (i, j) величину d.
Функция Прибавить: (дерево интервалов (v_i)_{i=1}^{2N-1}, целое i,
                    целое j, число d) → дерево интервалов
    i ← i + N.
    j ← j + N.
    (v_i)_{i=1}^{2N-1} ← Обновить((v_i)_{i=1}^{2N-1}, i).
    (v_i)_{i=1}^{2N-1} ← Обновить((v_i)_{i=1}^{2N-1}, j).
    Пока j − i > 1
        Если i чётное
            Ключ v_{i+1} ← ключ v_{i+1} + d.
        Если j нечётное
            Ключ v_{j−1} ← ключ v_{j−1} + d.
        i ← ⌊i/2⌋.
```

$j \leftarrow \lfloor j/2 \rfloor$.
Ключ $v_i \leftarrow \min\{\text{ключ } v_{2i}, \text{ ключ } v_{2i+1}\}$.
Ключ $v_j \leftarrow \min\{\text{ключ } v_{2j}, \text{ ключ } v_{2j+1}\}$.
Пока $i > 0$
Ключ $v_i \leftarrow \min\{\text{ключ } v_{2i}, \text{ ключ } v_{2i+1}\}$.
$i \leftarrow \lfloor i/2 \rfloor$.
Пока $j > 0$
Ключ $v_j \leftarrow \min\{\text{ключ } v_{2j}, \text{ ключ } v_{2j+1}\}$.
$j \leftarrow \lfloor j/2 \rfloor$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

Операция деления на степень двойки, востребованная для получения индекса предка вершины на нужном уровне, осуществима за константное время при помощи битового сдвига. Также отметим, что для получения значения элемента необходимо сначала обновить значение ключа соответствующего ему листа.

Теперь перейдём к подсчёту суммы. Нам по-прежнему не требуется дополнительная информация о проведённом интервальном изменении, хватает значений ключей. Действительно, оба сына одной вершины соответствуют равному количеству элементов последовательности, значит если увеличение было применено ко всем элементам, соответствующим вершине, то ключи её сыновей должны были увеличиться на вдвое меньшую величину. Величину, на которую изменилось значение ключа вершины, можно узнать, отняв от текущего значения сумму ключей сыновей. Интервальное изменение, помимо функции пересчёта ключа вершины по ключам сыновей, будет отличаться от предыдущего примера в одном моменте. Поскольку при подъёме на один уровень число элементов, которым соответствует одна вершина, удваивается, то и значение, прибавляемое к ключам вершин, должно удваиваться при переходе на следующий уровень.

Интервальное увеличение
при интервальном подсчёте суммы

// Обновляет ключи предков вершины v_i и их братьев.
Функция Обновить: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$,
целое i) $\rightarrow$ **дерево интервалов**

Если $i = N - 1$ **или** $i = 2N$
    Вернуть $(v_i)_{i=1}^{2N-1}$.
Для всех k **от** $\log_2 N$ **до** 1
    $j \leftarrow \lfloor i/2^k \rfloor$. // *Индекс предка* v_i *на уровне* k.
    $x \leftarrow$ (ключ v_j − ключ v_{2j} − ключ v_{2j+1}) / 2.
    Ключ $v_{2j} \leftarrow$ ключ $v_{2j} + x$.
    Ключ $v_{2j+1} \leftarrow$ ключ $v_{2j+1} + x$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

// *Прибавляет к элементам последовательности*
// *с индексами из* (i, j) *величину* d.
Функция Прибавить: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i,
        целое j, **число** d) → **дерево интервалов**
$i \leftarrow i + N$.
$j \leftarrow j + N$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, i)$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, j)$.
Пока $j - i > 1$
    Если i чётное
        Ключ $v_{i+1} \leftarrow$ ключ $v_{i+1} + d$.
    Если j нечётное
        Ключ $v_{j-1} \leftarrow$ ключ $v_{j-1} + d$.
    $i \leftarrow \lfloor i/2 \rfloor$.
    $j \leftarrow \lfloor j/2 \rfloor$.
    $d \leftarrow 2d$.
    Ключ $v_i \leftarrow$ ключ $v_{2i} +$ ключ v_{2i+1}.
    Ключ $v_j \leftarrow$ ключ $v_{2j} +$ ключ v_{2j+1}.
Пока $i > 0$
    Ключ $v_i \leftarrow$ ключ $v_{2i} +$ ключ v_{2i+1}.
    $i \leftarrow \lfloor i/2 \rfloor$.
Пока $j > 0$
    Ключ $v_j \leftarrow$ ключ $v_{2j} +$ ключ v_{2j+1}.
    $j \leftarrow \lfloor j/2 \rfloor$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

Теперь вернёмся к минимуму и рассмотрим *интервальную инициализацию*, т. е. интервальное изменение, которое представляет собой установление всем элементам запроса одинакового значения. Приём, которым мы воспользовались при интервальном увеличении, здесь не применим. Действительно, если ключ вершины не совпадает с минимумом ключей сыновей, то всем элементам, которым соответствует вершина, было установлено значение, равное значению ключа вершины. Но если ключ вершины совпадает с минимумом ключей сыновей, из этого не следует, что элементам, которым она соответствует, не было установлено такое же значение одним из предыдущих запросов. Проблему можно решить, «испортив» значения ключей сыновей при применении интервального изменения к вершине, например установив им значение, большее или меньшее того, которое принимает ключ самой вершины. (В приводимом ниже псевдокоде предполагается, что элемент последовательности не может принимать значение $+\infty$, поэтому число, на 1 большее ключа любой вершины, отличается от значения этого ключа.) Старые значения ключей сыновей больше не имеют смысла, поскольку все соответствующие им элементы последовательности приняли новое значение, а «испорченные» значения ключей сыновей сигнализируют о необходимости обновить значения ключей всех потомков вершины.

Интервальное увеличение
при интервальном подсчёте минимума

```
// Обновляет ключи предков вершины v_i и их братьев.
Функция Обновить: (дерево интервалов (v_i)_{i=1}^{2N-1},
                   целое i) → дерево интервалов
    Если i = N − 1 или i = 2N
        Вернуть (v_i)_{i=1}^{2N-1}.
    Для всех k от log_2 N до 1
        j ← ⌊i/2^k⌋.
        Если ключ v_j ≠ min{ключ v_{2j}, ключ v_{2j+1}}
            Ключ v_{2j} ← ключ v_j.
            Ключ v_{2j+1} ← ключ v_j.
            Если 2j < N // Сыновья не листья.
```

```
                Ключ v_{4j} ← ключ v_{2j} + 1.
                Ключ v_{4j+1} ← ключ v_{2j} + 1.
                Ключ v_{4j+2} ← ключ v_{2j+1} + 1.
                Ключ v_{4j+3} ← ключ v_{2j+1} + 1.
    Вернуть (v_i)_{i=1}^{2N-1}.

// Устанавливает элементам последовательности
// с индексами из (i, j) значение d.
Функция Установить: (дерево интервалов (v_i)_{i=1}^{2N-1},
    целое i, целое j, число x) → дерево интервалов
    i ← i + N.
    j ← j + N.
    (v_i)_{i=1}^{2N-1} ← Обновить((v_i)_{i=1}^{2N-1}, i).
    (v_i)_{i=1}^{2N-1} ← Обновить((v_i)_{i=1}^{2N-1}, j).

    Пока j − i > 1
        Если i чётное
            Ключ v_{i+1} ← x.
            Если i < N
                Ключ v_{2i+2} ← x + 1.
                Ключ v_{2i+3} ← x + 1.
        Если j нечётное
            Ключ v_{j−1} ← x.
            Если j < N
                Ключ v_{2j−2} ← x + 1.
                Ключ v_{2j−1} ← x + 1.
        i ← ⌊i/2⌋.
        j ← ⌊j/2⌋.
        Ключ v_i ← min{ключ v_{2i}, ключ v_{2i+1}}.
        Ключ v_j ← min{ключ v_{2j}, ключ v_{2j+1}}.
    Пока i > 0
        Ключ v_i ← min{ключ v_{2i}, ключ v_{2i+1}}.
        i ← ⌊i/2⌋.
    Пока j > 0
```

Ключ $v_j \leftarrow \min\{\text{ключ } v_{2j}, \text{ ключ } v_{2j+1}\}$.
$j \leftarrow \lfloor j/2 \rfloor$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

Та же идея применима и при поиске суммы элементов последовательности с возможностью установления всем элементам запроса одинакового значения. Псевдокод приводить не будем, поскольку все изменения абсолютно аналогичны случаю интервального увеличения.

1.4.2. Комбинация интервальных изменений

Итак, мы рассмотрели несколько возможных пар интервального изменения и интервального подсчёта. Теперь рассмотрим случай с интервальными изменениями разных типов. Здесь нам нужно ввести новый тип интервального изменения, включающий в себя все необходимые изменения, но обладающий тем же свойством: композиция двух изменений является изменением того же класса. Очевидно, что не любую пару интервальных изменений можно объединить.

Пусть искомая функция является минимумом, при этом допустимы интервальное увеличение и интервальная инициализация. Получить интервальное изменение смешанного типа здесь не составит труда — достаточно объединить два класса возможных интервальных изменений, ничего не добавляя к результату. Действительно, добавление к элементам последовательности x и последующее установление их значения в y эквивалентны установлению их значения в y, а установление значения элементов в x и последующее увеличение на y эквивалентны установлению их значения в $x + y$.

Чтобы ограничиться использованием ключей вершин, будем использовать для пометки о проведённой интервальной инициализации специальное значение, которое не может быть ключом вершины при любых допустимых значениях элементов последовательности. Будем считать, что $+\infty$ обладает таким свойством. Если же все возможные значения ключа вершины допустимы, можно добавить флаг, сигнализирующий о произошедшей интервальной инициализации. При обновлении этот флаг вместе со значением инициализации передаётся сыновьям и сбрасывается у самой вершины.

Для обновления вершины и передачи сыновьям информации о накопленных для соответствующих элементов изменений будем проверять, равны ли ключи сыновей $+\infty$. В случае равенства установим их ключам такое же значение, как у текущей вершины, в противном случае будем считать, что интервальная инициализация элементов не производилась, но произошло интервальное увеличение (возможно, на 0). Разница между подсчётом суммы и подсчётом минимума уже была рассмотрена, поэтому приведём псевдокод только для случая подсчёта суммы.

Смешанное интервальное изменение при интервальном подсчёте суммы

```
// Обновляет ключи предков вершины v_i и их братьев.
Функция Обновить: (дерево интервалов (v_i)_{i=1}^{2N-1},
                   целое i) → дерево интервалов
    Если i = N − 1 или i = 2N
        Вернуть (v_i)_{i=1}^{2N-1}.
    Для всех k от log_2 N до 1
        j ← ⌊i/2^k⌋. // Индекс предка v_i на уровне k.
        Если ключ v_{2j} = +∞
            Ключ v_{2j} ← ключ v_j / 2.
            Ключ v_{2j+1} ← ключ v_j / 2.
            Если 2j < N // Сыновья не листья.
                Ключ v_{4j} ← +∞.
                Ключ v_{4j+1} ← +∞.
                Ключ v_{4j+2} ← +∞.
                Ключ v_{4j+3} ← +∞.
        Иначе
            x ← (ключ v_j − ключ v_{2j} − ключ v_{2j+1}) / 2.
            Ключ v_{2j} ← ключ v_{2j} + x.
            Ключ v_{2j+1} ← ключ v_{2j+1} + x.
    Вернуть (v_i)_{i=1}^{2N-1}.

// Прибавляет к элементам последовательности
// с индексами из (i, j) величину d.
```

Функция Прибавить: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i, **целое** j, **число** d) $\to$ **дерево интервалов**
$i \leftarrow i + N$.
$j \leftarrow j + N$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, i)$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, j)$.

Пока $j - i > 1$
Если i чётное
Ключ $v_{i+1} \leftarrow$ ключ $v_{i+1} + d$.
Если j нечётное
Ключ $v_{j-1} \leftarrow$ ключ $v_{j-1} + d$.
$i \leftarrow \lfloor i/2 \rfloor$.
$j \leftarrow \lfloor j/2 \rfloor$.
$d \leftarrow 2d$.
Ключ $v_i \leftarrow$ ключ $v_{2i} +$ ключ v_{2i+1}.
Ключ $v_j \leftarrow$ ключ $v_{2j} +$ ключ v_{2j+1}.
Пока $i > 0$
Ключ $v_i \leftarrow$ ключ $v_{2i} +$ ключ v_{2i+1}.
$i \leftarrow \lfloor i/2 \rfloor$.
Пока $j > 0$
Ключ $v_j \leftarrow$ ключ $v_{2j} +$ ключ v_{2j+1}.
$j \leftarrow \lfloor j/2 \rfloor$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

// Устанавливает элементам последовательности
// с индексами из (i, j) *значение* d.
Функция Установить: (**дерево интервалов** $(v_i)_{i=1}^{2N-1}$, **целое** i, **целое** j, **число** x) $\to$ **дерево интервалов**
$i \leftarrow i + N$.
$j \leftarrow j + N$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, i)$.
$(v_i)_{i=1}^{2N-1} \leftarrow$ Обновить$((v_i)_{i=1}^{2N-1}, j)$.

Пока $j - i > 1$
    Если i чётное
        Ключ $v_{i+1} \leftarrow x$.
        Если $i < N$
            Ключ $v_{2i+2} \leftarrow +\infty$.
            Ключ $v_{2i+3} \leftarrow +\infty$.
    Если j нечётное
        Ключ $v_{j-1} \leftarrow x$.
        Если $j < N$
            Ключ $v_{2j-2} \leftarrow +\infty$.
            Ключ $v_{2j-1} \leftarrow +\infty$.
    $i \leftarrow \lfloor i/2 \rfloor$.
    $j \leftarrow \lfloor j/2 \rfloor$.
    $x \leftarrow 2x$.
    Ключ $v_i \leftarrow$ ключ v_{2i} + ключ v_{2i+1}.
    Ключ $v_j \leftarrow$ ключ v_{2j} + ключ v_{2j+1}.
Пока $i > 0$
    Ключ $v_i \leftarrow$ ключ v_{2i} + ключ v_{2i+1}.
    $i \leftarrow \lfloor i/2 \rfloor$.
Пока $j > 0$
    Ключ $v_j \leftarrow$ ключ v_{2j} + ключ v_{2j+1}.
    $j \leftarrow \lfloor j/2 \rfloor$.
Вернуть $(v_i)_{i=1}^{2N-1}$.

Теперь обсудим, что делать в случае произвольного интервального изменения и произвольной функции подсчёта. Если невозможно обновить ключи сыновей вершины по актуальному ключу их родителя и старым значениям их ключей, как это было сделано в случае интервального увеличения, либо по актуальному ключу родителя, видя, что ключи сыновей «плохие», как это было в случае интервальной инициализации, требуется для каждой вершины помимо ключа сохранять информацию о проведённом изменении, достаточную как для передачи сыновьям и получения ими аналогичной информации, так и для композиции двух изменений. То же относится к комбинации интервальных изменений. Поиск способа использования дерева интервалов в каждой

конкретной ситуации может оказаться нетривиальной задачей. В определённых случаях может потребоваться некоторое изменение самих методов, построенных исходя из введённых ограничений на функцию подсчёта, но общая концепция структуры данных сохраняется. Отметим, что приведённое здесь избежание рекурсивных вызовов при обработке запросов существенно увеличивает практическую скорость (и полезность) рассмотренной структуры данных.

1.5. ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

1. Звёздочки. Астрономы часто изображают карту звёздного неба на бумаге, где каждая звезда имеет декартовы координаты. Пусть уровнем звезды будет количество других звёзд, которые расположены на карте не выше и не правее данной звезды. Астрономы решили узнать уровни всех звёзд, и за этим они обратились к вам. Требуется определить, сколько звёзд каждого уровня имеется на карте.

2. Хорды. На окружности отмечены $2N$ различных точек, пронумерованных против часовой стрелки от 1 до $2N$. Петя нарисовал N хорд, i-я из которых соединяет точки с номерами a_i и b_i. При этом каждая точка является концом ровно одной хорды.

Теперь Петя заинтересовался, сколько пар хорд пересекаются. Помогите ему определить это количество.

3. Мегаинверсии. Раньше Петя Булочкин писал сортировку вставками и интересовался вопросом, сколько сравнений выполнит его алгоритм, упорядочивая заданную перестановку натуральных чисел от 1 до N. Теперь он знает, что количество этих сравнений будет равно количеству инверсий в перестановке. К слову сказать, *инверсией* в перестановке $p_1, p_2, \ldots, p_N$ называется пара (i, j), для которой $i < j$ и $p_i > p_j$. Но, как уже говорилось, вопрос об инверсиях в перестановке сейчас мало волнует Петю. Теперь у него появилась новая идея, которая не даёт ему покоя. Петя ввёл математический термин. Он назвал *мегаинверсией* в перестановке $p_1, p_2, \ldots, p_N$ тройку (i, j, k), для которой $i < j < k$ и $p_i > p_j > p_k$. Сейчас Петя ломает голову, придумывая алгоритм для быстрого подсчёта количества мегаинверсий в переста-

новке. Так как Петя уже очень долго думает над задачей и никаких дельных мыслей у него пока нет, то ему кажется, что для этой задачи быстрого алгоритма вообще не существует. Докажите Пете, что он неправ.

4. Забор. Граффити стали ночным кошмаром для мэра города Гулбене (с населением в 10 000 человек) — они содержат разные слоганы, некоторые из которых неприятны людям: «Курят на льду», «Нам не нужен ледовый балет», «Заткнись и катайся на коньках» — и даже очень опасные: «Наша команда лучшая, остальные — дрянь». Мэр попросил полицию исследовать обстоятельства их появления, и после нескольких лет обширных поисков виновные были найдены — ими оказались фанаты местной хоккейной команды. Для помощи в сложившейся ситуации были приглашены дорогостоящие иностранные эксперты, и после нескольких лет исследований и траты денег для надписей был построен забор из 100 000 досок, каждая из которых предназначена для отдельной буквы. Это было с энтузиазмом воспринято фанатами, которые начали писать свои слоганы иногда на свободных досках, а иногда и поверх других!

После этого весьма приятного решения мэру понравился забор — он даже иногда совершает прогулки вдоль него. Интересуясь, насколько полезен забор, он считает длину самого длинного участка забора без каких-либо слоганов, а также длину самого длинного участка без единой свободной дощечки. Мэр очень занятой человек, поэтому прогуливается не вдоль всего забора, а вдоль какого-либо его участка. Считая длины этих участков, он учитывает только те дощечки, мимо которых он проходит при прогулке.

Вам задана последовательность событий двух типов: а) на участке с A-й по B-ю доску забора появился новый слоган; б) мэр совершил прогулку от A-й до B-й доски забора (числа A и B меняются от события к событию). События упорядочены по времени. Требуется для каждой прогулки мэра вдоль забора найти два числа — длину самого длинного участка забора без каких-либо слоганов и длину самого длинного участка забора без свободных от надписей досок.

Глава 2. СТРОКОВЫЕ АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Задачи обработки строк являются классическими и возникают во многих прикладных областях: при разработке программного обеспечения для работы с текстом (поиск в тексте, анализ кода компьютерных программ, автодополнение, обнаружение плагиата); в антивирусных программах при анализе исполняемых файлов на наличие в них вредоносного кода; в вычислительной биологии и биоинформатике (распознавание белок-кодирующих последовательностей биополимеров, анализ первичной структуры биополимеров, организация баз поиска белковых структур и нуклеотидных последовательностей) и др.

Особенность таких задач заключается в необходимости обработки огромных массивов данных, что невозможно без применения эффективных алгоримов. В данной главе рассматриваются некоторые из распространённых алгоритмов и структур данных, которые используются для решения следующих задач: поиск строки среди множества строк, поиск образца в строке и поиск множества образцов в строке.

2.1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Пусть задано некоторое конечное непустое множество символов Σ, называемое *алфавитом*. Количество элементов в этом множестве будем называть *мощностью алфавита*.

Определение 2.1. *Строка* — произвольная конечная последовательность символов из алфавита: $S = (s_1, \ldots, s_n)$, $s_i \in \Sigma$. *Длина* $|S|$ *строки* S — количество элементов данной последовательности.

Определение 2.2. *Подстрока* — непрерывная подпоследовательность строки:

$$S(i, j) = (s_i, s_{i+1}, \ldots, s_j).$$

В случае если $i > j$, подстроку $S(i, j)$ будем считать пустой.

Определение 2.3. Подстрока $S(1, i)$, которая состоит из i первых символов строки, называется *префиксом*.

Определение 2.4. Подстрока $S(i, |S|)$, состоящая из $|S| - i + 1$ последних символов строки, называется *суффиксом*.

Например, строки «*алгори*», «*гор*» и «*ритм*» являются подстроками строки «*алгоритм*». Причём «*алгори*» — её префикс, а «*ритм*» — суффикс.

Определение 2.5. Строки $S = (s_1, \ldots, s_n)$ и $T = (t_1, \ldots, t_m)$ называются *равными* или *совпадающими*, если соответствующие им последовательности символов равны:

$$S = T \iff s_i = t_i, \quad \forall\, i \in \{1, \ldots, |S|\},\ |S| = |T|.$$

Определение 2.6. *Конкатенация* строк S и T — строка, которая получается в результате приписывания последовательности символов строки T после последовательности символов строки S:

$$S \cdot T = (s_1, \ldots, s_n, t_1, \ldots, t_m).$$

Например, результатом конкатенации строк «*алго*» и «*ритм*» будет строка «*алгоритм*».

2.2. ПОИСК ОБРАЗЦА В СТРОКЕ

Одной из часто возникающих задач при работе со строками является задача поиска вхождений подстроки в строке. Пусть есть непустая строка S, называемая *образцом*, и некоторая непустая строка T, $|T| \geqslant |S|$, называемая *текстом*. *Задача поиска подстроки в строке* состоит в том, чтобы найти все подстроки строки T, которые совпадают с образцом S. Самый простой способ решения такой задачи — последовательно перебирать в T начальный символ для подстроки, а затем последовательно сравнивать подстроку T со строкой S. В таком случае трудоёмкость операции поиска будет $\mathrm{O}(nm)$, где n — длина строки S, а m — длина строки T. Однако существуют алгоритмы, решающие эту задачу с трудоёмкостью $\mathrm{O}(n + m)$. Ниже приведён один из таких алгоритмов.

2.2.1. Префиксная функция

Определение 2.7. *Префиксной функцией* π от строки S и позиции i в ней называется длина наибольшего префикса подстроки $S(1,i)$, который совпадает с суффиксом этой подстроки, но не равен всей подстроке:

$$\pi(S,i) = \max_{0 \leqslant k < i} \bigl\{ k \bigm| S(1,k) = S(i-k+1,i) \bigr\}.$$

Пример 2.1. Рассмотрим строку «`ababcabcda`» и позицию 7 в ней. Для подстроки «`ababcab`» существует префикс «`ab`», который совпадает с её же суффиксом. Причём ни один префикс большей длины для этой подстроки не совпадает с её суффиксом, за исключением всей подстроки. Таким образом, значение префиксной функции для строки «`ababcabcda`» в позиции 7 равно 2. Для всех позиций строки префиксная функция будет следующей:

i	1	2	3	4	5	6	7	8	9	10
s_i	`a`	`b`	`a`	`b`	`c`	`a`	`b`	`c`	`d`	`a`
$\pi(S,i)$	0	0	1	2	0	1	2	0	0	1

Рассмотрим способ, позволяющий эффективно вычислять префиксную функцию для всех позиций строки. Для подстроки из одного символа, очевидно, префиксная функция будет равна нулю. Далее мы хотим научиться вычислять значение префиксной функции для позиции i, зная её значения для всех предыдущих позиций. Рассмотрим предыдущую позицию $i-1$, для которой значение префиксной функции уже известно и равно $\pi(S,i-1)$. Сравним текущий символ s_i с символом $s_{\pi(S,i-1)+1}$. Если эти символы совпадают, то мы можем продлить префикс, соответствующий значению префиксной функции для позиции $i-1$, на один символ и получить значение префиксной функции: $\pi(S,i) = \pi(S,i-1)+1$. Легко видеть, что большее значение префиксной функции для позиции i получить невозможно, так как в противном случае можно было бы увеличить значение префиксной функции для позиции $i-1$, отбросив i-й символ из суффикса, соответствующего значению префиксной функции для позиции i.

Предположим теперь, что символы s_i и $s_{\pi(S,i-1)+1}$ не совпадают. Если $\pi(S,i-1) = 0$, то очевидно, что и $\pi(S,i) = 0$. Рассмотрим случай, когда $\pi(S,i-1) > 0$. В этом случае $\pi(S,i) \leqslant \pi(S,i-1)$, а значит,

префикс $S\big(1, \pi(S,i)\big)$, соответствующий значению префиксной функции для позиции i, будет подстрокой префикса $S\big(1, \pi(S,i-1)\big)$, определяемого значением префиксной функции для позиции $i-1$. Теперь рассмотрим строку T, получаемую из суффикса строки S, соответствующего значению префиксной функции для позиции i, путём отбрасывания последнего символа: $T = S(i - \pi(S,i), i-1)$. Поскольку $|T| = \pi(S,i) - 1$, а $\pi(S,i) - 1 < \pi(S, i-1)$, то получаем, что строка T является подстрокой суффикса $S\big(i - \pi(S,i-1), i-1\big)$, заданного значением префиксной функции для позиции $i-1$. Кроме этого, из определения префиксной функции для позиции $i-1$ следует, что

$$S\big(1, \pi(S,i-1)\big) = S\big(i - \pi(S,i-1), i-1\big).$$

Значит, для строки T должна существовать равная ей подстрока T' префикса $S\big(1, \pi(S,i-1)\big)$:

$$T' = S\big(\pi(S,i-1) - \pi(S,i) + 1, \pi(S,i-1)\big).$$

Рассмотрим строку S', полученную из префикса $S\big(1, \pi(S,i-1)\big)$ приписыванием i-го символа строки S:

$$S' = S\big(1, \pi(S,i-1)\big) \cdot (s_i).$$

Для строки S' значению префиксной функции $\pi(S', |S'|)$ будут соответствовать префикс $S'\big(\pi(S', |S'|)\big)$ и суффикс $S'\big(|S'| - \pi(S', |S'|) + 1, |S'|\big)$, совпадающий с ним, причём из построения строки S' следует, что

$$S'\big(|S'| - \pi(S', |S'|) + 1, |S'|\big) = T' \cdot (s_i).$$

Используя это и тот факт, что $T' = T$, вместо рассмотрения префиксной функции для подстроки $S(1,i)$ можно перейти к рассмотрению префиксной функции для строки S' меньшей длины, причём значение префиксной функции $\pi(S', |S'|)$ будет равно значению префиксной функции $\pi(S,i)$.

Можно последовательно переходить к подзадаче нахождения префиксной функции от строки меньшей длины до тех пор, пока на некотором шаге не будет получено нулевое значение префиксной функции

либо не встретится совпадение требуемых символов. Псевдокод алгоритма, соотвутствующего указанному выше способу вычисления префиксной функции, приведён ниже.

Заметим, что если доопределить префиксную функцию для позиции 0 значением -1, можно заменить условие получения нулевого значения на условие получения отрицательного значения, чтобы избежать дальнейшего разделения на два случая, так как в обоих случаях потребуется увеличить полученное значение k на 1.

Вычисление префиксной функции

Функция Вычислить: **строка** $S \to$ **последовательность целых**
    $\pi(S,\ 0) \leftarrow -1$.
    Для всех i **от** 1 **до** $|S|$
        $k \leftarrow \pi(S,\ i\ -\ 1)$.
        Пока $k \geqslant 0$ **и** $s_i \neq s_{k+1}$
            $k \leftarrow \pi(S,\ k)$.
        $\pi(S,\ i) \leftarrow k\ +\ 1$.
    Вернуть $\pi(S)$.

На первый взгляд может показаться, что эта функция будет работать за квадратичное, а не за линейное время, так как содержит два вложенных цикла. Рассмотрим их работу более внимательно. Для того чтобы внутренний цикл мог выполнить итерацию, значение префиксной функции предварительно должно быть увеличено как минимум на 1 в результате найденного совпадения символов.

Представим, что вместо значения префиксной функции у нас есть стек, в котором будем хранить номера позиций строки. Начнём выполнение функции с пустым стеком, что соответствует нулевому значению префиксной функции. При увеличении значения префиксной функции на 1 будем добавлять в стек число i — номер позиции строки, в которой произошло увеличение префиксной функции. При выполнении перехода от префикса к префиксу (во время итерации внутреннего цикла) будем извлекать положительное число элементов из стека, чтобы размер стека стал равен значению префиксной функции в данной позиции строки. Сами позиции в стеке будут соответствовать символам, составляющим этот префикс.

Каждая позиция строки может быть добавлена в стек не более одного раза при увеличении значения префиксной функции. Во время выполнения итерации внутреннего цикла из стека извлекается хотя бы одна позиция строки. Причём каждая позиция строки может быть извлечена из стека не более одного раза, так как эта позиция была добавлена в стек не более одного раза. Таким образом, суммарное количество итераций внутреннего цикла не может быть больше, чем количество символов строки. Следовательно, трудоёмкость вычисления префиксной функции есть $\mathrm{O}(n)$, где n — длина строки.

2.2.2. Алгоритм Кнута – Морриса – Пратта

Алгоритм Кнута – Морриса – Пратта использует префиксную функцию, чтобы при поиске подстроки в строке отсечь заведомо ненужные операции сравнения символов текста с символами образца, что позволяет достичь трудоёмкости $\mathrm{O}(n+m)$. Для этого будем последовательно сравнивать символы текста T и образца S и, в случае совпадения символов, перемещаться вперёд по обеим строкам.

Предположим, что найдено несовпадение символов в позиции $i+1$ текста и $k+1$ образца:

$$T(i-k+1,i) = S(1,k) \text{ и } t_{i+1} \neq s_{k+1}.$$

Зная префиксную функцию для образца, мы можем продолжить поиск, сдвинув образец не на один символ вправо, а сразу на несколько символов, при этом длина сдвига зависит от значения префиксной функции в позиции k.

Из определения префиксной функции $\pi(S,k)$ известно, что подстрока $S\big(k-\pi(S,k)+1,k\big)$ длины $\pi(S,k)$, заканчивающаяся в позиции k, совпадает с префиксом $S\big(1,\pi(S,k)\big)$ длины $\pi(S,k)$. Кроме того, эта подстрока совпадает с подстрокой текста $T\big(i-\pi(S,k)+1,i\big)$ длины $\pi(S,k)$, заканчивающейся в позиции i, так как мы только что проверили на совпадение k символов, а $\pi(S,k) < k$. Следовательно, можно сдвинуть образец, получив сразу $\pi(S,k)$ совпадающих символов, если использовать сдвиг на $k-\pi(S,k)$ символов вправо.

Почему не имеет смысла рассматривать сдвиг образца на меньшее количество символов? Из определения префиксной функции мы зна-

ем, что $\pi(S,k)$ — наибольший префикс, совпадающий с суффиксом подстроки для позиции k. Если бы существовал меньший сдвиг образца ℓ, при котором образец также совпал бы с текстом до i-го символа включительно, т. е. $T\big(i-(k-\ell)+1,i\big) = S(1,k-\ell)$, то это потребовало бы существования префикса $S(1,k-\ell)$, совпадающего с подстрокой $S(\ell+1,k)$, но имеющего длину больше $\pi(S,k)$, что является противоречием с определением префиксной функции. Отсюда следует, что при сдвиге образца на количество символов меньше $k-\pi(S,k)$ мы получим несовпадение ещё до i-й позиции текста.

Пример 2.2. Рассмотрим образец «*ababc*» и текст «*abababca*». Префиксная функция образца будет следующей:

i	1	2	3	4	5
s_i	*a*	*b*	*a*	*b*	*c*
$\pi(S,i)$	0	0	1	2	0

Первые четыре символа образца совпадут с первыми четырьмя символами текста. На пятом символе возникнет несовпадение:

a	*b*	*a*	*b*	*a*	*b*	*c*
$=$	$=$	$=$	$=$	$\neq$		
a	*b*	*a*	*b*	*c*		

Поскольку $\pi(\text{«}ababc\text{»},4) = 2$, то префикс образца «*ab*» совпадает с суффиксом «*ab*» подстроки из первых четырёх символов образца. Следовательно, для продолжения сравнения можно сдвинуть образец сразу на два символа и получить совпадение с образцом до четвёртой позиции текста:

a	*b*	*a*	*b*	*a*	*b*	*c*
		$=$	$=$	$\overset{?}{=}$	$\overset{?}{=}$	$\overset{?}{=}$
		a	*b*	*a*	*b*	*c*

Приведём псевдокод поиска подстроки в строке с использованием алгоритма Кнута – Морриса – Пратта.

Алгоритм Кнута – Морриса – Пратта

Функция Найти: (**строка** T, **последовательность целых** $\pi(S)$, **строка** S) $\to$ **множество целых**

$k \leftarrow 0$.

```
Ответ ← ∅.
Для всех i от 0 до |T| − 1
    Пока k ⩾ 0 и t_{i+1} ≠ s_{k+1}
        k ← π(S, k).
    k ← k + 1.
    Если k = |S|
        Ответ ← ответ ∪ {i}.
        k ← π(S, k).
Вернуть ответ.
```

2.3. БОР

Предположим, что есть множество строк $\mathcal{S} = \{S_1, \ldots, S_n\}$ из букв латинского алфавита и необходимо быстро проверять, есть ли среди них некоторая заданная строка $A = (a_1, \ldots, a_\ell)$. Каким образом это можно сделать? Самый простой вариант — поместить все строки в некоторый массив и выполнять поиск, последовательно его просматривая. В этом случае мы получим трудоёмкость $O(n\ell)$, где n — количество имеющихся строк, а ℓ — длина строки A (проверка одного элемента массива в данном случае — сравнение двух строк, которое выполняется за $O(\ell)$, а всего в массиве n строк). Кроме того, для хранения такого массива нам понадобится $\Theta(|S_1|+|S_2|+\ldots+|S_n|)$ памяти.

Массив — не самая удобная структура данных для поиска, и можно попробовать воспользоваться каким-нибудь вариантом сбалансированного бинарного поискового дерева [2]. В этом случае трудоёмкость операции поиска будет равна $O(\ell \log n)$, так как вместо $O(n)$ элементов, просматриваемых в процессе поиска, для поиска в сбалансированном дереве нам достаточно обойти лишь $O(\log n)$ элементов.

Но есть структуры данных, которые позволяют осуществлять поиск строки более эффективно. Оба описанных выше подхода не учитывают особенностей структуры строки. Попробуем использовать тот факт, что строка — последовательность символов, и будем осуществлять поиск не по самим строкам, а по символам строк.

Рассмотрим все строки $\mathcal{S}$. Среди них есть подмножество строк $\mathcal{S}'$, у которых первый символ совпадает с первым символом искомой стро-

ки a_1. Очевидно, что если искомая строка A присутствует среди строк множества $\mathcal{S}$, то она тоже попадёт в подмножество $\mathcal{S}'$. Таким образом, мы можем сократить область поиска до подмножества $\mathcal{S}'$, исключив из рассмотрения те строки, которые в это подмножество не попадают. Если искомая строка состоит из одного символа, то достаточно лишь проверить наличие в полученном множестве строки такой же длины. Если же искомая строка содержит более одного символа, то мы можем перейти к задаче поиска среди подмножества $\mathcal{S}'$ строки длиной на 1 меньше, удалив первый символ строки A и всех строк из $\mathcal{S}'$. Следовательно, для поиска искомой строки нам потребуется выполнить ℓ таких операций.

Для отсеивания неподходящих строк по первому символу можно воспользоваться тем же принципом, который используется в черпачной сортировке [2]. Заведём массив, в качестве индексов которого будут выступать символы алфавита, а в каждом элементе массива будет храниться множество строк. Пример подобной индексации приведён на рис. 2.1. При такой организации данных адресация к множеству строк по первому символу строки может выполняться за O(1).

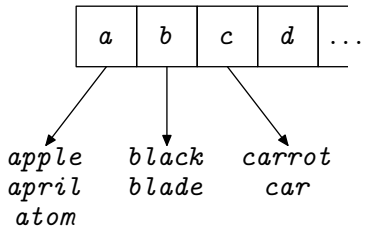


Рис. 2.1. Пример массива, в котором индексация множеств строк производится по их первому символу

Приведённое выше иерархическое разбиение строк на множества естественным образом представляется в виде корневого дерева, показанного на рис. 2.2. Дуги дерева будут соответствовать символам в словах, а сами слова будут записаны в дереве в виде путей от корня до некоторой вершины. Таким образом, каждой вершине v бора B будет соответствовать строка $B(v)$, которая определяется последовательностью символов, встречающихся на дугах в пути по дереву от корня до вершины v.

Каждый лист дерева будет соответствовать некоторому слову из множества $\mathcal{S}$. Однако возможна ситуация, когда одно слово является

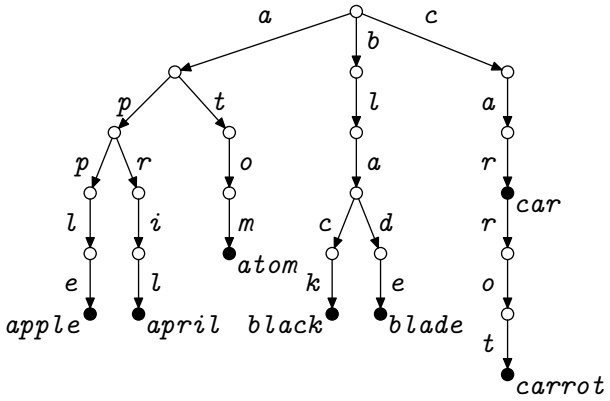


Рис. 2.2. Пример структуры данных *бор*

префиксом другого. В этом случае меньшее слово будет заканчиваться не в листе дерева. Для обработки таких ситуаций в вершинах дерева вводятся пометки, показывающие, существует ли в дереве слово, заканчивающееся в этой вершине. Такая структура данных называется *бором*.

Каждая вершина дерева в боре (далее — вершина бора) содержит информацию о вершинах, в которые можно перейти по каждому допустимому символу. В случае если переход по дуге с некоторым символом невозможен, обозначим результат перехода как $\varnothing$. При работе с бором также может пригодиться переход к родителю вершины v. Дополнительно каждой вершине припишем булеву переменную (пометку), которая будет принимать значение «истина», если строка $B(v)$ присутствует в множестве $\mathcal{S}$, и «ложь» — в противном случае.

Для поиска строки A в боре будем последовательно спускаться от корня дерева по дугам, соответствующим символам строки A, пока строка A не закончится и в боре будет существовать дуга, соответствующая текущему символу строки A. Если в результате спуска мы попадём в помеченную вершину и дойдём до конца строки A, то искомое слово найдено. Если мы остановимся в непомеченной вершине либо во время спуска не найдём в дереве дуги, соответствующей текущему символу, то строка A в боре отсутствует.

Поиск строки в боре

Функция Найти: (**бор** B, **строка** A) $\to$ **вершина**
    $v \leftarrow$ корень B.

Для всех i **от** 1 **до** ℓ
    $u \leftarrow$ переход из v по a_i.
    Если $u = \varnothing$
        Вернуть $\varnothing$.
    $v \leftarrow u$.

Если v помечена // $B(v) \in \mathcal{S}$
    Вернуть v.
Вернуть $\varnothing$.

Всего для поиска искомой строки A длины ℓ необходимо выполнить не более ℓ переходов по дугам. Следовательно, трудоёмкость операции поиска линейно зависит от длины искомой строки A и не зависит от количества строк в боре, т. е. равна $\mathrm{O}(\ell)$.

Также бор позволяет осуществлять операцию добавления нового слова длины ℓ за время $\mathrm{O}(\ell)$. Для этого мы, как и во время поиска, спускаемся по дереву, начиная с корня. Если во время спуска мы попадаем в ситуацию, когда нужно перейти по несуществующей дуге, то добавляем к дереву новую вершину и ведущую в неё дугу с соответствующим символом в качестве пометки.

Добавление строки в бор

Функция Добавить: (**бор** B, **строка** A) $\rightarrow$ **бор**
    $v \leftarrow$ корень B.
    Для всех i **от** 1 **до** ℓ
        $u \leftarrow$ переход из v по a_i.
        Если $u = \varnothing$
            $u \leftarrow$ **новая вершина**.
            пометка $u \leftarrow$ ложь.
            переход из v по $a_i \leftarrow u$.
        $v \leftarrow u$.
    пометка $v \leftarrow$ истина.

2.4. ПОИСК МНОЖЕСТВА ОБРАЗЦОВ В СТРОКЕ

Пусть у нас есть текст T и несколько образцов $\mathcal{S} = \{S_1, \ldots, S_n\}$. Теперь задача состоит в том, чтобы найти все подстроки текста T, которые совпадали бы с одним из образцов. Для её решения можно применить алгоритм из раздела 2.2 по одному разу для каждого образца. Но трудоёмкость такого решения составит

$$\mathrm{O}\Big(\sum_{i=1}^{n}\big(|S_i| + |T|\big)\Big) = \mathrm{O}\Big(n|T| + \sum_{i=1}^{n}|S_i|\Big).$$

Алгоритм Ахо – Корасик позволяет объединить результаты из разделов 2.2 и 2.3 и выполнять поиск вхождений множества образцов в тексте более эффективно — с трудоёмкостью $\mathrm{O}\big(|T| + |K| + \sum_{i=1}^{n}|S_i|\big)$, где $|K|$ — общее количество вхождений.

Построим бор из множества образцов $\mathcal{S} = \{S_1, \ldots, S_n\}$. Начнём поиск с первой позиции текста и корня бора. Встретив в тексте очередной символ, будем совершать переход по дуге, соответствующей этому символу, если такая дуга присутствует в боре. Если же в боре нет дуги, которая соответствовала бы встреченному символу текста, то, по аналогии с алгоритмом Кнута – Морриса – Пратта, нужно осуществить переход к вершине бора, которая соответствует наибольшему собственному префиксу, совпадающему с суффиксом подстроки, соответствующей текущей вершине бора. Можно заметить, что в отличие от алгоритма Кнута – Морриса – Пратта данный префикс не обязательно будет принадлежать этому же образцу.

2.4.1. Функция суффиксных ссылок

Введём *функцию суффиксных ссылок*, которая будет указывать вершину, соответствующую наибольшему подходящему собственному префиксу строки для текущей вершины:

$$\pi(v) = \arg\max_{w \in X \setminus \{v\}} \big\{|B(w)|\big\},$$

где

$$X = \Big\{w \;\Big|\; B(v)\big(|B(v)| - |B(w)| + 1, |B(v)|\big) = B(w)\Big\}.$$

Функцию суффиксных ссылок можно вычислить для каждой позиции строки за линейное от длины строки время, применяя приведённый ниже псевдокод.

Вычисление функции суффиксных ссылок

Функция Вычислить суффиксные ссылки: **бор** $B \to$ **функция** $\pi \leftarrow$ **новая функция**.
$\pi(\text{корень } B) \leftarrow \text{корень } B$.
$Q \leftarrow$ **новая очередь**.
$Q \leftarrow Q + (\text{корень } B,\ \varnothing)$.

Пока Q не пуста
$(v,\ c) \leftarrow \text{начало } Q$.
$Q \leftarrow Q - (v,\ c)$.

Для всех c **из** Σ
$u \leftarrow \text{переход из } v \text{ по } c$.
Если $u \neq \varnothing$
$Q \leftarrow Q + (u,\ c)$.
Если $v = \text{корень } B$
Начать следующую итерацию цикла.
$w \leftarrow \text{родитель } v$.
$u \leftarrow v$.
Пока $u \neq \varnothing$ **и** $w \neq \text{корень } B$
$w \leftarrow \pi(w)$.
$u \leftarrow \text{переход из } w \text{ по } c$.
Если $u \neq \varnothing$
$\pi(v) \leftarrow u$.
Иначе
$\pi(v) \leftarrow \text{корень } B$.
Вернуть π.

2.4.2. Функция вывода

Теперь осталось научиться определять, существует ли в текущей вершине v совпадение с некоторым образцом. Каждая пометка окон-

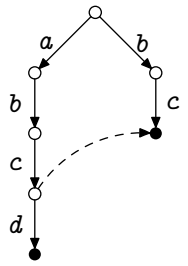


Рис. 2.3. Пример бора с переходами функции суффиксных ссылок

чания слова из структуры бор, описанной в разделе 2.3, будет соответствовать найденному вхождению слова. Однако этих пометок здесь не достаточно.

Пример 2.3. На рис. 2.3 изображена структура, которая получается для множества образцов {«bc», «abcd»}. Сплошными линиями показаны дуги бора, а пунктирной — переход функции суффиксных ссылок. На рисунке видно, что, находясь в вершине, которая соответствует префиксу «abc» слова «abcd», алгоритм должен обнаружить вхождение строки «bc».

Чтобы иметь возможность обнаруживать все вхождения образцов, нужно проверять на наличие пометки не только текущую вершину v, но также и все вершины, соответствующие суффиксам строки $B(v)$. Таким образом, вершина v информирует о некотором вхождении образца, если существует помеченная вершина w, достижимая из v с помощью одного или более переходов по суффиксным ссылкам.

Введём *функцию вывода* $f(v)$, которая будет указывать ближайшую помеченную вершину, достижимую из v по суффиксным ссылкам. Тогда для перечисления всех вхождений слов, соответствующих вершине v, достаточно будет пройти от вершины v до корня по переходам функции вывода, выписывая слова, соответствующие встречающимся по пути вершинам.

Псевдокод алгоритма вычисления функции вывода будет выглядеть следующим образом:

Вычисление функции вывода

Функция Вычислить функцию вывода: (**бор** B, **функция** π) $\rightarrow$ **функция**

$f \leftarrow$ **новая функция**.
$f($корень $B) \leftarrow$ корень B.
$Q \leftarrow$ **новая очередь**.
$Q \leftarrow Q +$ корень B.

Пока Q не пуста

```
        v ← начало Q.
        Q ← Q − v.

        Для всех c из Σ
            u ← переход из v по c.
            Если u ≠ ∅
                Q ← Q + u.
        Если v = корень B
            Начать следующую итерацию цикла.
        w ← π(v).
        Если w помечена
            f(v) ← w.
        Иначе
            f(v) ← f(w).
    Вернуть f.
```

2.4.3. Алгоритм Ахо – Корасик

Используя переходы по дугам бора, а также функцию суффиксных ссылок $\pi(v)$ и функцию вывода $f(v)$, можно осуществлять поиск строк в тексте. Псевдокод алгоритма будет выглядеть следующим образом:

Алгоритм Ахо – Корасик

```
Функция Найти: (строка T, бор B, функция π,
                функция f) → множество
    ответ ← ∅.
    v ← корень B.
    Для всех i от 1 до |T|
        Пока переход из v по t_i = ∅ и v ≠ корень B
            v ← π(v).
        Если переход из v по t_i ≠ ∅
            v ← переход из v по t_i.
            u ← v.
            Если u не помечена
                u ← f(u).
            Пока u ≠ корень B
```

ответ $\leftarrow$ ответ $\cup$ $\{(B(u),\ i)\}$.
$u \leftarrow f(u)$.

Вернуть ответ.

Пример 2.4. Есть следующее множество образцов: {«ab», «aab», «abac», «back», «c»}. Для него получим структуру, изображённую на рис. 2.4, где сплошными линиями показаны дуги бора, пунктирными — переходы функции суффиксных ссылок, а штрихпунктирными — переходы функции вывода (переходы функции вывода, которые идут к корню, на рисунке не показаны).

Если не учитывать добавление строк к ответу, то трудоёмкость алгоритма Ахо–Корасик для обнаружения всех позиций текста T, в которых заканчивается вхождение хотя бы одной из строк из множества $\mathcal{S}$, будет равна $O(|S_1| + \ldots + |S_N| + |T|)$. Доказательство этого факта аналогично доказательству корректности алгоритма Кнута–Морриса–Пратта с той лишь разницей, что вместо позиций строки-образца будут рассматриваться вершины бора.

Однако нужно учитывать, что в одной позиции текста T может оканчиваться несколько вхождений различных строк из множества $\mathcal{S}$, каждую из которых нужно добавить к ответу. Кроме этого, получение каждого такого вхождения будет занимать линейное от его длины время, так как для восстановления строки $B(v)$ требуется подняться по бору от текущей вершины v до корня. Таким образом, общая трудоёмкость алгоритма Ахо–Корасик, соответствующего приведённому выше псевдокоду, будет равна

$$\Theta(|S_1| + \ldots + |S_N| + |T| + |K|),$$

где $|S_1|, \ldots, |S_N|$ — длины строк $S_1, \ldots, S_N$ соответственно, $|K|$ — суммарная длина ответа, $|T|$ — длина текста T.

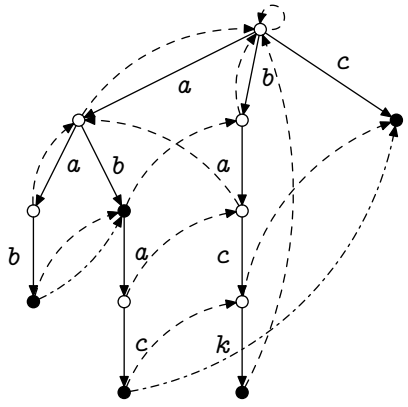


Рис. 2.4. Пример бора с переходами суффиксных ссылок и функции вывода

Заметим, что суммарная длина ответа $|K|$ может превышать $O(|S_1| + \ldots + |S_N|)$.

Пример 2.5. Пусть есть следующее множество образцов:

$$\mathcal{S} = \{«a», «aa», «aaa», \ldots, «\underbrace{aaa\ldots a}_{N}»\}.$$

Предположим, что мы будем искать эти образцы в тексте

$$T = «\underbrace{aaa\ldots a}_{N}».$$

Тогда для первой позиции текста найдём одно вхождение (строка «a»), для второй — два вхождения (строки «a» и «aa» с суммарной длиной 3), для третьей — три вхождения (строки «a», «aa» и «aaa» с суммарной длиной 6). Всего для позиции $i \leqslant N$ будет найдено i вхождений с суммарной длиной $\Theta(i^2)$. Таким образом, суммарная длина ответа для всех позиций текста будет равна $\Theta(N^3)$, в то время как общая длина строк множества образцов составляет $\Theta(N^2)$.

2.5. СУФФИКСНЫЙ МАССИВ

В разделе 2.4 мы разобрали задачу поиска образцов из множества $\mathcal{S}$ в тексте T. Сейчас рассмотрим вариант подобной задачи, когда поиск осуществляется в режиме реального времени. Пусть есть фиксированный текст T. На вход поступают запросы с образцами S, и нам необходимо эффективно определять, встречается ли образец S в качестве подстроки в тексте T. Данную задачу можно решить с помощью суффиксного массива.

2.5.1. Построение суффиксного массива

Суффиксным массивом для строки T называется последовательность лексикографически отсортированных суффиксов $T(i, |T|)$ строки. Очевидно, что достаточно хранить только номера суффиксов.

Пример 2.6. Рассмотрим строку $T =$ «$ababcabcda$». Суффиксный массив для неё будет иметь следующий вид:

Индекс массива	$T(i, \|T\|)$	i
1	a	10
2	$ababcabcda$	1

Индекс массива	$T(i, \|T\|)$	i
3	*abcabcda*	3
4	*abcda*	6
5	*babcabcda*	2
6	*bcabcda*	4
7	*bcda*	7
8	*cabcda*	5
9	*cda*	8
10	*da*	9

Суффиксный массив можно построить за время $O(n \log n)$, где n — длина строки, если воспользоваться модификацией поразрядной сортировки. Для начала отсортируем суффиксы по их первому символу, используя черпачную сортировку [2]. Трудоёмкость этой операции будет $O(n)$. Выделим группы суффиксов с совпадающим первым символом. Запомним для каждого суффикса $T(i, |T|)$ номер такой группы $g(i)$. Результат операции будет выглядеть следующим образом:

Индекс массива	$T(i, \|T\|)$	i	$g(i)$
1	*ababcabcda*	1	1
2	*abcabcda*	3	1
3	*abcda*	6	1
4	*a*	10	1
5	*babcabcda*	2	2
6	*bcabcda*	4	2
7	*bcda*	7	2
8	*cabcda*	5	3
9	*cda*	8	3
10	*da*	9	4

Очевидно, что при дальнейшей сортировке относительный порядок суффиксов может поменяться только внутри одной группы. Если отбросить первый совпадающий символ у всех суффиксов одной группы, то полученные строки также будут суффиксами исходной строки. Более того, мы знаем, в каком порядке эти суффиксы будут располагаться, если их отсортировать по первому символу. Это позволяет нам с помощью черпачной сортировки упорядочить суффиксы уже по двум символам. Для этого каждому суффиксу поставим в соответствие упорядоченную пару чисел $(g(i), g(i + \ell))$: номер группы теку-

щего суффикса и номер группы суффикса, который получается при отбрасывании уже учтённой части из ℓ символов. Номер группы пустой строки будем считать равным 0. Для данного примера получим следующие пары:

Индекс массива	$T(i, \|T\|)$	i	$(g(i), g(i+1))$
1	*abababcda*	1	(1, 2)
2	*abcabcda*	3	(1, 2)
3	*abcda*	6	(1, 2)
4	*a*	10	(1, 0)
5	*babcabcda*	2	(2, 1)
6	*bcabcda*	4	(2, 3)
7	*bcda*	7	(2, 3)
8	*cabcda*	5	(3, 1)
9	*cda*	8	(3, 4)
10	*da*	9	(4, 1)

Теперь полученные пары чисел можно упорядочить за время $\mathrm{O}(n)$ с помощью поразрядной сортировки и получить последовательность, упорядоченную по первым двум символам. При этом каждому суффиксу требуется приписать новый номер его группы:

Индекс массива	$T(i, \|T\|)$	i	$(g(i), g(i+1))$	$g'(i)$
1	*a*	10	(1, 0)	1
2	*ababcabcda*	1	(1, 2)	2
3	*abcabcda*	3	(1, 2)	2
4	*abcda*	6	(1, 2)	2
5	*babcabcda*	2	(2, 1)	3
6	*bcabcda*	4	(2, 3)	4
7	*bcda*	7	(2, 3)	4
8	*cabcda*	5	(3, 1)	5
9	*cda*	8	(3, 4)	6
10	*da*	9	(4, 1)	7

Далее аналогичным образом рассмотрим подпоследовательности суффиксов с совпадающими первыми двумя символами и получим массив суффиксов, отсортированных по первым четырём символам:

Индекс массива	$T(i, \|T\|)$	i	$(g'(i), g'(i+2))$
1	*a*	10	(1, 0)
2	*ababcabcda*	1	(2, 2)

Индекс массива	$T(i, \|T\|)$	i	$(g'(i), g'(i+2))$
3	*abcabcda*	3	(2, 5)
4	*abcda*	6	(2, 6)
5	*babcabcda*	2	(3, 4)
6	*bcabcda*	4	(4, 2)
7	*bcda*	7	(4, 7)
8	*cabcda*	5	(5, 4)
9	*cda*	8	(6, 1)
10	*da*	9	(7, 0)

В каждой итерации мы можем увеличивать количество отсортированных символов в 2 раза. Тогда построение суффиксного массива потребует $\mathrm{O}(\log n)$ таких итераций, а общая трудоёмкость построения будет равна $\mathrm{O}(n \log n)$, где n — длина строки.

Пусть $(a_i)_{i=1}^n$ — последовательность номеров всех суффиксов, ℓ — количество уже упорядоченных символов, $g(i)$ — номер группы, в которую входит суффикс $T(i, |T|)$, $\big(p(i), q(i)\big)$ — пара чисел, которая ставится суффиксу в соответствие. Тогда псевдокод для каждой итерации будет выглядеть следующим образом:

Итерация построения суффиксного массива

```
// Упорядочивает элементы последовательности (a_i)_{i=1}^n
// по неубыванию соответствующих значений p_i.
// При равенстве p_i и p_j относительный порядок
// a_i и a_j не изменяется.
Функция Упорядочить: (последовательность (a_i)_{i=1}^n,
        последовательность (p_i)_{i=1}^n) → последовательность
    m ← 0.
    Для всех i от 1 до n
        m ← max{m, p_i}.
    Для всех i от 0 до m
        c(i) ← 0.
    Для всех i от 1 до n
        c(p_i) ← c(p_i) + 1.

    h(0) ← 0.
    Для всех j от 1 до m
```

$h(j) \leftarrow h(j-1) + c(j-1)$.

Для всех i **от** 1 **до** n
$b(h(p_i)) \leftarrow a_i$.
$h(p_i) \leftarrow h(p_i) + 1$.
Вернуть b.

Функция Построить: (**последовательность** $(a_i)_{i=1}^n$, **последовательность** g, **целое** ℓ) $\rightarrow$ $\rightarrow$ (**последовательность**, **последовательность**)
Для всех i **от** 1 **до** n
Если $i + \ell \leqslant n$
$p(i) \leftarrow g(i + \ell)$.
Иначе
$p(i) \leftarrow 0$.

$(a_i)_{i=1}^n \leftarrow$ Упорядочить$((a_i)_{i=1}^n, p)$.
$(a_i)_{i=1}^n \leftarrow$ Упорядочить$((a_i)_{i=1}^n, g)$.

$q(a_1) \leftarrow 1$.
Для всех i **от** 2 **до** n
Если $(g(a_i), p(a_i)) = (g(a_{i-1}), p(a_{i-1}))$
$q(a_i) \leftarrow q(a_{i-1})$.
Иначе
$q(a_i) \leftarrow q(a_{i-1}) + 1$.
Вернуть $((a_i)_{i=1}^n, q)$.

2.5.2. On-line задача поиска образца в строке

Полученный суффиксный массив можно использовать для поиска подстрок. Пусть есть суффиксный массив $(a_i)_{i=1}^n$, построенный по тексту T, и некоторый образец S, вхождение которого нужно найти в тексте. Тот факт, что суффиксный массив хранит суффиксы в лексикографическом порядке, может быть использован для поиска подстрок. Пусть есть диапазон $[\ell, r]$ суффиксов $(a_i)_{i=\ell}^r$, где ℓ и r — индексы суффиксов. Начнём поиск с диапазона $[1, n]$, где $n = |T|$, и первого сим-

вола образца s_1. Если рассмотреть первые символы всех суффиксов в массиве, то можно заметить, что они идут в неубывающем порядке. Зная это, можно использовать бинарный поиск, для того чтобы найти символ s_1 среди первых символов суффиксов. Если символ не будет найден, то искомый образец в тексте не встречается. Если же символ удастся найти среди первых символов суффиксов, то можно сузить рассматриваемый диапазон $[1, n]$ до поддиапазона $[\ell, r]$, состоящего только из тех суффиксов, у которых первый символ совпадает с s_1. Аналогичным образом можно продолжить поиск в диапазоне $[\ell, r]$, рассматривая символ s_2 и последующие, пока не получим пустой диапазон либо не дойдём до конца S. Псевдокод поиска образца S в тексте T по его суффиксному массиву $(a_i)_{i=1}^n$ будет выглядеть следующим образом:

Поиск образца в тексте

Функция Найти: (**строка** T, **последовательность** $(a_i)_{i=1}^n$, **строка** S) $\to$ **множество целых**

$\ell \leftarrow 1$.
$r \leftarrow n$.
Для всех k **от** 1 **до** $|S|$
    $(\ell, r) \leftarrow (\min\{ i \in [\ell, r] \mid t_{a_i+k-1} = s_k \},$
        $\max\{ i \in [\ell, r] \mid t_{a_i+k-1} = s_k \})$.
    Если $\ell > r$
        Вернуть $\varnothing$.
Ответ $\leftarrow \varnothing$.
Для всех i **от** ℓ **до** r
    Ответ $\leftarrow$ ответ $\cup \{a_i\}$.
Вернуть ответ.

Пример 2.7. Рассмотрим текст «*ababcabcda*» и образец «*bcab*». На первой итерации можно сузить диапазон $[1, 10]$ до $[5, 7]$:

Индекс массива	$T(i, \lvert T\rvert)$
1	*a*
2	*ababcabcda*
3	*abcabcda*
4	*abcda*
5	*babcabcda*

Индекс массива	$T(i, \|T\|)$
6	$\underline{b}cabcda$
7	$\underline{b}cda$
8	$cabcda$
9	cda
10	da

На второй итерации перейдём к диапазону $[6, 7]$:

Индекс массива	$T(i, \|T\|)$
5	$babcabcda$
6	$\underline{bca}$bcda
7	$\underline{bcd}$a

На третьей итерации получим диапазон $[6, 6]$, состоящий из единственного суффикса — найдено совпадение:

Индекс массива	$T(i, \|T\|)$
6	$\underline{bcab}cda$
7	$bcda$

2.6. ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

1. Циклический сдвиг. Циклическим сдвигом строки на K символов будем называть строку, которая получается из исходной строки переносом первых K символов в конец строки. Необходимо определить, можно ли получить из одной строки другую при помощи некоторого циклического сдвига.

2. Период. Для каждого префикса данной строки S, состоящей из n символов, необходимо найти его наименьший период. Это означает, что для каждого i от 2 до n нужно узнать максимальное k, для которого префикс $S(1, i)$ может быть записан как некоторая строка A, повторённая k раз.

3. Телефонные номера. В окружающем мире вы часто встречаете много телефонных номеров, и они становятся всё длиннее. Вам надо запомнить некоторые из них. Один из способов, чтобы легче это сделать, — соотнести буквы с цифрами, как показано в таблице:

1	I J
2	A B C
3	D E F
4	G H
5	K L
6	M N
7	P R S
8	T U V
9	W X Y
0	O Q Z

Таким образом, каждый номер может быть сопоставлен одному слову или группе слов, и вы легко запомните слова вместо номеров. Очевидно, что особенно приятно, если есть возможность найти какую-то простую связь между словом и самим человеком. Например, номер вашего друга по шахматной игре:

```
9 4 1 8 3 7 2 9 6
W H I T E P A W N
```

или номер вашего любимого учителя:

```
2 8 5 5 3 0 4
B U L L D O G
```

Необходимо разработать алгоритм для нахождения самой короткой последовательности слов из заданного словаря, которая соответствует данному номеру.

4. Интернет-форум. Вася разработал новый интернет-форум для небольшого новостного портала. К сожалению, не все посетители форума являются культурными людьми, поэтому заказчик попросил Васю добавить в форум систему фильтрации нецензурных сообщений. Сообщение форума относится к нецензурным, если в нём есть хотя бы одно нецензурное слово. Слово считается нецензурным, если оно содержит в качестве подстроки нецензурное буквосочетание из словаря, но при этом его нет в словаре цензурных слов. Предварительно Вася составил два словаря: словарь нецензурных буквосочетаний и словарь цензурных слов. Помогите Васе разработать алгоритм, который определит, является ли сообщение форума нецензурным.

Глава 3. ГРАФОВЫЕ АЛГОРИТМЫ

Графы являются весьма популярными комбинаторными объектами, получившими широкое применение в самых разнообразных областях науки и практики: экономике, физике, химии, молекулярной биологии, программировании, логистике, сетевом и календарном планировании, проектировании интегральных схем и сетей оптико-волоконной связи, распределении ресурсов и многих других. Язык теории графов позволяет упростить формулировки достаточно сложных задач и предоставляет эффективный инструмент для их исследования.

Существует большое количество графовых задач. В практических приложениях чаще всего возникают задачи о специальных размещениях и укладках графов на различных геометрических объектах (задача об оптимальной укладке графа на линии); задачи поиска в графах наибольших подграфов с заданным свойством (задача о наибольшем паросочетании в графе); задачи нахождения в графах специальных маршрутов, обладающих какой-либо экстремальной характеристикой (задача о кратчайшей цепи между заданной парой вершин графа). Многие из этих задач хорошо изучены в литературе, и для их решения разработаны эффективные алгоритмы. В этой главе рассматривается несколько классических задач такого рода и приводятся алгоритмы их решения.

3.1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

В настоящем разделе даётся краткая сводка основных теоретико-графовых определений и базовых фактов, примыкающих к понятию графа и используемых далее в пособии.

Неориентированный граф (или просто *граф*) — упорядоченная пара (V, E), где V — непустое конечное множество (элементы которого называются *вершинами*), а E — множество неупорядоченных пар вер-

шин. Неупорядоченная пара $\{u, v\} \in E$ вершин u и v ($u \neq v$) называется *ребром* графа (часто такое ребро обозначают uv). Множество вершин графа G обозначают $V(G)$, а множество его рёбер — $E(G)$.

Если $e = uv$ — ребро, то вершины u и v называют его *концами* и говорят, что вершины u и v *смежны*. Множество всех вершин графа G, смежных с данной вершиной v, называется *окружением* вершины v и обозначается через $N(v)$. Два ребра — *смежные*, если они имеют общий конец. Ребро и вершину называют *инцидентными*, если вершина является концом данного ребра.

Обобщением графа является *мультиграф*, в котором допускается наличие *кратных рёбер* — рёбер, соединяющих одну и ту же пару вершин. Дальнейшим обобщением служит *псевдограф* — мультиграф, в котором могут присутствовать *петли* — рёбра, которые соединяют вершину с ней самой. В дальнейшем, говоря о графе, будем предполагать, что он не содержит кратных рёбер и петель.

Маршрутом в графе между заданной парой вершин v_1 и v_{k+1} называется последовательность вершин вида $v_1, v_2, \ldots, v_{k+1}$, где $v_i v_{i+1} \in E$ при $i = 1, \ldots, k$ (часто такой маршрут называют (v_1, v_{k+1})-*маршрутом*). Число k рёбер маршрута называется его *длиной*. Маршрут называется *замкнутым*, если $v_1 = v_{k+1}$.

Цепь в графе — маршрут, в котором каждое ребро графа повторяется не более одного раза. *Простая цепь* в графе — цепь, в которой каждая вершина повторяется не более одного раза. *Цикл* в графе — замкнутая цепь. *Простой цикл* в графе — простая замкнутая цепь.

Граф H называется *подграфом* графа G, если выполняются включения $V(H) \subseteq V(G)$ и $E(H) \subseteq E(G)$.

Граф называется *связным*, если любые две его несовпадающие вершины соединены маршрутом. *Деревом* называется связный граф без циклов.

В дальнейшем будем предполагать, что $|V| = n$ и $|E| = m$. Число вершин графа называется его *порядком*. Если граф не содержит рёбер, то он является *пустым*. Граф называется *полным*, если любые две его вершины смежны (полный граф порядка n обозначается K_n). На рис. 3.1 приведён полный граф порядка 5.

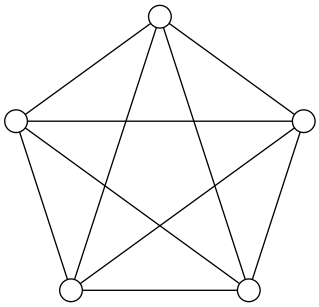


Рис. 3.1. Полный граф K_5

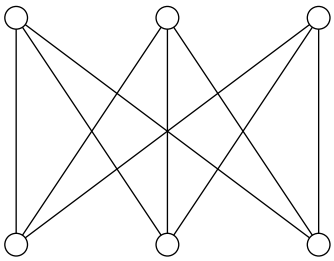


Рис. 3.2. Полный двудольный граф $K_{3,3}$

Граф называется *двудольным*, если множество его вершин можно разбить на два подмножества (*доли*) таким образом, что концы каждого ребра принадлежат разным подмножествам. Критерий двудольности графа даёт известная теорема Кёнига (1936), в которой утверждается, что для двудольности графа необходимо и достаточно, чтобы он не содержал циклов нечётной длины. Двудольный граф G, для которого первая доля задаётся подмножеством X, а вторая — подмножеством Y $(X \cup Y = V)$, обозначают $G(X, Y, E)$.

Граф называется *полным двудольным*, если каждая вершина одной доли смежна с каждой вершиной другой доли (полный двудольный граф, доли которого состоят из p и q вершин, обозначают $K_{p,q}$). На рис. 3.2 приведён полный двудольный граф, каждая доля которого содержит по 3 вершины.

Паросочетанием в графе называется произвольное подмножество его попарно несмежных рёбер. Паросочетание является *максимальным*, если оно не содержится в паросочетании с бóльшим числом рёбер, и *наибольшим*, если число рёбер в нём наибольшее среди всех паросочетаний графа. Паросочетание M называется *совершенным*, если каждая вершина графа инцидентна ровно одному ребру из M, т. е. M покрывает все вершины графа. На рис. 3.3, *а* выделены рёбра некоторого максимального паросочетания, которое не является наибольшим, так как существует паросочетание с бóльшим числом рёбер. Паросочетание на рис. 3.3, *б* является совершенным паросочетанием и, следовательно, наибольшим.

Пусть G и H — графы, а отображение $\varphi\colon V(G) \to V(H)$ — биекция, которая обладает следующим свойством: для любых двух вершин u

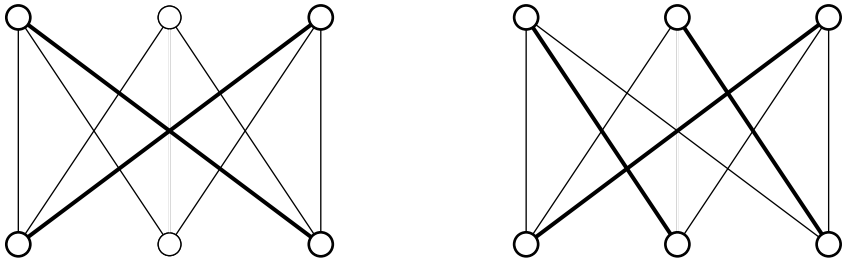


Рис. 3.3. Паросочетания в графе:
а — максимальное; *б* — совершенное

и v графа G эти вершины смежны в G тогда и только тогда, когда их образы $\varphi(u)$ и $\varphi(v)$ смежны в графе H, т. е.

$$\{u, v\} \in E(G) \Leftrightarrow \{\varphi(u), \varphi(v)\} \in E(H).$$

Отображение φ в этом случае называется *изоморфизмом графа G на граф H* (или просто *изоморфизмом*), а сами графы G и H — *изоморфными*. На рис. 3.4 приведены изоморфные графы.

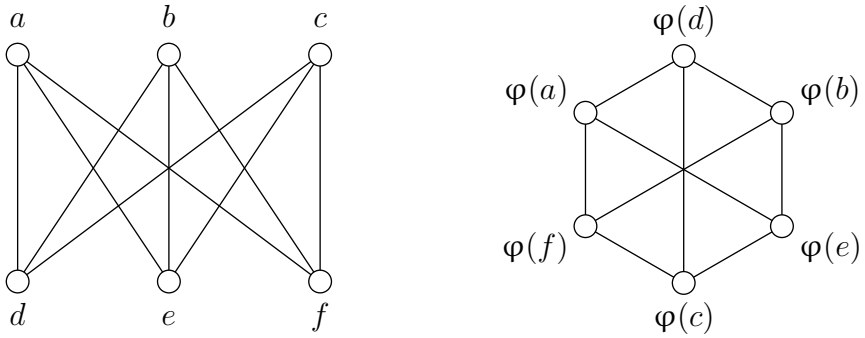
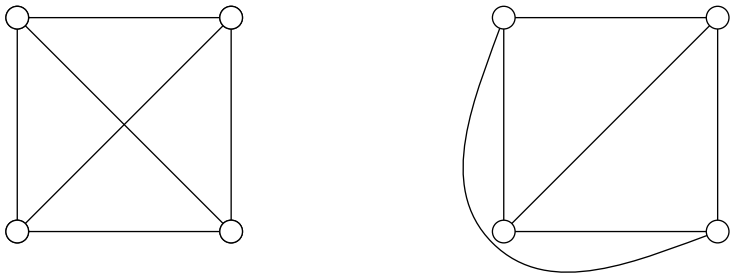


Рис. 3.4. Изоморфизм графов

Для того чтобы различать изоморфные графы, вводят понятие «помеченный граф». Граф порядка n назовём *помеченным*, если всем его вершинам присвоены метки, например целые числа $1, 2, \ldots, n$. Говорят, что два помеченных графа G и H, заданные на одном и том же множестве $\{1, 2, \ldots, n\}$ вершин, *равны*, если выполняется равенство $E(G) = E(H)$.

Изоморфные графы несут одну и ту же информацию, поэтому в некоторых ситуациях не имеет значения, как изобразить граф. Однако для многих практических приложений, например в радиоэлектронике при изготовлении микросхем, важно выяснить, можно ли изобразить



а *б*

Рис. 3.5. Планарность графов:
а — планарный граф; *б* — изоморфный ему плоский граф

граф на плоскости определённым образом. *Плоским* графом называется граф, вершины которого являются точками плоскости, а рёбра — непрерывные плоские линии без самопересечений, соединяющие вершины графа так, что никакие два ребра не имеют общих точек, кроме инцидентной им обоим вершины (рис. 3.5, *б*). Граф, изоморфный плоскому графу, называют *планарным* графом (рис. 3.5, *а*).

Теорема 3.1. *Для связного планарного графа* $G = (V, E)$, $|V| = n$, $n \geqslant 3$, $|E| = m$ *выполняется следующее неравенство:*

$$m \leqslant 3n - 6. \tag{3.1}$$

Если известно, что граф планарный, то для чисел n и m его вершин и рёбер будет выполняться соотношение (3.1). С другой стороны, если для некоторого графа известно, что $m > 3n - 6$ при $n \geqslant 3$, то сразу можно сделать вывод о том, что такой граф не является планарным. Однако если для некоторого графа соотношение (3.1) выполняется, то для выяснения вопроса о его планарности необходимы дальнейшие исследования.

Первым критерием планарности графов является *критерий Понтрягина – Куратовского*: граф планарен тогда и только тогда, когда он не содержит подграфов, гомеоморфных K_5 или $K_{3,3}$. Напомним, что два графа называются *гомеоморфными*, если они могут быть получены из одного и того же графа подразбиением его рёбер (подразбиение ребра vw заключается в удалении этого ребра из графа и добавлении двух новых рёбер: vu и uw, где u — новая вершина графа). Алгоритмически реализация критерия Понтрягина – Куратовского для проверки планарности графа не может быть выполнена за полиномиальное

время (если верна гипотеза об экспоненциальном времени). Однако существуют полиномиальные алгоритмы для проверки планарности графа. В 1970 г. алгоритм Хопкрофта – Тарьяна [4] стал первым линейным алгоритмом определения планарности графа. Алгоритм Хопкрофта – Тарьяна за время $O(n)$ проверяет граф на планарность и, если он планарен, производит его плоскую укладку.

Пусть $G = (V, E)$ — граф и $w\colon E \to \mathbb{R}^+$ — функция, ставящая в соответствие каждому ребру $e \in E$ неотрицательное число $w(e)$ — вес (или длину) ребра e. Пара (G, w) называется *взвешенным графом*. Под *весом* (или *длиной*) любого *подграфа* H графа (G, w) будем понимать сумму весов его рёбер, т. е.

$$w(H) = \sum_{e \in E(H)} w(e).$$

Ориентированный граф (*орграф*) $G = (V, A)$ состоит из непустого множества вершин V и множества упорядоченных пар вершин A (в дальнейшем будем предполагать, что $|V| = n$ и $|A| = m$). Упорядоченная пара вершин (u, v) называется *дугой*. Если (u, v) — дуга, то вершины u и v называют *началом* и *концом* дуги соответственно. В дальнейшем будем предполагать, что в орграфе отсутствуют кратные дуги и петли.

Если в графе допускаются рёбра и дуги, то такой граф называют *смешанным* графом.

Ориентированным маршрутом в орграфе (или просто *маршрутом*) между заданной парой вершин v_1 и v_{k+1} называется последовательность вершин вида $v_1, v_2, \ldots, v_{k+1}$, где $(v_i, v_{i+1}) \in A$ при $i = 1, \ldots, k$. Маршрут является *замкнутым*, если $v_1 = v_{k+1}$.

Цепь в орграфе — ориентированный маршрут, в котором каждая дуга орграфа повторяется не более одного раза.

Путь в орграфе — цепь, в которой каждая вершина повторяется не более одного раза. *Контур* в орграфе — замкнутый путь.

Корневое дерево — орграф, который удовлетворяет следующим условиям:

1) имеется в точности одна вершина, в которую не входит ни одна дуга (*корень* дерева);

2) в каждую вершину, кроме корня, входит ровно одна дуга;

3) из корня дерева есть путь к каждой вершине.

Пусть (u, v) — дуга корневого дерева, тогда вершину u называют *отцом* вершины v, а вершину v — *сыном* вершины u. Если у вершины u нет сыновей, то говорят, что вершина u — *лист* корневого дерева. Вершины корневого дерева, отличные от листьев, называют *внутренними* вершинами.

Пусть u — некоторая вершина корневого дерева, тогда все вершины v, которые достижимы из вершины u (т. е. существует путь из u в v), называют *потомками* вершины u. Подграф, состоящий из всех потомков вершины u (включая вершину u) и всех исходящих из них дуг, называется *поддеревом с корнем в вершине* u и обозначается через T^u.

3.2. УКЛАДКА КОРНЕВЫХ ДЕРЕВЬЕВ

На практике в различных приложениях, например таких, как решение больших разреженных систем линейных алгебраических уравнений и проектирование интегральных схем, часто возникает необходимость в укладке вершин графа в целочисленных точках прямой с требованием минимизировать сумму модулей разностей между номерами смежных вершин — длину укладки графа. В задаче о ширине укладки графа требуется разместить вершины графа в целочисленных точках прямой так, чтобы минимизировать максимальное значение модуля разности номеров смежных вершин. Задачи о длине и ширине укладки графа относятся к задачам поиска минимальной нумерации вершин графа. Общим для всех задач данного класса является вид допустимого решения — нумерация $\varphi\colon V(G) \to P$ — вершин графа G, а отличаются они друг от друга лишь оптимизируемой функцией. В задачах об укладках минимальной длины и минимальной ширины n-вершинных графов можно считать, не ограничивая общности, что $P = \{1, 2, \ldots, n\}$, поскольку известно, что в этом случае оптимальное значение длины (соответственно, ширины) укладки не больше, чем при выборе любой другой нумерующей последовательности. В общем случае рассматриваемые задачи являются NP-трудными [5], поэтому особый интерес

представляет выделение классов графов, для которых эти задачи являются полиномиально разрешимыми. В настоящем разделе рассмотрены задачи укладки минимальной длины и минимальной ширины для корневых деревьев и приводятся эффективные алгоритмы их решения.

3.2.1. Постановка задачи

Определение 3.1. Для ориентированного графа $G = (V, A)$ порядка n нумерация $P = \{1, 2, \ldots, n\}$ его вершин называется *допустимой нумерацией*, если она удовлетворяет свойству

$$\mathrm{Number}_P(u) > \mathrm{Number}_P(v), \quad \forall\, (u, v) \in A,$$

где $\mathrm{Number}_P(u)$ — номер вершины u в нумерации P.

Определение 3.2. Допустимая нумерация вершин корневого дерева T порядка n называется *укладкой корневого дерева*.

Для корневого дерева T^a (с корнем в вершине a), изображённого на рис. 3.6, приведены две укладки: P_1 и P_2 (рис. 3.7).

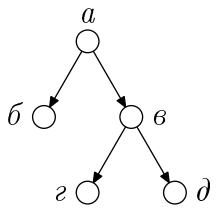


Рис. 3.6. Корневое дерево T^a

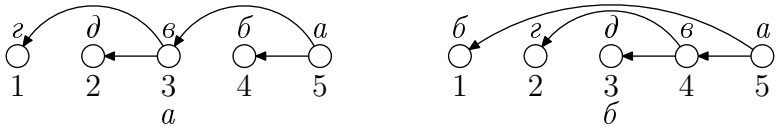


Рис. 3.7. Укладки дерева T^a: *а* — укладка P_1; *б* — укладка P_2

Для укладки корневого дерева можно определить некоторые параметры. Распространёнными являются длина и ширина укладки.

Обозначим через $P(T)$ множество всех допустимых нумераций вершин корневого дерева T. Не умаляя общности, будем считать, что имя вершины в укладке P' совпадает с её номером.

Для дерева T, укладки P' и вершины i определим множество дуг $\mathrm{Cut}_i(P', T)$, которые обладают следующим свойством:

$$\mathrm{Cut}_i(P', T) = \varnothing, \text{ при } i = 1,$$
$$\mathrm{Cut}_i(P', T) = \big\{(u, v) \in A(T) \;\big|\; u \in \{i, i+1, \ldots, n\}, \; v \in \{1, 2, \ldots, i-1\}\big\}, \text{ при } 2 \leqslant i \leqslant n.$$

Для $P' \in P(T)$ величину

$$\mathrm{Length}(P', T) = \sum_{(u,v)\in A(T)} \big(\mathrm{Number}_{P'}(u) - \mathrm{Number}_{P'}(v)\big)$$

называют *длиной укладки P' корневого дерева T*, а величину

$$\mathrm{Width}(P', T) = \max_{i\in V(T)} \big|\,\mathrm{Cut}_i(P', T)\big| -$$

шириной укладки P' корневого дерева T.

Задача укладки корневого дерева минимальной длины состоит в том, чтобы найти такую нумерацию P_0, для которой выполняется

$$\mathrm{Length}(P_0, T) = \min_{P'\in P(T)} \mathrm{Length}(P', T).$$

Задача укладки корневого дерева минимальной ширины заключается в том, чтобы найти нумерацию P_0, для которой выполняется

$$\mathrm{Width}(P_0, T) = \min_{P'\in P(T)} \mathrm{Width}(P', T).$$

Для дерева T^a длина укладки P_1 (рис. 3.7, *а*) равна 6, а ширина — 2; для укладки P_2 (рис. 3.7, *б*) длина равна 8, а ширина — 3. Можно показать, что укладка P_1 является укладкой минимальной длины и ширины дерева T^a.

Задача укладки корневого дерева — одна из возможных математических моделей для ряда прикладных задач. В качестве примера рассмотрим *задачу оптимизации вычисления функции*, которая задана в виде корневого дерева:

1) корню дерева соответствует главная функция, значение которой требуется вычислить;

2) каждая внутренняя вершина u корневого дерева является функцией, для вычисления значения которой необходимо, чтобы были известны и находились в памяти компьютера все её аргументы, соответствующие в дереве таким вершинам v, что $(u, v) \in A(T)$;

3) предполагаем, что вычисление каждой функции выполняется мгновенно, а для размещения аргумента в памяти компьютера необходима одна единица времени; вычисленное значение хранится в памяти до тех пор, пока оно не будет использовано для вычисления функции;

4) под временем хранения значения будем понимать количество тактов (единиц времени), в течение которых это значение находится в памяти до его использования при вычислении функции в качестве аргумента.

Нетрудно заметить, что в такой постановке одной из возможных математических моделей задачи является оптимальная укладка корневого дерева:

• укладка корневого дерева минимальной длины соответствует организации вычислений, при которой обеспечивается вычисление главной функции с минимальным суммарным временем хранения всех данных;

• укладка корневого дерева минимальной ширины соответствует организации вычислений, при которой обеспечивается вычисление главной функции с использованием минимального количества ячеек памяти.

Одним из известных подходов для обоснования корректности алгоритмов решения задач является *перестановочный метод*, который мы продемонстрируем при решении задачи оптимальной укладки.

Теорема 3.2. *Оптимальную (по длине, ширине) укладку корневого дерева T можно искать на множестве укладок, в которых каждое из его поддеревьев будет иметь последовательную нумерацию.*

Доказательство. Предположим, что есть некоторая укладка P' дерева T, в которой x — корень поддерева с минимальным номером. Докажем, что в оптимальном решении вершины этого поддерева имеют последовательную нумерацию.

Предположим, что это не так (рис. 3.8). Перенумеруем вершины поддерева с корнем в вершине x и вершину v, принадлежащую поддереву с корнем в вершине z, как показано на рис. 3.9. В результате получим новую укладку P'', в которой вершины поддерева с корнем в вершине x имеют последовательную нумерацию вершин.

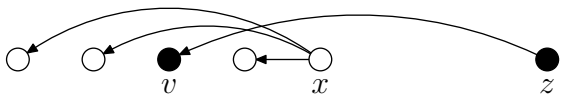


Рис. 3.8. Фрагмент укладки P'

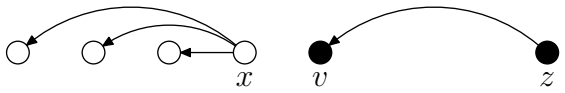


Рис. 3.9. Фрагмент укладки P''

Для укладки P'' не увеличится длина (ширина) укладки поддеревьев с корнями в вершинах x и z, и очевидно, что не увеличится и длина (ширина) укладки всего дерева. Теперь сведём нашу задачу к задаче с меньшим количеством вершин, удаляя из укладки P'' поддерево с корнем в вершине x. □

3.2.2. Укладка корневого дерева минимальной длины

При рассмотрении задачи укладки корневого дерева минимальной длины теорема 3.2 позволяет сузить всю область допустимых решений этой задачи на класс укладок, состоящий только из тех, в которых вершины каждого из поддеревьев имеют последовательную нумерацию.

Предположим, что у корня x дерева T^x есть сыновья $x_0, x_1, \ldots, x_k$, которые являются корнями поддеревьев $T_0^x, T_1^x, \ldots, T_k^x$ соответственно. Пусть $\mathrm{Length}(T_i^x)$ — длина оптимальной укладки поддерева T_i^x, в котором $\overline{k}(x_i)$ вершин $(i = 0, \ldots, k)$. Тогда при укладке поддеревьев вершины x в порядке $T_0^x, T_1^x, T_2^x, \ldots, T_k^x$ (рис. 3.10) длина укладки всего дерева T^x равна

$$\sum_{i=0}^{k} \big(\mathrm{Length}(T_i^x) + i \cdot \overline{k}(x_i) \big) + (k+1).$$

Для оптимальной (по длине) укладки корневого дерева необходимо определить порядок её поддеревьев, при котором будет минимальна длина укладки всего дерева. Несложно показать, что для минимизации длины укладки корневого дерева T^x достаточно, чтобы

$$\overline{k}(x_0) \geqslant \overline{k}(x_1) \geqslant \ldots \geqslant \overline{k}(x_{k-1}) \geqslant \overline{k}(x_k).$$

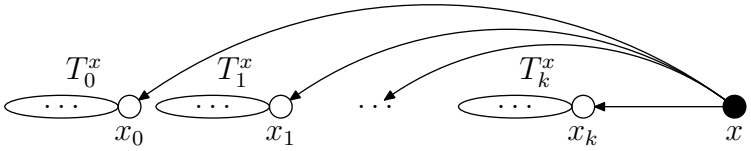


Рис. 3.10. Укладка дерева T^x

Для доказательства данного утверждения можно использовать, например, перестановочный метод. Перестановочный метод на этот раз сужает класс допустимых решений к единственному элементу, задающему укладку корневого дерева минимальной длины. Таким образом, справедлива следующая теорема.

Теорема 3.3. *Для минимизации длины укладки корневого дерева T, необходимо для любой внутренней вершины дерева сначала выполнить оптимально укладку её поддеревьев в порядке, когда первым укладывается поддерево с наибольшим количеством вершин, а последним — с наименьшим.*

Приведём сейчас один из возможных алгоритмов для решения задачи оптимальной по длине укладки корневого дерева.

Алгоритм укладки корневого дерева минимальной длины

1. На начальном этапе припишем каждой вершине u дерева T метку $\mathrm{Size}(u)$, равную количеству вершин в дереве, корнем которого она является. Для определения этих меток можно использовать поиск в глубину, начинающийся из корня дерева. Во время поиска в глубину при удалении вершины u из стека присваиваем ей метку $\mathrm{Size}(u)$ по следующему правилу:

- если u — лист, то $\mathrm{Size}(u) = 1$;
- если u — внутренняя вершина, то

$$\mathrm{Size}(u) = \sum_{v\colon (u,v)\in A(T)} \mathrm{Size}(v) + 1.$$

2. Для оптимальной нумерации вершин дерева T будем использовать поиск в глубину с приоритетами, начинающийся из корня дерева. Во время поиска в глубину с приоритетами:

- из текущей вершины u путь идёт в вершину v ($(u,v) \in A(T)$), которая имеет наибольшую метку $\mathrm{Size}(v)$, так как для минимизации длины укладки сначала надо выполнить укладку поддерева с наибольшим количеством вершин;
- при удалении вершины u из стека полагаем $\mathrm{Number}_P(u) = k$ (первоначально берём $k = 1$) и увеличиваем счётчик k на 1.

Трудоёмкость приведённого выше алгоритма укладки корневого дерева минимальной длины складывается из трудоёмкости вычисления меток $\mathrm{Size}(u)$ всех вершин u дерева T (метки могут быть вычислены за время $\mathrm{O}(n)$) и трудоёмкости поиска в глубину ($\mathrm{O}(n)$). Следовательно, трудоёмкость приведённого алгоритма укладки корневого дерева минимальной длины равна $\mathrm{O}(n)$.

Пример 3.1. Рассмотрим задачу оптимизации вычисления функции, которая задана в виде корневого дерева, приведённого на рис. 3.11. Требуется определить порядок вычисления функции, который обеспечивает минимальное суммарное время хранения всех промежуточных данных в памяти компьютера.

Одной из возможных математических моделей для данной задачи является оптимальная (по длине) укладка корневого дерева. Выполним укладку P минимальной длины для корневого дерева, приведённого на рис. 3.11. Длина оптимальной укладки P равна 28 (рис. 3.12).

В соответствии с укладкой P в таблице для каждого такта времени показаны промежуточные данные, хранящиеся в памяти компьютера:

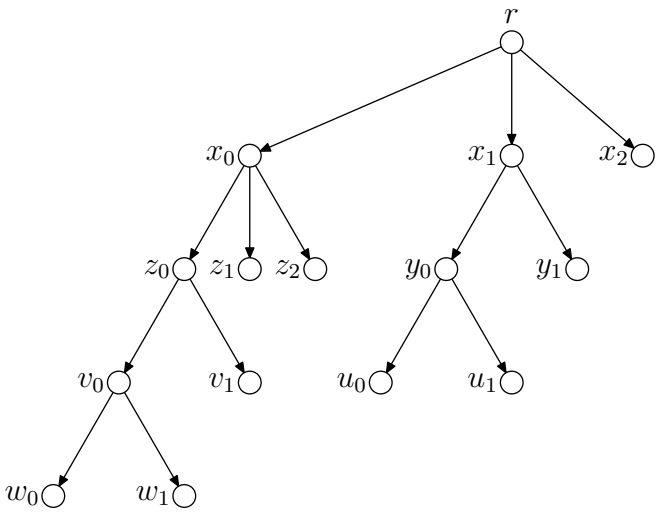


Рис. 3.11. Корневое дерево T^r

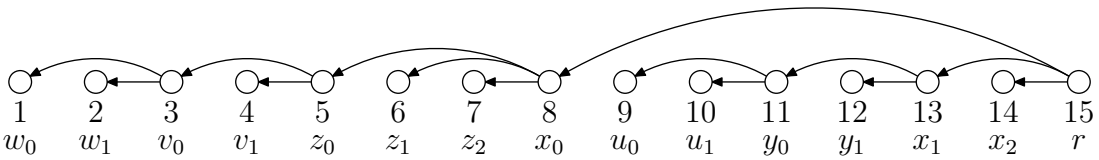


Рис. 3.12. Укладка P минимальной длины 28 дерева T^r

Такт	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Состояние памяти	w_0	w_0 w_1	v_0	v_0 v_1	z_0	z_0 z_1	z_0 z_1 z_2	x_0	x_0 u_0	x_0 u_0 u_1	x_0 y_0	x_0 y_0 y_1	x_0 x_1	x_0 x_1 x_2
Количество данных в памяти	1	2	1	2	1	2	3	1	2	3	2	3	2	3

На последнем этапе (15-й такт) происходит вычисление главной функции r (к этому моменту времени все аргументы x_0, x_1 и x_2 главной функции уже вычислены и хранятся в памяти компьютера).

Сумма чисел в последней строке таблицы равна 28 и соответствует суммарному времени хранения в памяти компьютера всех данных, требуемых для вычисления главной функции r в порядке, который определён оптимальной укладкой P.

3.2.3. Укладка корневого дерева минимальной ширины

При рассмотрении задачи укладки корневого дерева минимальной ширины теорема 3.2 позволяет сузить всю область допустимых решений этой задачи на класс укладок, где рассматриваются только те, в которых каждое из поддеревьев укладывается последовательно.

Предположим, что у корня x дерева T^x есть сыновья x_0, x_1, …, x_k, которые являются корнями поддеревьев T_0^x, T_1^x, …, T_k^x соответственно. Пусть $\mathrm{Width}(T_i^x)$ — ширина оптимальной укладки поддерева T_i^x при $i = 0, \ldots, k$. Тогда при укладке поддеревьев вершины x в порядке $T_0^x, T_1^x, T_2^x, \ldots, T_k^x$ (рис. 3.13) ширина укладки дерева T^x равна

$$\mathrm{Width}(T^x) = \max\big\{\, \mathrm{Width}(T_0^x), \mathrm{Width}(T_1^x) + 1, \ldots, \\ \mathrm{Width}(T_k^x) + k, k + 1\big\}.$$

Для оптимальной (по ширине) укладки дерева необходимо определить порядок укладки её поддеревьев, при котором будет минимальна

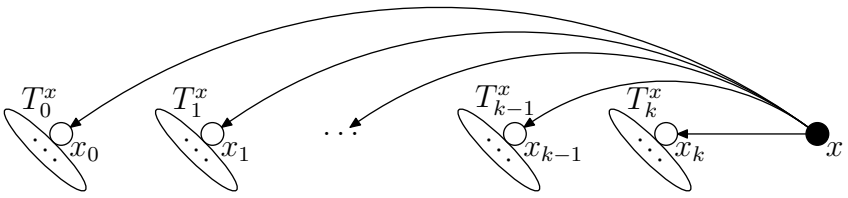


Рис. 3.13. Укладка P дерева T^x

ширина укладки всего дерева. Несложно показать, что для минимизации ширины укладки дерева T^x достаточно, чтобы

$$\mathrm{Width}(T_0^x) \geqslant \mathrm{Width}(T_1^x) \geqslant \ldots \geqslant \mathrm{Width}(T_k^x).$$

Для доказательства данного утверждения можно использовать, например, перестановочный метод. Таким образом, справедлива следующая теорема.

Теорема 3.4. *Для минимизации ширины укладки корневого дерева T необходимо для любой внутренней вершины дерева сначала выполнить оптимально укладку её поддеревьев в порядке невозрастания их ширины, когда первым укладывается поддерево с наибольшей шириной укладки, а последним — с наименьшей.*

Приведём один из возможных алгоритмов для решения задачи оптимальной по ширине укладки корневого дерева.

Алгоритм укладки
корневого дерева минимальной ширины

1. На начальном этапе припишем каждой вершине u дерева T метку $\mathrm{Width}(u)$, которая равна ширине укладки поддерева T^u. Для определения этих меток можно использовать поиск в глубину, начинающийся из корня дерева. Во время поиска в глубину при удалении вершины u из стека присваиваем ей метку $\mathrm{Width}(u)$ по правилу:

• если u — лист, то $\mathrm{Width}(u) = 0$;

• если u — внутренняя вершина дерева, а T_i^u $(i = 0, \ldots, k)$ — её поддеревья, для которых

$$\mathrm{Width}(T_0^u) \geqslant \mathrm{Width}(T_1^u) \geqslant \ldots \geqslant \mathrm{Width}(T_k^u),$$

то полагаем

$$\mathrm{Width}(u) = \max\big\{\mathrm{Width}(T_0^u), \mathrm{Width}(T_1^u) + 1, \ldots, \mathrm{Width}(T_k^u) + k, k + 1\big\}.$$

Метка корня дерева соответствует минимальной ширине укладки корневого дерева T.

2. Для оптимальной нумерации вершин корневого дерева T будем использовать поиск в глубину с приоритетами, начинающийся из корня дерева. Во время поиска в глубину с приоритетами:

• из текущей вершины u путь идёт в вершину v ($(u, v) \in A(T)$), которая имеет наибольшую метку $\mathrm{Width}(v)$, так как для минимизации ширины укладки сначала надо выполнить укладку поддерева с наибольшей шириной укладки;

• при удалении вершины u из стека полагаем $\mathrm{Number}_P(u) = k$ (первоначально берём $k = 1$) и увеличиваем счётчик k на 1.

Трудоёмкость приведённого выше алгоритма укладки корневого дерева минимальной ширины складывается из трудоёмкости вычисления меток $\mathrm{Width}(u)$ всех вершин u корневого дерева T (метки могут быть вычислены за время $\mathrm{O}(n \log n)$) и трудоёмкости поиска в глубину (составляет $\mathrm{O}(n)$). Следовательно, трудоёмкость приведённого алгоритма укладки корневого дерева минимальной ширины равна $\mathrm{O}(n \log n)$.

Пример 3.2. Рассмотрим задачу оптимизации вычисления функции, которая задана в виде корневого дерева, приведённого на рис. 3.14. Требуется определить порядок вычисления функции, который обеспечивает минимальное количество ячеек дополнительной памяти, необходимой для вычисления функции.

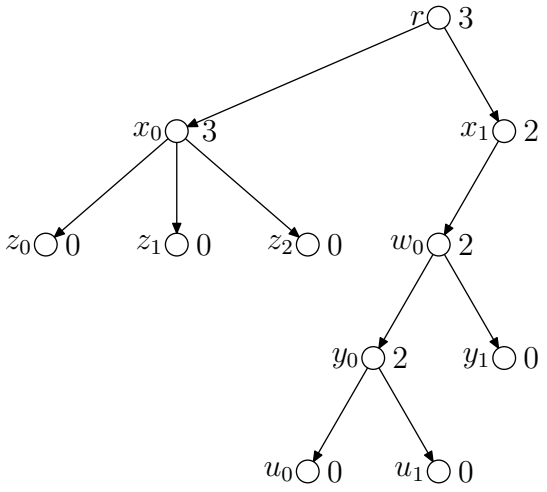


Рис. 3.14. Корневое дерево T^r

Одной из возможных математических моделей для данной задачи является оптимальная (по ширине) укладка корневого дерева. Выполним укладку минимальной ширины для корневого дерева T^r, приведённого на рис. 3.14. Сначала определим для каждой вершины v мини-

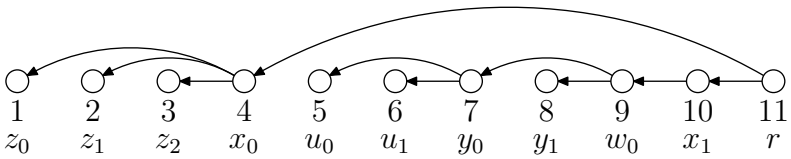


Рис. 3.15. Укладка P минимальной ширины 3 дерева T^r

мальную ширину укладки поддерева T^v (на рис. 3.14 данная величина указана возле каждой вершины).

На рис. 3.15 показана укладка P дерева T^r минимальной ширины 3. В соответствии с укладкой P в таблице для каждого такта времени показаны промежуточные данные, хранящиеся в памяти компьютера:

Такт	1	2	3	4	5	6	7	8	9	10
Состояние памяти	z_0	z_0 z_1	z_0 z_1 z_2	x_0	x_0 u_0	x_0 u_0 u_1	x_0 y_0	x_0 y_0 y_1	x_0 w_0	x_0 x_1
Количество используемых ячеек памяти	1	2	3	1	2	3	2	3	2	2

На последнем этапе (11-й такт) происходит вычисление главной функции r (к этому моменту времени все требуемые аргументы x_0 и x_1 главной функции уже вычислены и хранятся в памяти компьютера).

Наибольшее из чисел в последней строке таблицы соответствует минимальному количеству ячеек дополнительной памяти, требующемуся для вычисления главной функции, в порядке, который определён укладкой P.

Можно показать, что для корневого дерева T^r (см. рис. 3.14) укладка P (см. рис. 3.15) является укладкой минимальной ширины, но не укладкой минимальной длины (длина укладки P равна 21). На рис. 3.16 для дерева T^r приведена укладка P' минимальной длины 19 (ширина укладки P' равна 4).

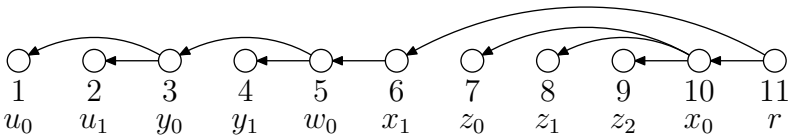


Рис. 3.16. Укладка P' минимальной длины 19 дерева T^r

3.3. НАИБОЛЬШЕЕ ПАРОСОЧЕТАНИЕ В ГРАФЕ

Напомним, что *паросочетанием* в графе называется множество рёбер, попарно не имеющих общих вершин. Паросочетание называется *совершенным*, если оно покрывает всё множество вершин графа. Задачи, связанные с нахождением паросочетаний, моделируют широкий спектр прикладных задач, возникающих при эффективной организации современного производства, статистической обработке данных, при планировании и оптимальной организации работы транспортных средств, при проектировании защищённых каналов передачи информации в широковещательных сетях.

В качестве примера рассмотрим следующую модельную задачу [6], сводящуюся к нахождению в некотором графе совершенного паросочетания. Фармацевтической компании требуется проверить n лекарств на n добровольцах. Предварительные исследования показали, что у некоторых испытуемых возможна аллергия на отдельные лекарства. Требуется так организовать эксперимент, чтобы каждый испытуемый получил в точности одно лекарство, на которое у него отсутствует аллергия, и чтобы каждое лекарство было дано в точности одному испытуемому. Для данной задачи можно построить следующую графовую модель — двудольный граф $G = (X, Y, E)$, в котором доля X соответствует n испытуемым, а доля Y — n лекарствам; две вершины $x \in X$ и $y \in Y$ смежны в графе G тогда и только тогда, когда у испытуемого x отсутствует аллергия на лекарство y. Нетрудно понять, что в графе G существует совершенное паросочетание тогда и только тогда, когда существует такое назначение «испытуемый – лекарство», при котором каждый испытуемый и каждое лекарство используется ровно один раз. В [6] приведён целый ряд других примеров, когда решение практически важной задачи сводится к нахождению совершенного паросочетания в некотором графе.

Всюду в этом разделе под термином «цепь» будем понимать простую цепь.

Пусть G — граф и M — паросочетание в этом графе. Вершины графа G, которые инцидентны рёбрам паросочетания M, называют *насыщенными*, а оставшиеся вершины — *ненасыщенными*.

Цепь графа G, в которую поочередно входят и не входят рёбра паросочетания M, называют *чередующейся цепью относительно паросочетания* M (или просто M-*чередующейся цепью*). Рёбра цепи, которые принадлежат паросочетанию, называют *тёмными* (или *сильными* рёбрами), а оставшиеся рёбра — *светлыми* (или *слабыми* рёбрами).

Если в графе существует M-чередующаяся цепь, соединяющая две различные ненасыщенные вершины графа, то можно построить паросочетание с бóльшим (на единицу), чем в M, числом рёбер, поэтому такую цепь называют *увеличивающей цепью относительно паросочетания* M (или просто M-*увеличивающей цепью*).

Пример 3.3. Рассмотрим граф G, изображённый на рис. 3.17, в котором построено паросочетание $M = \big\{\{1,6\},\{3,4\}\big\}$.

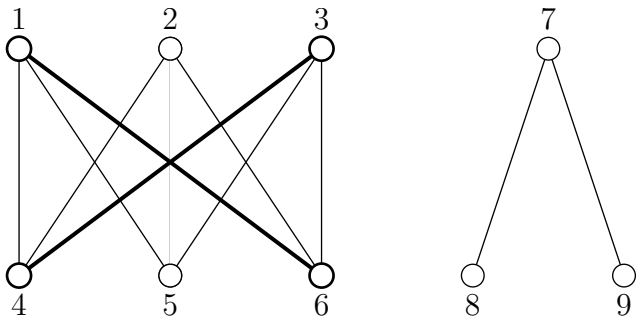


Рис. 3.17. Паросочетание M в графе G

На рис. 3.17 рёбра паросочетания M и насыщенные вершины графа G выделены тёмными линиями. В цепь, заданную последовательностью вершин

$$2-6-1-4-3-5,$$

поочередно входят и не входят рёбра паросочетания M, поэтому такая цепь является чередующейся цепью относительно паросочетания M (рёбра $\{6,1\}$ и $\{4,3\}$ — тёмные, а рёбра $\{2,6\},\{1,4\}$ и $\{3,5\}$ — светлые). Поскольку данная цепь соединяет ненасыщенные вершины 2 и 5, то она является увеличивающей цепью относительно паросочетания M.

Теорема 3.5. *Пусть M и M' — паросочетания в графе G. Тогда каждая компонента подграфа $G(M \oplus M')$ является либо циклом чётной длины, либо цепью.*

Доказательство. Рассмотрим подграф $H = G(M \oplus M')$, состоящий из рёбер графа G, которые входят в одно и только одно из паросочетаний M и M', и инцидентных им вершин. Поскольку каждая вершина подграфа H инцидентна не более чем двум рёбрам (одному из M, другому — из M'), то её степень не больше, чем 2. Если степень некоторой вершины подграфа H равна 2, то одно из инцидентных ей рёбер входит в M, другое — в M'. Следовательно, любая компонента подграфа H является либо циклом, содержащим одно и то же число рёбер из M и M' (т. е. циклом чётной длины), либо цепью. □

Теорема 3.6 (К. Берж, 1957)**.** *Паросочетание M в графе является наибольшим тогда и только тогда, когда в этом графе нет M-увеличивающих цепей.*

Доказательство. Пусть G — граф и M — паросочетание в этом графе. Покажем, что в графе G существует паросочетание M', для которого $|M'| > |M|$, тогда и только тогда, когда в этом графе есть M-увеличивающая цепь. Из этого факта будет следовать утверждение теоремы.

Если в графе G есть M-увеличивающая цепь, то, заменяя вдоль этой цепи тёмные рёбра на светлые и наоборот, получим новое паросочетание M', в котором число рёбер на единицу больше, чем в M. Для доказательства обратного утверждения рассмотрим паросочетание M' в графе G, для которого $|M'| > |M|$. Покажем, что тогда в графе G имеется увеличивающая цепь относительно M. Рассмотрим подграф $H = G(M \oplus M')$. Согласно теореме 3.5, каждая компонента связности графа H является либо циклом чётной длины, содержащим одно и то же число рёбер из M и M', либо цепью. Поскольку $|M'| > |M|$, то существует компонента, в которой рёбер из M' содержится больше, чем рёбер из M. Этой компонентой может быть только цепь, концевые рёбра которой принадлежат M'. Относительно паросочетания M эта цепь будет увеличивающей цепью, что и требовалось доказать. □

3.3.1. Наибольшее паросочетание в двудольном графе

Алгоритм Куна построения наибольшего паросочетания в двудольном графе непосредственно основывается на теореме Бержа. Предположим, что есть некоторое начальное паросочетание M (например, пустое). В алгоритме Куна просматриваем по очереди все вершины графа и проверяем наличие M-увеличивающей цепи поиском в глубину или ширину из каждой ненасыщенной вершины. Если M-увеличивающая цепь найдена, то вдоль цепи происходит перестройка текущего паросочетания M, а затем вновь происходит процесс построения увеличивающей цепи относительно нового паросочетания. Если на некотором шаге алгоритма Куна не удаётся найти M-увеличивающую цепь, то паросочетание M является наибольшим.

Эффективность описанной стратегии построения наибольшего паросочетания зависит от алгоритма построения увеличивающей цепи в графе. Рассмотрим три различные реализации алгоритма Куна.

Первая использует алгоритм нахождения максимального потока в сети, строящейся по исходному двудольному графу (для нахождения максимального потока в сети можно использовать, например, алгоритм Форда – Фалкерсона, который подробно описан в [2]). Для данной реализации необходимо, чтобы заранее было известно распределение вершин двудольного графа по долям.

Первая реализация алгоритма Куна

1. Достраиваем двудольный граф $G(X, Y, E)$ до сети:

   • вводим новую вершину s — источник, которую соединяем дугой (s, x) с каждой вершиной $x \in X$ графа G;

   • вводим новую вершину t — сток, с которой соединяем дугой (y, t) каждую вершину $y \in Y$ графа G;

   • рёбра двудольного графа заменяем дугами, направленными от X к Y;

   • пропускные способности всех дуг орграфа полагаем равными 1.

2. В построенной (s, t)-сети находим максимальный поток. Величина максимального потока равна мощности наибольшего паросоче-

тания, а рёбрам наибольшего паросочетания будут соответствовать те дуги сети, поток по которым равен 1 и которым соответствуют рёбра двудольного графа.

Трудоёмкость приведённой реализации алгоритма Куна будет равна $O(n'm)$, где $n' = \min\{|X|, |Y|\}$.

Пример 3.4. Найдём наибольшее паросочетание в двудольном графе $G(X, Y, E)$ с $X = \{1, 2, 3\}$ (первая доля) и $Y = \{4, 5, 6\}$ (вторая доля), изображённом на рис. 3.18.

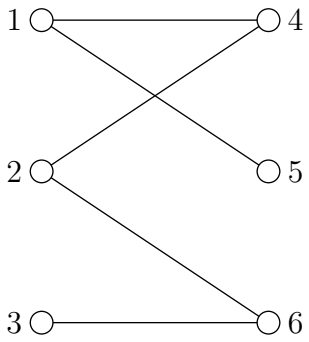


Рис. 3.18. Исходный двудольный граф $G(X, Y, E)$: $X = \{1, 2, 3\}$, $Y = \{4, 5, 6\}$

Достраиваем исходный граф $G(X, Y, E)$ до (s, t)-сети. Для этого вводим фиктивные вершины s и t, соединяем вершину s дугами $(s, 1)$, $(s, 2)$ и $(s, 3)$ со всеми вершинами доли X и соединяем дугами $(4, t)$, $(5, t)$ и $(6, t)$ все вершины доли Y с вершиной t.

Ориентируем все рёбра двудольного графа в направлении от X к Y (рис. 3.19). Пропускные способности всех дуг полагаем равными 1. Найдём максимальный поток в сети (дуги, по которым проходит максимальный поток, на рис. 3.19 выделены жирными линиями). Величина

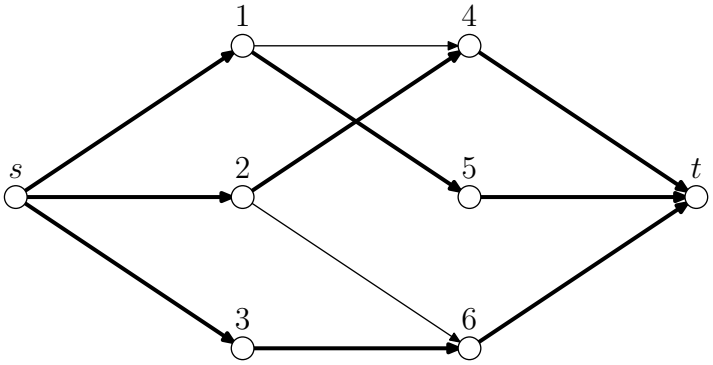


Рис. 3.19. Построенная (s, t)-сеть для исходного двудольного графа $G(X, Y, E)$

максимального потока равна 3 и, следовательно, мощность наибольшего паросочетания для двудольного графа, приведённого на рис. 3.18, равна 3.

Рёбра исходного двудольного графа, поток по которым равен 1 (см. рис. 3.19), соответствуют множеству рёбер наибольшего паросочетания (рис. 3.20)

$$\{1,5\},\ \{2,4\},\ \{3,6\}.$$

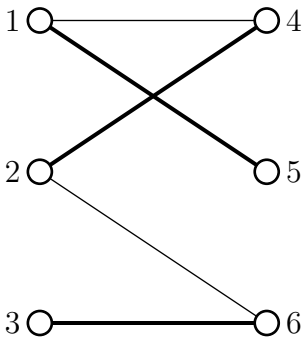


Рис. 3.20. Наибольшее паросочетание в двудольном графе $G(X,Y,E)$

Вторая реализация алгоритма Куна также предполагает, что известно распределение вершин двудольного графа по долям, и основывается на построении путей в специальном орграфе, который строится по текущему паросочетанию в двудольном графе [7].

Вторая реализация алгоритма Куна

0. В двудольном графе $G(X,Y,E)$ строим произвольное паросочетание M (в качестве такого паросочетания можно выбрать произвольное ребро графа либо взять любое паросочетание в графе).

1. По паросочетанию M строим ориентированный граф: заменяем рёбра графа, принадлежащие паросочетанию M, дугами, направленными от Y к X, а все оставшиеся рёбра графа — дугами, направленными от X к Y.

2. Используя поиск в ширину, ищем путь из множества ненасыщенных вершин первой доли (вершин, которые не покрыты паросочетанием) в множество ненасыщенных вершин второй доли (в дальнейшем такой путь будем называть *увеличивающим путём*). При этом получается, что при движении из вершины v первой доли мы идём во все

смежные с v не посещённые ранее вершины второй доли, а при движении из вершины второй доли идём по ребру текущего паросочетания в не посещённую ранее вершину первой доли. Если увеличивающего пути не существует, то, по теореме Бержа, текущее паросочетание M является наибольшим и алгоритм завершает свою работу.

3. Если увеличивающий путь найден (путь завершается в ненасыщенной вершине второй доли и имеет нечётную длину), то переориентируем дуги этого пути (заменим ориентации дуг противоположными) и вернёмся к шагу 2 алгоритма. Переориентация дуг вдоль увеличивающего пути модифицирует текущее паросочетание следующим образом: рёбра увеличивающего пути, которые не принадлежали паросочетанию M, будут добавлены в него, а рёбра увеличивающего пути, которые принадлежали паросочетанию M, будут удалены (в результате в паросочетании станет на одно ребро больше).

Поскольку поиск увеличивающего пути осуществляется за $\mathrm{O}(m)$, то трудоёмкость приведённой реализации алгоритма Куна равна

$$\mathrm{O}\big(m \cdot \min\big\{|X|, |Y|\big\}\big)$$

(целесообразно запускать поиск увеличивающего пути из меньшей по мощности доли двудольного графа).

Пример 3.5. Найдём наибольшее паросочетание в двудольном графе $G(X, Y, E)$ с $X = \{1, 2, 3, 7\}$ и $Y = \{4, 5, 6, 8, 9\}$, изображённом на рис. 3.21.

Пусть $M = \big\{\{1, 6\}, \{3, 4\}\big\}$ — начальное паросочетание в исходном двудольном графе (на рис. 3.21 рёбра паросочетания M и насыщенные вершины выделены жирными линиями).

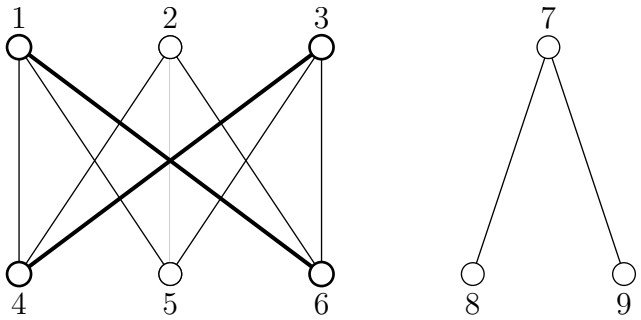


Рис. 3.21. Начальное паросочетание M в двудольном графе $G(X, Y, E)$

Построим по паросочетанию M ориентированный граф, в котором найдём некоторый увеличивающий путь из множества ненасыщенных вершин первой доли в множество ненасыщенных вершин второй доли. Увеличивающий путь, заданный последовательностью вершин

$$2 \to 4 \to 3 \to 5,$$

соединяет ненасыщенную вершину 2 первой доли с ненасыщенной вершиной 5 второй доли (на рис. 3.22 насыщенные вершины выделены жирными линиями, дуги увеличивающего пути выделены пунктирными линиями).

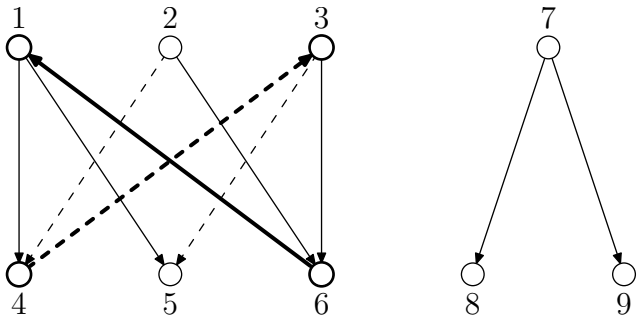


Рис. 3.22. Увеличивающий путь между ненасыщенными вершинами 2 и 5

Изменяем ориентации дуг найденного увеличивающего пути на противоположные, модифицируя текущее паросочетание M.

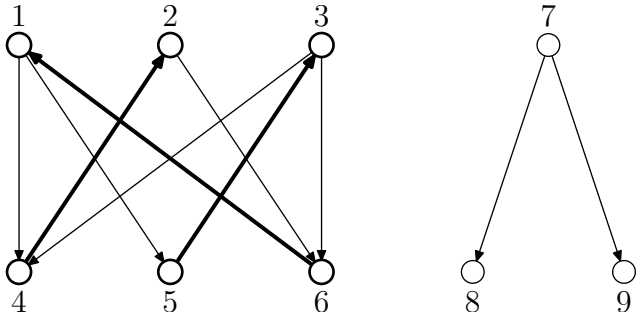


Рис. 3.23. Новое паросочетание M

В полученном орграфе (рис. 3.23) найдём некоторый увеличивающий путь из множества ненасыщенных вершин первой доли в множество ненасыщенных вершин второй доли. Увеличивающий путь, заданный последовательностью вершин

$$7 \to 8,$$

соединяет ненасыщенную вершину 7 первой доли с ненасыщенной вершиной 8 второй доли (на рис. 3.24 дуги увеличивающего пути выделены пунктирными линиями).

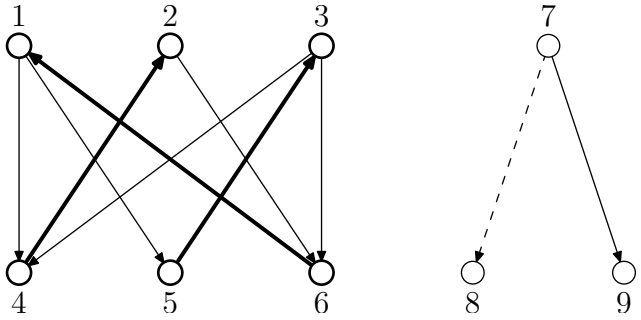


Рис. 3.24. Увеличивающий путь между ненасыщенными вершинами 7 и 8

Изменяем ориентации дуг найденного увеличивающего пути на противоположные (рис. 3.25), и так как все вершины первой доли насыщены, то наибольшее паросочетание для графа, изображённого на рис. 3.21, построено.

Мощность наибольшего паросочетания равна сумме степеней полузахода всех вершин первой доли двудольного графа, т. е. 4 (рис. 3.25).

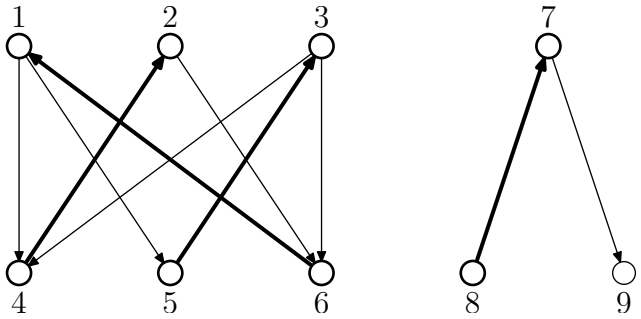


Рис. 3.25. Переориентация дуг увеличивающего пути $7 \to 8$

Наибольшему паросочетанию соответствуют дуги орграфа, входящие в вершины первой доли, т. е. наибольшее паросочетание имеет вид

$$M = \big\{\{1,6\},\{2,4\},\{3,5\},\{7,8\}\big\}.$$

Рассмотрим третью реализацию алгоритма Куна, которая работает в предположении, что граф двудольный, но нет предварительной

информации о разбиении множества вершин на доли. В данной реализации во время *модифицированного поиска в ширину* (или глубину), который по текущему паросочетанию M строит M-увеличивающую цепь, все вновь посещаемые вершины делятся на *чётные* и *нечётные*.

Вершину x назовём *чётной* или *нечётной* в зависимости от того, чётно или нечётно расстояние между x и стартовой вершиной поиска (расстояние между вершинами u и v равно длине наименьшей по числу рёбер (u, v)-цепи).

В процессе построения увеличивающей цепи для чётных вершин исследуются все инцидентные им светлые рёбра, а для каждой из нечётных — единственное инцидентное ей тёмное ребро.

Алгоритм
модифицированного поиска в ширину
построения M-увеличивающей цепи

0. Очередь Q — пустая. Все вершины графа — «новые» (вершина перестаёт быть «новой» при занесении её в очередь Q).

1. Выбираем любую новую ненасыщенную вершину u, заносим её в очередь Q и полагаем вершину u чётной.

2. Пока очередь Q не пуста, выполняем следующие шаги:

2.1. Удаляем вершину v из начала очереди.

2.2. Если вершина v — нечётная, то:

- если v — ненасыщенная вершина, то M-увеличивающая цепь найдена (может быть построено паросочетание большей мощности) и алгоритм модифицированного поиска в ширину завершает свою работу;
- если v — насыщенная вершина, то добавляем вершину w, смежную в паросочетании с вершиной v, в очередь Q (vw — ребро, принадлежащее паросочетанию), полагаем вершину w чётной, предком вершины w считаем вершину v.

2.3. Если вершина v — чётная, то заносим в очередь Q все смежные с v новые вершины, полагая их нечётными. Предком для этих вершин считаем вершину v.

3. Если очередь Q пуста (M-увеличивающая цепь не найдена) и в графе есть новые ненасыщенные вершины, то возвращаемся к шагу 1

алгоритма, чтобы попытаться построить M-увеличивающую цепь из некоторой новой ненасыщенной вершины.

4. Если очередь Q пуста (M-увеличивающая цепь не найдена) и в графе нет новых ненасыщенных вершин, то M-увеличивающей цепи не существует и алгоритм модифицированного поиска в ширину завершает свою работу.

Приведём теперь третью реализацию алгоритма Куна.

Третья реализация алгоритма Куна

0. Предположим, что у нас есть некоторое текущее паросочетание M, относительно которого все вершины двудольного графа делятся на насыщенные (покрытые паросочетанием) и ненасыщенные.

1. Для построения M-увеличивающей цепи выполним модифицированный поиск в ширину.

2. Если в результате модифицированного поиска в ширину M-увеличивающая цепь найдена, то:

- чередуем рёбра паросочетания M вдоль M-увеличивающей цепи (см. теорему 3.6 Бержа), получая при этом новое паросочетание M' (мощности на единицу большей, чем $|M|$);
- полагаем все вершины графа новыми;
- в качестве текущего паросочетания M берём паросочетание M' (т. е. $M = M'$);
- возвращаемся к шагу 1 алгоритма.

3. Если в результате модифицированного поиска в ширину M-увеличивающая цепь не найдена, то текущее паросочетание M является наибольшим и алгоритм завершает свою работу.

Замечание 3.1. Если для некоторого паросочетания M двудольного графа дерево поиска, построенное из ненасыщенной вершины u, содержит хотя бы одну вершину увеличивающего пути, то оно содержит и увеличивающий путь.

Из замечания 3.1 следует, что, если дерево достижимости, построенное из ненасыщенной вершины u для некоторого паросочетания M, не содержит других ненасыщенных вершин, кроме корневой вершины u, то ни одна вершина этого дерева не принадлежит никакому

увеличивающему пути для данного паросочетания M. Следовательно, приступая к построению следующего дерева достижимости для данного паросочетания M из некоторой новой ненасыщенной вершины u', все посещённые вершины первого дерева можно не рассматривать (т. е. метки посещения вершин не нужно обнулять).

Алгоритм *модифицированного поиска в ширину* строит увеличивающую цепь за время $O(m)$. Количество запусков алгоритма поиска в ширину не превосходит мощности наибольшего паросочетания, т. е. $n/2$, так как каждый раз после построения увеличивающей цепи мощность текущего паросочетания увеличивается на 1. Таким образом, трудоёмкость третьей реализации алгоритма Куна есть $O(nm)$.

Пример 3.6. Найдём наибольшее паросочетание для двудольного графа, изображённого на рис. 3.26.

В качестве текущего паросочетания возьмём паросочетание

$$M = \big\{\{1,6\},\{2,4\}\big\}.$$

Все вершины графа полагаем новыми. Выбираем новую ненасыщенную вершину 3 в качестве стартовой вершины для модифицированного поиска в ширину, полагаем очередь Q пустой. Заносим вершину 3 в очередь: $Q = [3]$ (как только вершина заносится в очередь, она перестаёт быть новой) и полагаем эту вершину чётной.

Строим дерево поиска в ширину с корнем в вершине 3.

1. Удаляем из очереди $Q = [3]$ вершину $v = 3$. Вершина 3 является чётной, поэтому заносим в очередь все новые смежные с вершиной 3 вершины: $Q = [4, 6]$. Полагаем вершины 4 и 6 нечётными и указываем, что предком для вершин 4 и 6 является вершина 3.

2. Удаляем из очереди $Q = [4, 6]$ вершину $v = 4$. Вершина 4 является нечётной и принадлежит ребру паросочетания, поэтому заносим в очередь Q смежную с ней в паросочетании M вершину 2: $Q = [6, 2]$, полагаем вершину 2 чётной и указываем, что предком для вершины 2 является вершина 4.

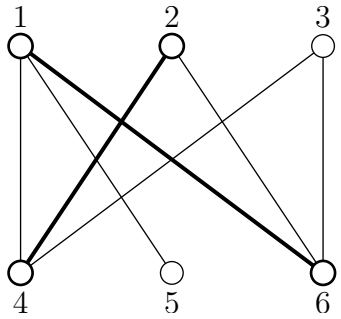


Рис. 3.26.
Двудольный граф

3. Удаляем из очереди $Q = [6, 2]$ вершину $v = 6$. Вершина 6 является нечётной и принадлежит ребру паросочетания, поэтому заносим в очередь Q смежную с ней в паросочетании вершину 1: $Q = [2, 1]$, полагаем вершину 1 чётной, и указываем, что предком для вершины 1 является вершина 6.

4. Удаляем из очереди $Q = [2, 1]$ вершину $v = 2$. Вершина 2 является чётной, поэтому надо занести в очередь Q все новые вершины, смежные с вершиной 2, но таких вершин нет. Поэтому новые элементы в очередь $Q = [1]$ добавлены не будут.

5. Удаляем из очереди $Q = [1]$ вершину $v = 1$. Вершина 1 является чётной, поэтому надо занести в очередь все новые вершины, смежные с вершиной 1: $Q = [5]$. Полагаем вершину 5 нечётной и указываем, что предком для вершины 5 является вершина 1.

6. Удаляем из очереди $Q = [5]$ вершину $v = 5$. Вершина 5 является нечётной и ненасыщенной. Следовательно, M-увеличивающая цепь построена. Восстановить такую цепь можно, используя массив предков, т. е. корневое дерево поиска в ширину (рис. 3.27):

$$3 - 6 - 1 - 5.$$

Модифицируем текущее паросочетание

$$M = \big\{\{1, 6\}, \{2, 4\}\big\}$$

вдоль M-увеличивающей цепи

$$3 - 6 - 1 - 5,$$

удаляя из текущего паросочетания M рёбра, которые принадлежат M-увеличивающей цепи, и добавляя к паросочетанию M рёбра M-увеличивающей цепи, которые не принадлежат паросочетанию M. После модификации получим новое паросочетание

$$M = \big\{\{1, 5\}, \{2, 4\}, \{3, 6\}\big\},$$

мощность которого на единицу больше, чем мощность предыдущего паросочетания (на рис. 3.28 выделены рёбра паросочетания).

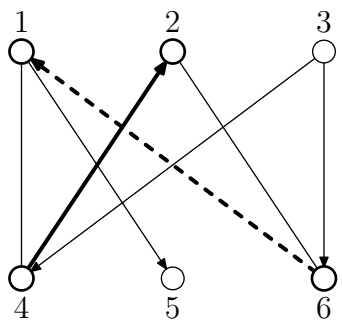


Рис. 3.27. Корневое дерево поиска в ширину

Поскольку в двудольном графе больше нет ненасыщенных вершин, то текущее паросочетание

$$M = \big\{\{1,5\},\{2,4\},\{3,6\}\big\}$$

является наибольшим паросочетанием для двудольного графа, изображённого на рис. 3.26.

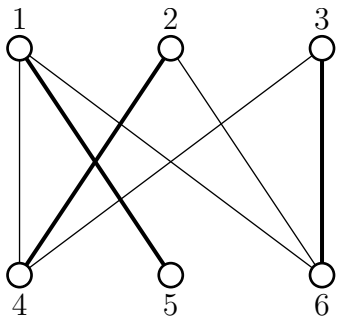


Рис. 3.28. Наибольшее паросочетание в двудольном графе

Заметим, что деление вершин на чётные и нечётные окажется полезным и в дальнейшем, когда мы будем строить наибольшее паросочетание в произвольном графе и нам потребуется находить «плохие» чередующиеся циклы нечётной длины.

3.3.2. Наибольшее паросочетание в произвольном графе

Для произвольного графа поиск M-увеличивающей цепи является более сложной задачей. Трудность может возникнуть в ситуации, когда в графе при построении увеличивающей цепи по тёмному ребру vw приходят в вершину w и попадают в так называемый *плохой цикл* C нечётной длины (рис. 3.29): чередующаяся цепь нечётной длины из вершины w в саму себя, начинающаяся и заканчивающаяся в вершине w светлыми рёбрами (такой цикл называют *бутоном*). В плохом цикле C существует ровно одна вершина (*база* цикла), которая не покрыта рёбрами паросочетания, принадлежащими циклу (см. рис. 3.29).

При построении увеличивающей цепи (используя, например, модифицированный поиск в ширину) при попадании на базу дальнейшее движение по бутону не однозначно и при произвольном выборе такого направления движения оно может пойти не в том направлении. На рис. 3.29 стрелками показаны два возможных направления движения по плохому циклу. Одно из выбранных направлений, при котором из вершины w выполнено движение в вершину x, пройдёт все вершины цикла и не сможет выйти из него. Второе выбранное направление, при котором из вершины w выполнено движение в вершину y, приведёт к успешному выходу из цикла через вершину u.

Может показаться, что подобная проблема существует и в двудольном графе. Покажем, что это не так. В двудольном графе (по критерию двудольности) нет циклов нечётных длин. Неоднозначность выбора на-

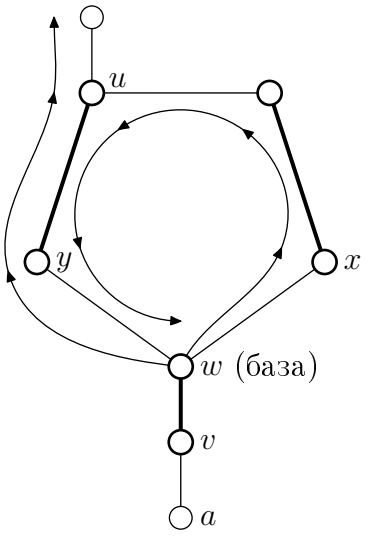


Рис. 3.29. Цикл нечётной длины

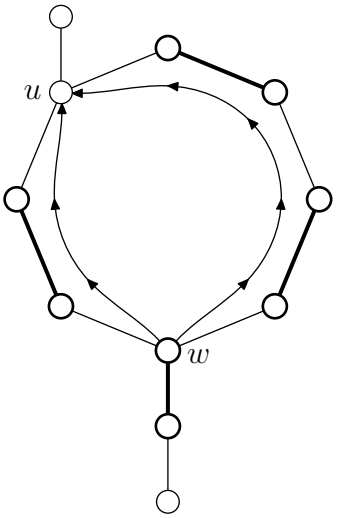


Рис. 3.30. Цикл чётной длины

правления движения в двудольном графе возможна только после входа в цикл чётной длины (рис. 3.30), в котором светлых рёбер на два больше, чем тёмных, причём вход в цикл осуществляется через базу (в таком цикле всегда точно две базы: на рис. 3.30 это вершины w и u). Чередующиеся цепи по рёбрам такого чётного цикла идут от вершины w, не перекрывая друг друга (в отличие от плохого цикла нечётной длины, изображённого на рис. 3.29), и ко второй базе u цикла подходят по светлым рёбрам (на рис. 3.30 возможные направления движения по циклу показаны стрелками). Поэтому если одна из чередующихся цепей «пришла» в эту вершину раньше (т. е. вершина u станет заблокированной) и смогла (или не смогла) от вершины u продолжить себя, то вторую цепь можно исключить из рассмотрения, так как начиная с вершины u эта цепь будет совпадать с первой цепью.

Теперь рассмотрим граф, представленный на рис. 3.31, который не является двудольным. В качестве текущего паросочетания выберем следующее подмножество рёбер:

$$M = \{cb, df, hg, k\ell\}.$$

Предположим, что построение увеличивающей цепи начато из ненасыщенной вершины a. В плохой цикл мы попадаем по тёмному ребру cb. Вершина b цикла покрыта только светлыми рёбрами цикла, поэтому

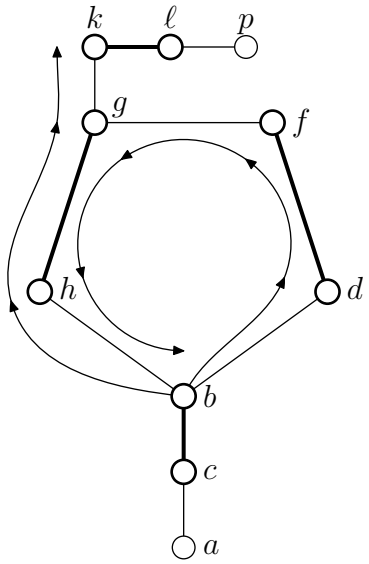


Рис. 3.31. Исходный граф G

является базой бутона. Чередующуюся цепь, ведущую из ненасыщенной вершины a к базе бутона b, называют *стеблем* и его длина всегда чётна (стебель вместе с бутоном называют *цветком*). После того как мы оказались в вершине цикла b, дальнейшее движение по циклу становится неоднозначным: можно выбрать направление движения либо в вершину d, либо в вершину h. Покажем важность выбора направления движения по циклу (см. рис. 3.31).

1. Если при построении чередующейся цепи из вершины b перейти в вершину d, то дальнейшее движение вдоль чередующейся цепи приведёт в тупик:

$$a - c - b - d - f - g - h,$$

так как мы вернёмся в стартовую вершину a и будет сделан ошибочный вывод о том, что увеличивающей цепи из ненасыщенной вершины a не существует.

2. Если при построении чередующейся цепи, находясь в базе b, перейти к вершине h, то будет построена увеличивающая цепь

$$a - c - b - h - g - k - \ell - p,$$

соединяющая ненасыщенную вершину a с ненасыщенной вершиной p.

Рассмотрим алгоритм, который разработан Дж. Эдмондсом (1965), корректно работает в случае существования в графе плохих циклов и

может быть реализован за время $O(n^3)$. В алгоритме Эдмондса при обнаружении плохого цикла нечётной длины (бутона) все вершины этого цикла сжимают в одну вершину $\overline{b}$, а рёбра, инцидентные хотя бы одной вершине цикла, будут теперь инцидентны вершине $\overline{b}$ (такие циклы легко отслеживаются, например, если при поиске увеличивающейся цепи используется поиск в ширину, то к плохому циклу приводит ребро vw с чётными вершинами v и w, принадлежащими дереву поиска).

До выполнения процесса сжатия базу бутона нужно сделать ненасыщенной вершиной: светлые рёбра стебля заменить на тёмные, а тёмные — на светлые (так как длина стебля чётная, то, меняя на стебле цвет рёбер, мы не изменим мощность текущего паросочетания, но добьемся того, что база бутона b станет ненасыщенной вершиной). После такого преобразования гарантируется, что в графе нет тёмных рёбер, инцидентных сжатой вершине $\overline{b}$.

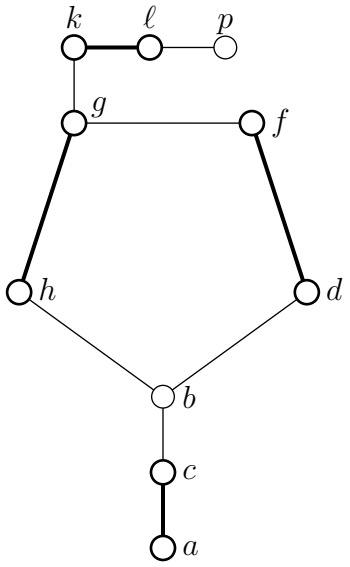


Рис. 3.32. Модификикация текущего паросочетания

Проиллюстрируем описанные выше действия для графа G, изображённого на рис. 3.31 (текущее паросочетание $M = \{cb, df, hg, k\ell\}$ на рис. 3.31 выделено тёмными линиями). Сначала сделаем базу бутона (вершину b) ненасыщенной вершиной (рис. 3.32). Затем выполним сжатие бутона, получив в результате из исходного графа G модифицированный граф G' (рис. 3.33).

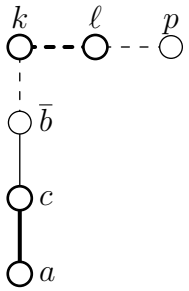


Рис. 3.33. Модифицированный граф G' и увеличивающая цепь P'

В графе G' отсутствуют плохие циклы, поэтому используя, например, поиск в ширину (глубину), строим в графе G' увеличивающую цепь.

Пусть P' — увеличивающая цепь графа G', тогда возможны следующие варианты.

1. Если увеличивающая цепь P' не проходит через вершину, соответствующую сжатому бутону, то цепь P' является увеличивающей цепью и для исходного графа G.

2. Если в увеличивающей цепи P' одним из концов является вершина, соответствующая сжатому бутону, то двигаясь вдоль чередующейся цепи P', подходя к такому бутону, его *раскрывают*.

Поясним процедуру *раскрытия бутона* на нашем примере. Если в графе G' взять ненасыщенную вершину p (или ненасыщенную вершину $\overline{b}$), то будет построена увеличивающая цепь P' нечётной длины, заданная последовательностью вершин

$$p - \ell - k - \overline{b}.$$

На рис. 3.33 рёбра увеличивающей цепи P' изображены пунктирной линией.

Увеличивающей цепи P' в исходном графе G соответствует увеличивающая цепь P, которая строится по следующему правилу. Пусть вершина k предшествует сжатой вершине $\overline{b}$ в увеличивающей цепи P'. Тогда в бутоне существует вершина, соединённая с вершиной k светлым ребром. В нашем примере такой вершиной является вершина g. Раскрываем бутон, движемся сначала от вершины k к вершине g, а далее — вдоль цикла к базе бутона b, но при этом движение от вершины g начинаем в направлении тёмного рёбра, инцидентного вершине g

(в нашем примере из вершины g идём в вершину h, так как ребро gh — тёмное). Длина раскрытой части бутона (это чередование рёбер: тёмное, светлое, тёмное, светлое и т. д.) всегда чётна, поэтому полученная из цепи P' цепь P имеет нечётную длину (её длина складывается из нечётной длины цепи P' и чётной длины раскрытого бутона). Более того, по построению цепь P является чередующейся и соединяет две ненасыщенные вершины (одна из которых — база бутона). Таким образом, цепь P является увеличивающей цепью для графа G.

3. Ситуация, когда сжатая вершина $\overline{b}$ является внутренней вершиной увеличивающей цепи P', не возможна. Если бы такая ситуация была возможна, то в этом случае в чередующейся цепи P' существовали бы два ребра, инцидентные вершине $\overline{b}$, причём одно из них обязательно было бы тёмным. Однако это не возможно, так как по построению в графе G' все рёбра, инцидентные сжатой вершине $\overline{b}$, являются светлыми.

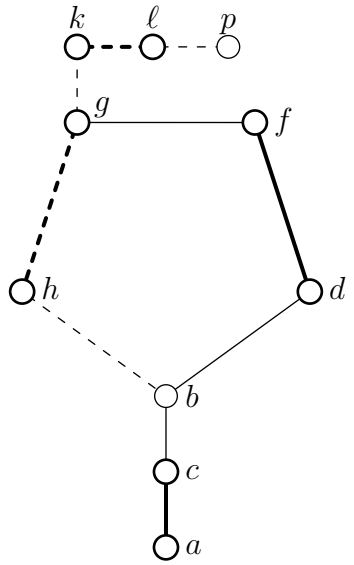


Рис. 3.34. Увеличивающая цепь P после раскрытия бутона

На рис. 3.34 пунктирной линией показана увеличивающая цепь P, полученная из цепи P' после раскрытия бутона.

Модифицируя вдоль увеличивающей цепи

$$P = p - \ell - k - g - h - b$$

текущее паросочетание

$$M = \{\ell k, gh, df, ac\},$$

получим новое паросочетание

$$M' = \{ac, df, bh, gk, \ell p\}$$

на единицу большей мощности, чем паросочетание M (рис. 3.35).

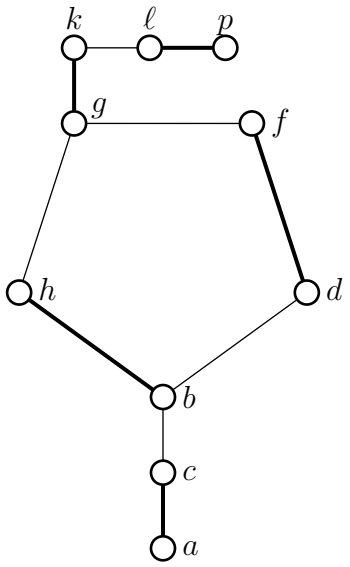


Рис. 3.35. Наибольшее паросочетание в графе

Для графа, приведённого на рис. 3.35, все вершины являются насыщенными, а следовательно, паросочетание $M' = \{ac, df, bh, gk, \ell p\}$ — наибольшее паросочетание для данного графа.

Приведём псевдокод алгоритма Эдмондса с трудоёмкостью $O(n^3)$. В процессе работы алгоритма для каждой вершины v будет поддерживаться следующая информация:

• `Пара` v — вершина, смежная с данной в текущем паросочетании. Если `пара` $v = \varnothing$, то вершина не покрыта ни одним ребром текущего паросочетания.

• `Родитель` v — чётная вершина, являющаяся предком v в корневом дереве поиска в ширину. (При переходе по ребру паросочетания `родитель` v не обновляется.) Если `родитель` $v = \varnothing$, то цепь нечётной длины до вершины v не найдена.

• `База` v — вершина, являющаяся базой самого большого сжатого бутона, которому принадлежит v. В начале построения увеличивающей цепи каждая вершина считается бутоном, содержащим только эту вершину.

• v `посещена` — булева переменная, являющаяся индикатором посещения вершины v при поиске наименьшего общего предка.

При программной реализации вся вышеперечисленная информация может храниться как в монолитной структуре для каждой вершины, так и в отдельных массивах для каждой или некоторых частей этой информации. (Массив может использоваться, если вершины задаются своими номерами, т. е. $v \in \{1, \ldots, n\}$ для всех $v \in V(G)$.) Совокупную информацию для всех вершин графа вместе со структурой самого графа будем называть *псевдосжатым графом*, поскольку явным образом сжатие бутонов в графе не происходит, тем не менее для каждой вершины можно быстро определить, частью какой вершины она оказалась бы при явном сжатии бутонов.

Помимо этого, будет использоваться очередь Q, которая формируется при построении увеличивающей цепи. В процессе работы алгоритма в очередь заносятся только чётные вершины.

Псевдокод алгоритма Эдмондса

```
// В псевдосжатом графе G находит
// наименьшего общего предка вершин u и v.
Функция Наименьший общий предок: (псевдосжатый граф G,
                                   вершина u, вершина v) → вершина
    Для всех w из V(G)
        w посещена ← ложь.
    Пока пара базы u ≠ ∅
        u ← база u.
        u посещена ← истина.
        u ← родитель пары u.
    Пока база v не посещена
        v ← родитель пары базы v.
    Вернуть v.
```

```
// В псевдосжатом графе G обновляет информацию
// для вершин нового бутона (с базой b), лежащих
// между вершинами v и b при обходе цикла из v
// в направлении пары v.
Функция Сжать бутон: (псевдосжатый граф G,
                      очередь Q, вершина v, вершина b) →
                                  → (граф, очередь)
    Пока база v ≠ b
        База v ← b.
        v ← пара v.
        Родитель родителя v ← v.
        База v ← b.
        Q ← Q + v.
        v ← родитель v.
    Вернуть (G, Q).

// В графе G осуществляет поиск увеличивающей цепи
// из свободной вершины u.
// Возвращает пару (G, v), где G задаёт тот же граф
// с обновлёнными родителями вершин, участвующих
// в увеличивающей цепи, а вершина v — конец этой
// цепи (отличный от u), или v = ∅, если цепь
// не существует.
Функция Найти увеличивающую цепь:
                      (псевдосжатый граф G, вершина u) →
                          → (псевдосжатый граф, вершина)
    Для всех v из V(G)
        Родитель v ← ∅.
    Для всех v из V(G)
        База v ← v.
    Q ← новая очередь.
    Q ← Q + u.
    Пока Q не пуста
        v ← начало Q.
```

```
        Q ← Q − v.
        Для всех w из N(v)
            Если база v = база w или пара v = w
                Начать следующую итерацию цикла.

            // Если найден цикл нечётной длины.
            Если w = u или
                (пара w ≠ ∅ и родитель пары w ≠ ∅)

                Родитель v ← w.
                Родитель w ← v.
                p ← Наименьший общий предок(G, v, w).
                (G, Q) ← Сжать бутон(G, Q, v, p).
                (G, Q) ← Сжать бутон(G, Q, w, p).
                Начать следующую итерацию цикла.

            Если родитель w = ∅
                Родитель w ← v.
                Если пара w = ∅
                    Вернуть (G, w).
                Q ← Q + пара w.
    Вернуть (G, ∅).

Функция Найти наибольшее паросочетание: граф G
                                    → множество рёбер
    Для всех u из V(G)
        Пара u ← ∅.
    Для всех u из V(G)
        Если пара u ≠ ∅
            Начать следующую итерацию цикла.
        (G, v) ← НайтиУвеличивающуюЦепь(G, u).
        Если v = ∅ // Цепь не найдена.
            Начать следующую итерацию цикла.
        Пока v ≠ u
```

$w \leftarrow$ родитель v.
Пара $v \leftarrow w$.
Пара $w \leftrightarrows v$.
$M \leftarrow \varnothing$.
Для всех u **из** $V(G)$
Если пара $u \neq \varnothing$
$M \leftarrow M \cup \{\{u,$ пара $u\}\}$.
Вернуть M.

Отметим, что модификация функции нахождения наименьшего общего предка в корневом дереве поиска в ширину, при которой движение к корню осуществляется из обеих вершин одновременно до появления первой дважды посещённой вершины, позволяет понизить время работы до $O(nm)$. Доказательство справедливости обеих оценок трудоёмкости оставим читателю в качестве упражнения.

3.3.3. Специальные паросочетания в двудольном графе

Наибольшее паросочетание максимального веса в двудольном графе. Задан взвешенный двудольный граф (G, w), $V(G) = X \cup Y$, где X и Y — множества вершин первой и второй долей графа G соответственно.

Задача нахождения в двудольном графе (G, w) наибольшего паросочетания с максимальным суммарным весом рёбер называется *задачей о наибольшем паросочетании максимального веса в двудольном графе*.

Пусть $P = \{x_1y_1, x_2y_2, \ldots, x_\ell y_\ell\}$, $x_i \in X$, $y_i \in Y$ — множество рёбер некоторого наибольшего паросочетания в двудольном графе, которое может быть построено, например, любой из реализаций алгоритма Куна, рассмотренных в подразделе 3.3.1. Приведём алгоритм построения наибольшего паросочетания максимального веса в двудольном графе.

Алгоритм
построения наибольшего паросочетания
максимального веса в двудольном графе

1. Для исходного двудольного взвешенного графа (G, w) и текущего наибольшего паросочетания M строим взвешенный орграф (G', w') по следующим правилам:

• каждому ребру $x_k y_k$ двудольного графа (G, w), которое принадлежит паросочетанию M, в орграфе (G', w') будет соответствовать дуга (x_k, y_k) веса $w'(x_k, y_k) = w(x_k y_k)$;

• каждому ребру $x_k y_k$ двудольного графа (G, w), которое не принадлежит паросочетанию M, в орграфе (G', w') будет соответствовать дуга (y_k, x_k) веса $w'(y_k, x_k) = -w(x_k y_k)$.

2. Находим в орграфе (G', w') контур C отрицательного веса (для этого можно использовать, например, алгоритм Форда–Беллмана [2] с трудоёмкостью $O(nm)$).

3. Если в орграфе (G', w') существует контур отрицательного веса, то это означает, что имеющееся паросочетание M не является наибольшим паросочетанием максимального веса. Действительно, если получен контур отрицательного веса (так как граф двудольный, контур всегда имеет чётную длину), то в таком случае половина дуг контура будет соответствовать рёбрам паросочетания M, а половина — нет. Нетрудно заметить, что одно множество может быть заменено на другое и при этом получится новое наибольшее паросочетание большего веса (часть рёбер паросочетания M заменилась другими рёбрами, суммарный вес которых больше).

Преобразуем паросочетание M по следующему правилу:

• если дуга (y_k, x_k) контура C имеет отрицательный вес, то добавляем ребро $x_k y_k$ к паросочетанию M;

• если дуга (x_k, y_k) контура C имеет положительный вес, то удаляем ребро $x_k y_k$ из паросочетания M.

Если вес контура C есть отрицательное число z, то вес нового паросочетания M будет больше веса предыдущего паросочетания на величину $|z|$. Возвращаемся к шагу 1 алгоритма.

4. Если в орграфе G' не существует контура отрицательного веса, то текущее паросочетание M является наибольшим паросочетанием максимального веса и алгоритм завершает свою работу.

Пример 3.7. Найдём наибольшее паросочетание максимального веса в двудольном взвешенном графе (G, w) с $V(G) = X \cup Y$, где $X = \{1, 2, 3\}$, $Y = \{4, 5, 6\}$, изображённом на рис. 3.36 (веса рёбер обозначены числами возле них).

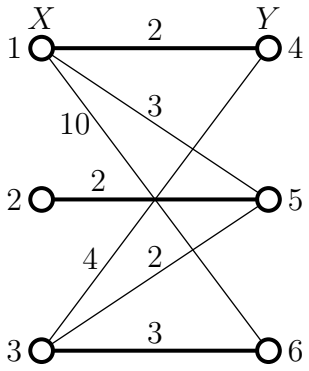


Рис. 3.36. Наибольшее паросочетание M веса 7 в графе (G, w)

Для двудольного взвешенного графа (G, w) в качестве начального наибольшего паросочетания M берём следующий набор рёбер:

$$M = \big\{\{1,4\},\{2,5\},\{3,6\}\big\}$$

(на рис. 3.36 рёбра паросочетания M выделены жирными линиями). Вес паросочетания M равен 7.

По графу (G, w) и текущему наибольшему паросочетанию M строим взвешенный орграф (G', w') (рис. 3.37). В орграфе (G', w') существует контур

$$C = 3 \to 6 \to 1 \to 4 \to 3$$

отрицательного веса (на рис. 3.37 дуги контура отрицательного веса выделены пунктирными линиями). Вес контура C равен -9.

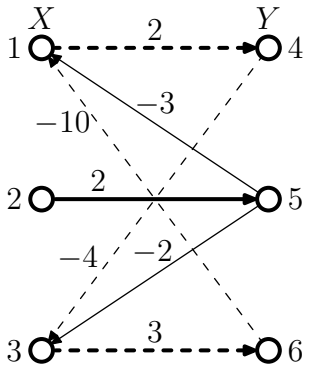


Рис. 3.37. Контур отрицательного веса в орграфе (G', w')

Для контура C дуги $(1,4)$ и $(3,6)$ имеют положительный вес, поэтому исключаем рёбра $\{1,4\}$ и $\{3,6\}$ из паросочетания M. Для контура C

дуги $(6,1)$ и $(4,3)$ имеют отрицательный вес, поэтому добавляем рёбра $\{1,6\}$ и $\{3,4\}$ в паросочетание M. Паросочетание M после преобразования имеет вид

$$M = \big\{\{1,6\},\{2,5\},\{3,4\}\big\},$$

а его вес равен 16 (на рис. 3.38 рёбра паросочетания M выделены жирными линиями).

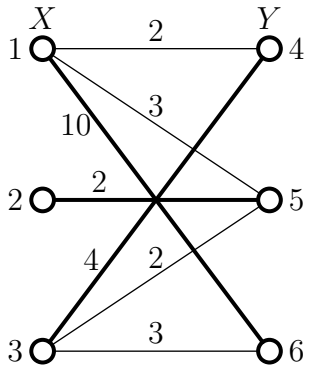


Рис. 3.38. Наибольшее паросочетание M веса 16 в графе (G,w)

По двудольному взвешенному графу (G,w) и текущему наибольшему паросочетанию $M = \big\{\{1,6\},\{2,5\},\{3,4\}\big\}$ строим новый взвешенный орграф (G',w') (рис. 3.39). В орграфе (G',w') отсутствуют контуры отрицательного веса. Следовательно, паросочетание

$$M = \big\{\{1,6\},\{2,5\},\{3,4\}\big\}$$

веса 16 является наибольшим паросочетанием максимального веса.

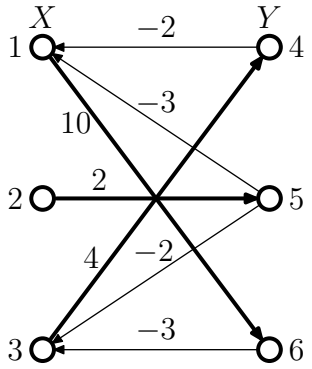


Рис. 3.39. Взвешенный орграф (G',w') без контуров отрицательного веса

Совершенное лексикографически меньшее паросочетание в двудольном графе. Задан двудольный граф $G = (X, Y, E)$, где X и Y — множества вершин первой и второй долей графа G соответственно. Предположим, что $|V(G)| = n$, n чётно и $|X| = |Y| = n/2$.

Пусть S — некоторое множество, на котором задан линейный порядок $\prec$.

Определение 3.3. *Лексикографическим порядком* на множестве S называется продолжение отношения на списки элементов из S, при котором

$$(s_1, s_2, \ldots, s_p) \prec (t_1, t_2, \ldots, t_q)$$

означает выполнение одного из условий:

• существует такое целое j, что $s_j \prec t_j$ и для всех $i < j$ справедливо $s_i = t_i$;

• $p \leqslant q$ и $s_i = t_i$ при $1 \leqslant i \leqslant p$.

Предположим, что вершины первой доли графа G (т. е. вершины множества X) занумерованы целыми числами от 1 до $n/2$, а вершины второй доли (т. е. вершины множества Y) — от $n/2 + 1$ до n. Тогда для двух паросочетаний M и M' паросочетание M *лексикографически меньше*, чем M' ($M \prec M'$), если список вершин, смежных в паросочетании M вершинам $1, 2, \ldots, n/2$, лексикографически меньше, чем список вершин, смежных в паросочетании M' вершинам $1, 2, \ldots, n/2$. Например, для двух паросочетаний

$$M = \big\{\{1, 5\}, \{2, 4\}, \{3, 6\}\big\}$$

и

$$M' = \big\{\{1, 4\}, \{2, 6\}, \{3, 5\}\big\}$$

паросочетание M' лексикографически меньше, чем M, так как выполняется соотношение $(4, 6, 5) \prec (5, 4, 6)$.

Рассмотрим задачу нахождения совершенного лексикографически меньшего паросочетания для двудольного графа. Для решения данной задачи сначала построим для заданного двудольного графа наибольшее паросочетание M, используя, например, любую из реализаций алгоритма Куна, описанных в подразделе 3.3.1. Если мощность построенного наибольшего паросочетания M равна $n/2$, то данное паросоче-

тание является совершенным, в противном случае совершенного паросочетания для данного двудольного графа не существует. Приведём алгоритм, который, основываясь на некотором текущем совершенном паросочетании двудольного графа, находит лексикографически меньшее совершенное паросочетание.

Алгоритм построения совершенного лексикографически меньшего паросочетания в двудольном графе

0. Пусть M — произвольное совершенное паросочетание для заданного двудольного графа $G = (X, Y, E)$. Полагаем все вершины графа G незаблокированными.

1. Пока существует более чем одна незаблокированная вершина в множестве X, выбираем из X незаблокированную вершину с меньшим номером (предположим, что выбрана вершина u) и выполняем следующие действия.

1.1. Ориентируем рёбра графа из паросочетания M от первой доли ко второй, а все оставшиеся рёбра двудольного графа — от второй доли к первой. Получаем орграф G'.

1.2. В орграфе G' находим множество вершин второй доли, достижимых из вершины u (например, используя поиск в ширину, строим дерево поиска в ширину с корнем в вершине u; при выполнении такого поиска заблокированные вершины посещать запрещено). Обозначим через Y_u множество вершин второй доли, которые достижимы из вершины u.

1.3. Из множества Y_u удаляем вершины, которые не соединены ребром с вершиной u в графе G, а затем из оставшихся в множестве Y_u вершин выбираем ту, номер которой меньше (предположим, что это вершина v).

1.4. Используя построенное на шаге 1.2 алгоритма дерево поиска в ширину, восстанавливаем путь P из вершины u в вершину v. Заменяем дуги пути P рёбрами и добавляем к данному маршруту ребро uv, получая при этом простой цикл C (чётной длины).

1.5. Преобразуем паросочетание M вдоль цикла C по следующему правилу: рёбра цикла C, которые не принадлежали паросо-

четанию, добавляем к паросочетанию, а рёбра цикла C, принадлежавшие паросочетанию, удаляем из него. В результате получаем совершенное паросочетание M', которое лексикографически меньше, чем M, т. е. $M' \prec M$.

1.6. Полагаем $M = M'$, блокируем вершины u и v и возвращаемся к шагу 1 алгоритма.

2. Построенное паросочетание M — лексикографически меньшее совершенное паросочетание в двудольном графе.

Трудоёмкость приведённого алгоритма построения лексикографически меньшего совершенного паросочетания для двудольного графа составляет $\mathrm{O}(nm)$ (в алгоритме $n/2$ раз запускается поиск в ширину).

Пример 3.8. Построим совершенное лексикографически меньшее паросочетание в двудольном графе $G = (X, Y, E)$ с $X = \{1, 2, 3\}$ и $Y = \{4, 5, 6\}$, приведённого на рис. 3.40.

Предположим, что при использовании, например, алгоритма Куна было найдено совершенное паросочетание

$$M = \big\{\{1, 6\}, \{2, 4\}, \{3, 5\}\big\}$$

(на рис. 3.40 рёбра совершенного паросочетания выделены жирными линиями). Построим по данному паросочетанию M ориентированный граф G' (рис. 3.41), все вершины орграфа G' полагаем незаблокированными.

Выбираем произвольную незаблокированную вершину первой доли, например $u = 1$. Из вершины $u = 1$ достижимы следующие вер-

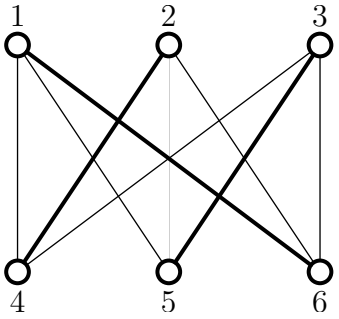


Рис. 3.40. Двудольный граф G и произвольное совершенное паросочетание M в нём

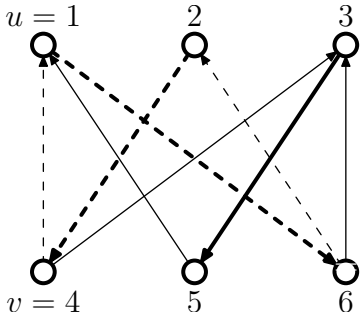


Рис. 3.41. Ориентированный граф G', построенный по совершенному паросочетанию

шины второй доли: $Y_1 = \{4, 5, 6\}$, причём каждая из них соединена с вершиной $u = 1$ в исходном графе G ребром. Выбираем из множества Y_1 вершину $v = 4$ с меньшим номером и восстанавливаем путь Q из вершины u в вершину v. Последовательность вершин пути Q следующая (на рис. 3.41 дуги пути выделены пунктирными линиями):

$$1 \to 6 \to 2 \to 4.$$

Если заменить дуги пути Q рёбрами и добавить к полученному множеству рёбер ребро $\{4, 1\}$, то получим простой цикл C чётной длины:

$$E(C) = \big\{\{1, 6\}, \{6, 2\}, \{2, 4\}, \{4, 1\}\big\}.$$

Преобразуем множество рёбер паросочетания

$$M = \big\{\{1, 6\}, \{2, 4\}, \{3, 5\}\big\}$$

вдоль цикла C, удаляя из паросочетания рёбра $\{1, 6\}$ и $\{2, 4\}$ и добавляя к паросочетанию рёбра $\{1, 4\}$ и $\{2, 6\}$. В результате выполненных преобразований получим новое лексикографически меньшее чем M паросочетание M' (на рис. 3.42 рёбра нового паросочетания выделены жирными линиями):

$$M' = \big\{\{1, 4\}, \{2, 6\}, \{3, 5\}\big\}.$$

Блокируем вершины 1 и 4, полагаем $M = M'$ (на рис. 3.42 заблокированные вершины выделены тёмными кружками).

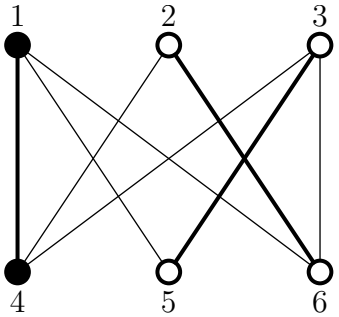


Рис. 3.42. Модифицированное совершенное паросочетание M' в двудольном графе

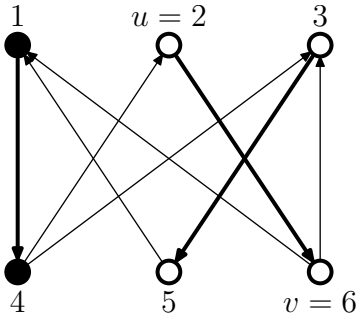


Рис. 3.43. Ориентированный граф G'', построенный по совершенному паросочетанию

Строим по новому паросочетанию $M = \{\{1,4\},\{2,6\},\{3,5\}\}$ ориентированный граф (рис. 3.43). Выбираем произвольную незаблокированную вершину первой доли, например $u = 2$. Из вершины 2 достижимы следующие незаблокированные вершины второй доли: $Y_2 = \{5,6\}$, из которых только вершина $v = 6$ соединена в исходном графе G ребром с вершиной $u = 2$. Однако так как ребро $\{2,6\} \in M$, то текущее паросочетание M не изменится. Блокируем вершины 2 и 6 (на рис. 3.44 текущее паросочетание M и заблокированные вершины закрашены).

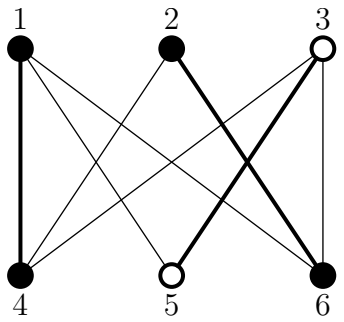


Рис. 3.44. Лексикографически меньшее совершенное паросочетание в двудольном графе

Поскольку осталась ровно одна незаблокированная вершина первой доли графа G (вершина с номером 3), то алгоритм завершает свою работу и текущее паросочетание

$$M = \{\{1,4\},\{2,6\},\{3,5\}\}$$

является лексикографически меньшим совершенным паросочетанием для двудольного графа, приведённого на рис. 3.40.

3.4. СПЕЦИАЛЬНЫЕ КРАТЧАЙШИЕ МАРШРУТЫ

В настоящем разделе для взвешенного графа (G, w) и заданной пары s и t его вершин рассматриваются задачи, которые являются обобщением известной задачи о кратчайшей цепи между заданной парой вершин в графе и имеют естественные приложения, связанные, в частности, с проектированием устойчивых к сбоям сетей оптико-волоконной связи [8, 9].

Пусть (G, w) — взвешенный граф ($w\colon E \to \mathbb{R}^+$), s и t — две различные вершины G. Обозначим через $P_{s,t} = \{P_i\}$ — множество всех (s,t)-маршрутов в графе G. Без ограничения общности будем считать, что $w(P_i) \leqslant w(P_{i+1})$ (т. е. элементы множества $P_{s,t}$ упорядочены по невозрастанию длин).

Для заданного $k \geqslant 1$ *задача построения в графе* (G, w) *первых* k *кратчайших* (s,t)*-маршрутов* заключается в нахождении первых k элементов множества $P_{s,t}$. Рассмотрим несколько вариантов этой задачи.

Если потребовать, чтобы все (s,t)-маршруты множества $P_{s,t}$ были простыми цепями, то получим *задачу нахождения первых* k *кратчайших простых* (s,t)*-цепей.*

Потребуем, чтобы все (s,t)-маршруты множества $P_{s,t}$ были простыми цепями и попарно не пересекались по рёбрам (или по вершинам, за исключением вершин s и t). Для $k \geqslant 1$ в графе (G, w) необходимо выбрать k маршрутов $P'_1, P'_2, \ldots, P'_k$ множества $P_{s,t}$ с минимальной суммарной длиной $\sum_{i=1}^{k} w(P'_i)$. Получаем *задачу нахождения оптимального* k*-множества простых рёберно-непересекающихся* (s,t)*-цепей.*

В данном разделе для взвешенного графа (G, w) заданной пары вершин s и t приведены алгоритмы решения следующих задач:

1) нахождение кратчайшей простой (s,t)-цепи;

2) нахождение кратчайшей простой (s,t)-цепи с ограничением на число рёбер;

3) нахождение первых k кратчайших простых (s,t)-цепей;

4) нахождение первых k кратчайших (s,t)-маршрутов;

5) нахождение оптимального k-множества простых рёберно-непересекающихся (вершинно-непересекающихся, за исключением вершин s и t) (s,t)-цепей.

3.4.1. Кратчайшая простая цепь

В данном подразделе приведено подробное описание и обоснование корректности классического алгоритма Дейкстры для решения базовой задачи построения кратчайшей простой цепи в графе. Также продемонстрирована одна из реализаций этого алгоритма с использованием структуры данных приоритетная очередь. Приводятся модификации

алгоритма Дейкстры и поиска в ширину для решения задачи поиска кратчайшего пути для специального класса смешанных графов.

Алгоритм Дейкстры. Пусть (G, w) — взвешенный граф, s и t — две его вершины. Оригинальный алгоритм Дейкстры корректно решает задачу поиска кратчайшей простой (s,t)-цепи для взвешенных графов, длины рёбер которых являются неотрицательными числами.

Общая схема алгоритма Дейкстры

Функция Найти кратчайшую цепь:
(**взвешенный граф** $(G,\ w)$, **вершина** s,
вершина t) $\to$ **цепь**

Для всех u **из** $V(G)$
    $L(u) \leftarrow +\infty$.
$L(s) \leftarrow 0$.
$S \leftarrow \varnothing$.

Пока $t \notin S$
    $u \leftarrow \arg\min_{v \in V(G) \setminus S} \{L(v)\}$.
    $S \leftarrow S \cup \{u\}$.
    Для всех v **из** $N(u)$
        Если $L(v) > L(u) + w(uv)$
        // *Выполняется релаксация по ребру* uv.
            $L(v) \leftarrow L(u) + w(uv)$.
            $P(v) \leftarrow u$.
$p_1 \leftarrow t$.
$k \leftarrow 1$.
$u \leftarrow t$.

Пока $u \neq s$
    $u \leftarrow P(u)$.
    $k \leftarrow k + 1$.
    $p_k \leftarrow u$.
Вернуть $(p_i)_{i=1}^{k}$.

В процессе работы алгоритма формируются массивы L и P, а также множество помеченных вершин S (помеченные вершины — верши-

ны, в которые уже построены кратчайшие простые цепи из стартовой вершины s). Каждая запись $L(u)$ содержит длину текущей простой (s,u)-цепи, в то время как $P(u)$ содержит предка вершины u, т. е. вершину, из которой вершина u была достигнута (помечена) в ходе построения текущей (s,u)-цепи (массив P используется на последнем этапе алгоритма для восстановления кратчайшей простой (s,t)-цепи). После завершения алгоритма величина $L(t)$ равна длине кратчайшей простой (s,t)-цепи.

Заметим, что если нам нужно найти кратчайшие простые цепи из s во все достижимые из неё вершины графа, то этот алгоритм может быть модифицирован заменой условия первого цикла «`Пока`» на проверку, что не все вершины помечены ($S \neq V(G)$) и расстояние до ближайшей непомеченной вершины u конечное ($L(u) < +\infty$).

В алгоритме Дейкстры количество значений, которые может принять переменная u, в худшем случае равно n. Поэтому сложность алгоритма есть $\mathrm{O}(n^2)$, если находить каждое новое значение u непосредственно перебором всех кандидатов.

Докажем корректность алгоритма Дейкстры. На итерациях алгоритма Дейкстры вершина v в результате релаксации по ребру uv получает временную метку $L(v)$, которая равна длине некоторой простой (s,v)-цепи. Корректность алгоритма следует из того факта, что если в алгоритме Дейкстры временная метка $L(v)$ вершины v объявляется постоянной (это происходит в момент, когда вершина v добавляется к множеству S), то значение метки $L(v)$ равно длине кратчайшей простой (s,v)-цепи.

В дальнейшем длину кратчайшей простой (u,v)-цепи будем обозначать через $D(u,v)$.

Лемма 3.1. *Пусть $P = \{v_1, \ldots, v_k\}$ — кратчайшая простая цепь между v_1 и v_k, а $p_{ij} = \{v_i, v_{i+1}, \ldots, v_j\}$ (при $1 \leqslant i \leqslant j \leqslant k$) — некоторая (v_i, v_j)-подцепь цепи P:*

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

Тогда цепь p_{ij} — кратчайшая простая (v_i, v_j)-цепь.

Доказательство. Доказательство выполним от противного. Допустим, что существует простая цепь p'_{ij} между вершинами v_i и v_j,

длина которой меньше, чем длина частичной простой цепи p_{ij}

$$w(p'_{ij}) < w(p_{ij}).$$

Тогда длина новой простой (v_1, v_k)-цепи P'

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

меньше, чем длина простой цепи P, так как

$$w(P') = w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) < w(P).$$

Приходим к противоречию с условием, что цепь P — кратчайшая простая (v_1, v_k)-цепь. □

Лемма 3.2 (неравенство треугольника)**.** *Справедливо следующее соотношение*

$$D(s, v) \leqslant D(s, u) + w(uv).$$

Доказательство. Кратчайшая простая (s, v)-цепь по длине не превосходит длины любой другой простой цепи между этими вершинами, например длины простой цепи из s в v, состоящей из кратчайшей простой (s, u)-цепи и ребра uv (рис. 3.45). □

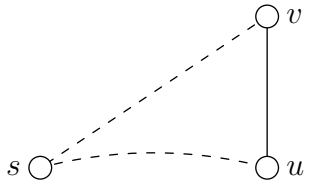


Рис. 3.45. Неравенство треугольника

Лемма 3.3 (свойство верхней границы для длины кратчайшей простой цепи)**.** *На каждой итерации алгоритма Дейкстры для временных меток вершин графа G справедливы следующие неравенства:*

$$L(v) \geqslant D(s, v), \quad v \in V, \tag{3.2}$$

и, как только временная метка $L(v)$ достигает свою нижнюю границу (т. е. выполняется равенство $L(v) = D(s, v)$), то она больше не изменяется.

Доказательство. Доказательство проведём индукцией по числу шагов релаксации рёбер. При начальной инициализации неравенство (3.2) выполняется для всех вершин, так как $L(s) = 0$, а $L(v) = +\infty$ для любой вершины $v \in V \setminus \{s\}$.

При релаксации по ребру uv свою метку может изменить (уменьшить) только вершина v, и если это происходит, то

$$\begin{aligned} L(v) &= L(u) + w(uv) \geqslant \\ &\geqslant \big(\text{по индукционному предположению } L(u) \geqslant D(s,u)\big) \geqslant \\ &\geqslant D(s,u) + w(uv) = \big(\text{по лемме 3.1}\big) = D(s,v). \end{aligned}$$

Таким образом, $L(v) \geqslant D(s,v)$, что и требовалось доказать. Более того, метка $L(v)$, достигая своей нижней границы $D(s,v)$, больше не изменяется. Действительно, изменить метку может только релаксация, которая может лишь уменьшить метку, что невозможно, так как метка $L(v)$ уже достигла своей нижней границы. □

Лемма 3.4. *После релаксации по ребру uv для вершин u и v графа G выполняется неравенство*

$$L(v) \leqslant L(u) + w(uv). \tag{3.3}$$

Доказательство. Если до релаксации по ребру uv выполнялось неравенство

$$L(v) > L(u) + w(uv),$$

то после релаксации $L(v) = L(u) + w(uv)$, а значит, неравенство (3.3) верно.

Если до релаксации

$$L(v) \leqslant L(u) + w(uv),$$

то после релаксации метки $L(u)$ и $L(v)$ не изменят своего значения, а значит, неравенство (3.3) верно и в этом случае. □

Лемма 3.5 (свойство сходимости)**.** *Пусть задана кратчайшая простая (s,v)-цепь*

$$s \leftrightsquigarrow u - v$$

и выполнена релаксация по ребру uv. Если до релаксации выполняется равенство

$$L(u) = D(s, u),$$

то после релаксации будет выполняться соотношение

$$L(v) = D(s, v).$$

Доказательство. Если $L(u) = D(s, u)$, то по лемме 3.3 значение метки $L(u)$ не изменится на протяжении всей работы алгоритма Дейкстры. После релаксации по ребру uv в силу леммы 3.4 выполняется следующее соотношение:

$$L(v) \leqslant L(u) + w(uv) = D(s, u) + w(uv) = \big(\text{по лемме } 3.1\big) = D(s, v).$$

Таким образом,

$$L(v) \leqslant D(s, v).$$

Но по лемме 3.3 для временных меток вершин справедливо неравенство

$$L(v) \geqslant D(s, v).$$

Следовательно, $L(v) = D(s, v)$, что и требовалось доказать. □

Теорема 3.7 (корректность алгоритма Дейкстры)**.** *После завершения работы алгоритма Дейкстры справедливы равенства*

$$L(u) = D(s, u), \quad u \in V.$$

Доказательство. Доказательство выполним от противного. Допустим, что u — первая вершина, после добавления которой в множество S выполняется неравенство

$$L(u) \neq D(s, u).$$

Рассмотрим кратчайшую простую (s, u)-цепь P. Цепь P может содержать вершины, как принадлежащие множеству S, так и не принадлежащие ему. Пусть y — первая от s вершина на цепи P, которая не принадлежит множеству S, а вершина x предшествует в цепи P вершине y (рис. 3.46). Поскольку вершина $x \in S$, то $L(x) = D(s, x)$, и

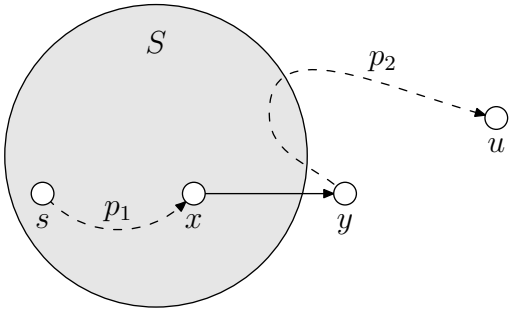


Рис. 3.46. Доказательство корректности алгоритма Дейкстры

по лемме 3.3 это соотношение остаётся верным до конца работы алгоритма Дейкстры. После добавления вершины x к множеству S была выполнена релаксация по ребру xy, и по лемме 3.5 для временной метки вершины y выполняется равенство

$$L(y) = D(s, y).$$

Поскольку длины всех рёбер графа G неотрицательны и вершина y лежит на цепи P до вершины u, то

$$D(s, y) \leqslant D(s, u).$$

Поэтому

$$L(y) = D(s, y) \leqslant D(s, u) \leqslant \big(\text{по лемме 3.3}\big) \leqslant L(u). \tag{3.4}$$

Но так как на очередной итерации алгоритма Дейкстры среди вершин, не принадлежащих множеству S, в качестве вершины с минимальной меткой была выбрана вершина u, то

$$L(u) \leqslant L(y). \tag{3.5}$$

Из неравенств (3.4) и (3.5) следует, что $L(y) = L(u)$, поэтому в соотношении (3.4) два неравенства являются равенствами, т. е. $L(u) = D(s, u)$. Получаем противоречие со сделанным предположением о том, что u — вершина, после добавления которой в множество S выполняется неравенство

$$L(u) \neq D(s, u).$$

Более того, в силу леммы 3.3 метка $L(u) = D(s, u)$ сохраняет своё значение на протяжении всей дальнейшей работы алгоритма Дейкстры. Доказательство корректности алгоритма Дейкстры завершено. □

Замечание 3.2. Небольшая модификация нахождения кратчайшей цепи позволяет получить информацию о нескольких кратчайших простых цепях между заданной парой вершин, если они существуют. Если мы заменим условие

$$L(v) > L(u) + w(uv)$$

на

$$L(v) \geqslant L(u) + w(uv),$$

то выполнение этого условия даст нам информацию о дополнительных кратчайших простых цепях (при этом также понадобятся дополнительные массивы для хранения всех возможных предков каждой вершины).

Заметим, что самой трудоёмкой операцией алгоритма Дейкстры является поиск минимального элемента (вершины с наименьшей временной меткой $L(u)$) в n-элементном массиве L. Для выполнения операций поиска и удаления минимального элемента более эффективной структурой данных является *куча* (*приоритетная очередь*). Напомним, что если кучу реализовать в виде полного бинарного дерева (*бинарная куча*), то операции добавления нового элемента и удаления минимального элемента выполняются за время $\mathrm{O}(\log n)$, где n — число элементов в куче [2].

Приведём реализацию алгоритма Дейкстры, используя в качестве структуры данных приоритетную очередь.

Реализация алгоритма Дейкстры с использованием структуры данных приоритетная очередь (куча)

1. Все вершины взвешеного графа (G, w) полагаем непросмотренными. Добавляем в кучу стартовую вершину s с временной меткой (*приоритетом*) $L(s) = 0$.

2. Пока не просмотрена конечная вершина t или куча не станет пустой, выполняем следующую последовательность шагов:

   2.1. из кучи удаляется вершина u (с минимальным приоритетом $L(u)$);

   2.2. если вершина u является просмотренной, т. е. эта вершина ранее уже удалялась из кучи, то она игнорируется и мы осуществляем переход к шагу 2 алгоритма;

2.3. если вершина u первый раз удаляется из кучи, то:

• временная метка $L(u)$, с которой данная вершина удалена из кучи, становится её постоянной меткой (*потенциалом*), а сама вершина u полагается просмотренной;

• для вершины u находятся смежные с ней непросмотренные вершины (эти вершины уже могут быть в куче) и каждая такая вершина v добавляется в кучу с временной меткой (приоритетом) $L(v) = L(u) + w(uv)$.

После завершения работы алгоритма все вершины u, достижимые из стартовой вершины s, имеют постоянные метки, которые соответствуют длине кратчайшей простой (s, u)-цепи. Для восстановления последовательности вершин цепи можно, например, одновременно с потенциалом вершины u хранить номер той вершины, из которой вершина u получила свой потенциал.

Трудоёмкость приведённой реализации алгоритма Дейкстры с использованием структуры данных приоритетная очередь (бинарная куча) есть $\mathrm{O}(n + m \log m)$.

Модификация алгоритма Дейкстры и алгоритма поиска в ширину для специальных классов смешанных графов. При решении ряда задач возникает специальный класс смешанных графов (рис. 3.47), которые обладают следующими свойствами:

• граф содержит дуги отрицательной длины и рёбра неотрицательной длины;

• в графе нет циклов отрицательной длины;

• отрицательные дуги составляют единую группу дуг, образующих (t, s)-путь; если дуги, образующие путь между вершинами t и s, заме-

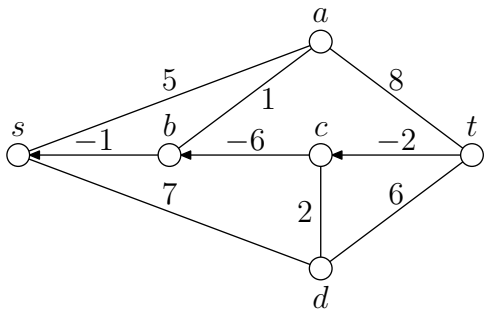


Рис. 3.47. Пример специального смешанного графа

нить рёбрами и обратить знак их длины, то маршрут, сформированный из этих рёбер, будет кратчайшей простой (s,t)-цепью в полученном графе с неотрицательными длинами рёбер.

Хорошо известно, что алгоритм Дейкстры работает некорректно в случае, когда в графе есть рёбра отрицательной длины. Однако особенность приведённого выше класса смешанных графов заключается в том, что если в некоторую вершину v ($v \neq t$) идёт (s,v)-путь и длина этого пути больше, чем длина некоторого текущего (s,t)-пути, то (s,v)-путь, будучи продолженным из вершины v до вершины t (если это возможно), породит (s,t)-путь большей длины, чем текущий (s,t)-путь. На этой особенности и основывается корректность приведённой ниже модификации алгоритма Дейкстры для специального класса смешанных графов. В данном алгоритме для некоторой вершины u под *окружением* $N(u)$ этой вершины будем понимать множество вершин смешанного графа, которое включает в себя как вершины, смежные с вершиной u, так и вершины, которые являются концами дуг, выходящих из вершины u. Например, для смешанного графа, приведённого на рис. 3.47, окружение вершины c задаётся следующим множеством:

$$N(c) = \{d, b\}.$$

Модификация алгоритма Дейкстры для специального класса смешанных графов

Функция Найти кратчайший путь:
(**взвешенный граф** $(G,\ w)$, **вершина** s,
вершина t) $\rightarrow$ **путь**

Для всех u **из** $V(G)$
    $L(u) \leftarrow +\infty$.
$L(s) \leftarrow 0$.
$S \leftarrow V(G)$.

Пока $t \in S$
    $u \leftarrow \arg\min_{v \in V(G) \setminus S}\{L(v)\}$.
    Если $L(u) = +\infty$
        Вернуть $\varnothing$.

$S \leftarrow S \setminus \{u\}$.
Для всех v **из** $N(u)$
Если $L(v) > L(u) + w(uv)$
$L(v) \leftarrow L(u) + w(uv)$.
$P(v) \leftarrow u$.
$S \leftarrow S \cup \{v\}$.
$p_1 \leftarrow t$.
$k \leftarrow 1$.
$u \leftarrow t$.

Пока $u \neq s$
$u \leftarrow P(u)$.
$k \leftarrow k + 1$.
$p_k \leftarrow u$.

Вернуть $(p_i)_{i=1}^{k}$.

Здесь множество S содержит все непомеченные вершины.

Пример 3.9. Для специального смешанного графа, приведённого на рис. 3.47, проиллюстрируем пошаговую работу алгоритма поиска кратчайшего пути.

Итерация 0. Множество $S = \{a, b, c, d, t\}$. Метки вершин:

Вершина u	s	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{t}$
$L(u)$	0	5	$+\infty$	$+\infty$	7	$+\infty$
$P(u)$	0	s	0	0	s	0

В дальнейшем вершины, которые принадлежат множеству S, в таблице будут выделены.

Итерация 1. Среди вершин множества S минимальную метку L имеет вершина $u = a$. Новое множество $S = \{b, c, d, t\}$.

Обновлённые метки вершин:

Вершина u	s	a	$\boldsymbol{b}$	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{t}$
$L(u)$	0	5	6	$+\infty$	7	13
$P(u)$	0	s	a	0	s	a

Итерация 2. Среди вершин множества S минимальную метку L имеет вершина $u = b$. Новое множество $S = \{c, d, t\}$.

Обновлённые метки вершин:

Вершина u	s	a	b	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{t}$
$L(u)$	0	5	6	$+\infty$	7	13
$P(u)$	0	s	a	0	s	a

Итерация 3. Среди вершин множества S минимальную метку L имеет вершина $u = d$. Новое множество $S = \{c, t\}$.

Обновлённые метки вершин:

Вершина u	s	a	b	$\boldsymbol{c}$	d	$\boldsymbol{t}$
$L(u)$	0	5	6	9	7	13
$P(u)$	0	s	a	d	s	a

Итерация 4. Среди вершин множества S минимальную метку L имеет вершина $u = c$. Новое множество $S = \{b, t\}$ (вершина b вернулась в множество S, так как через вершину c уменьшила свою метку).

Обновлённые метки вершин:

Вершина u	s	a	$\boldsymbol{b}$	c	d	$\boldsymbol{t}$
$L(u)$	0	5	3	9	7	13
$P(u)$	0	s	c	d	s	a

Итерация 5. Среди вершин множества S минимальную метку L имеет вершина $u = b$. Новое множество $S = \{a, t\}$ (вершина a вернулась в множество S, так как через вершину b уменьшила свою метку).

Обновлённые метки вершин:

Вершина u	s	$\boldsymbol{a}$	b	c	d	$\boldsymbol{t}$
$L(u)$	0	4	3	9	7	13
$P(u)$	0	b	c	d	s	a

Итерация 6. Среди вершин множества S минимальную метку L имеет вершина $u = a$. Новое множество $S = \{t\}$.

Обновлённые метки вершин:

Вершина u	s	a	b	c	d	$\boldsymbol{t}$
$L(u)$	0	4	3	9	7	12
$P(u)$	0	b	c	d	s	a

Итерация 7. Среди вершин множества S минимальную метку L имеет вершина $u = t$. Алгоритм завершает свою работу.

Кратчайший (s,t)-путь

$$s - d - c \to b - a - t$$

имеет длину $L(t) = 12$ (рёбра и дуги кратчайшего пути на рис. 3.48 выделены жирными линиями).

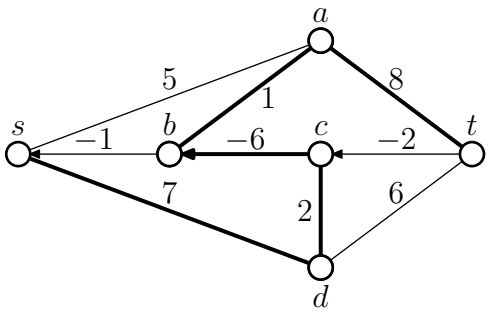


Рис. 3.48. Кратчайший (s,t)-путь в смешанном графе

Замечание 3.3. Для того чтобы функция нахождения кратчайшего пути корректно работала для произвольного графа с отрицательными длинами рёбер (дуг), необходимо условие окончания цикла $u = t$ заменить на условие $S \neq \varnothing$.

Другой алгоритм, находящий кратчайший путь для специального класса смешанных графов с отрицательными длинами дуг, — модифицированный поиск в ширину.

Модификация поиска в ширину для специального класса смешанных графов

Функция Найти кратчайший путь поиском в ширину:
(**взвешенный граф** $(G,\ w)$, **вершина** s, **вершина** t) $\to$ **путь**
    Для всех u **из** $V(G)$
        $L(u) \leftarrow +\infty$.
    $L(s) \leftarrow 0$.
    $S \leftarrow \{s\}$.

    Пока $S \neq \varnothing$
        $T \leftarrow \varnothing$.
        Для всех u **из** S
            Для всех v **из** $N(u)$

Если $L(v) > L(u) + w(uv)$
и $L(t) > L(u) + w(uv)$
$L(v) \leftarrow L(u) + w(uv)$.
$P(v) \leftarrow u$.
$T \leftarrow T \cup \{v\}$.
$S \leftarrow T \setminus \{t\}$.
$p_1 \leftarrow t$.
$k \leftarrow 1$.
$u \leftarrow t$.

Пока $u \neq s$
$u \leftarrow P(u)$.
$k \leftarrow k + 1$.
$p_k \leftarrow u$.

Вернуть $(p_i)_{i=1}^{k}$.

В функции нахождения кратчайшего пути поисков в ширину множество S имеет несколько иное значение, чем в алгоритме Дейкстры. Теперь S — множество тех вершин u, которые были помечены (т. е. расстояние до которых из вершины s изменилось) на предыдущей итерации (из множества S исключается конечная вершина t). На каждой итерации основного цикла пересчитываются метки для всех вершин, смежных с вершинами из множества S: если вершина v уменьшает свою метку $L(v)$ и при этом новая метка вершины v становится меньше, чем метка вершины t, то v добавляется к множеству вершин, помеченных на данной итерации. Алгоритм завершает свою работу, как только множество вершин, изменивших метки на текущей итерации, становится пустым или состоит из единственной конечной вершины t.

Пример 3.10. Для специального смешанного графа, приведённого на рис. 3.47, проиллюстрируем пошаговую работу алгоритма.

Итерация 0. Множество $S = \{s\}$. Метки вершин:

Вершина u	s	a	b	c	d	t
$L(u)$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$P(u)$	0	0	0	0	0	0

Итерация 1. Множество $S = \{s\}$. Новые метки вершин:

Вершина u	s	a	b	c	d	t
$L(u)$	0	5	$+\infty$	$+\infty$	7	$+\infty$
$P(u)$	0	s	0	0	s	0

Множество $T = \{a, d\}$.

Итерация 2. Множество $S = \{a, d\}$. Новые метки вершин:

Вершина u	s	a	b	c	d	t
$L(u)$	0	5	8	9	7	13
$P(u)$	0	s	a	d	s	a

Множество $T = \{b, t, c\}$.

Итерация 3. Множество $S = \{b, c\}$. Новые метки вершин:

Вершина u	s	a	b	c	d	t
$L(u)$	0	5	3	9	7	13
$P(u)$	0	s	c	d	s	a

Множество $T = \{b\}$.

Итерация 4. Множество $S = \{b\}$. Новые метки вершин:

Вершина u	s	a	b	c	d	t
$L(u)$	0	4	3	9	7	13
$P(u)$	0	b	c	d	s	a

Множество $T = \{a\}$.

Итерация 5. Множество $S = \{a\}$. Новые метки вершин:

Вершина u	s	a	b	c	d	t
$L(u)$	0	4	3	9	7	12
$P(u)$	0	b	c	d	s	a

Множество $T = \{t\}$. Алгоритм завершает свою работу.

Кратчайший (s, t)-путь

$$s - d - c \to b - a - t$$

имеет длину $L(t) = 12$ (рёбра и дуги кратчайшего пути на рис. 3.48 выделены).

Замечание 3.4. Для того чтобы описанная функция корректно работала для произвольного графа с отрицательными длинами рёбер (дуг), необходимо убрать условие $L(t) > L(u) + w(uv)$ из условия релаксации.

Замечание 3.5. Кратчайший простой (s,t)-путь, построенный алгоритмом модифицированного поиска в ширину, является кратчайшим (s,t)-путём с наименьшим числом рёбер.

3.4.2. Кратчайшая простая цепь с ограничением на число рёбер

Алгоритм Дейкстры можно модифицировать для решения задачи нахождения кратчайшей простой (s,t)-цепи во взвешенном графе при условии, что в цепи должно содержаться не более чем q рёбер (т. е. совершено не более чем q «прыжков») [8].

Алгоритм, приведённый ниже, находит и возвращает кратчайшую простую (s,t)-цепь, содержащую не более чем q рёбер. В этом алгоритме вместо массивов L и P нам понадобятся массивы L_i и P_i (для $1 \leqslant i \leqslant q$), где в $L_i(u)$ хранится длина кратчайшей простой (s,u)-цепи, содержащей не более чем i рёбер, а в $P_i(u)$ — соответствующий предок. Также вместо множества помеченных вершин S для каждого $i = 1, 2, \ldots, q$ будет храниться отдельное множество S_i вершин, помеченных на уровне i, т. е. тех, для которых кратчайшая простая цепь с числом рёбер, равным i, была найдена. В алгоритме нам понадобится ещё одно множество R — все пары (u,i), где вершина u ещё не помечена на уровне i $(1 \leqslant i \leqslant q)$ и $L_i(u) < +\infty$, но может быть достигнута из стартовой вершины s с использованием ровно i рёбер.

Алгоритм построения кратчайшей простой (s,t)-цепи с ограничением q на число рёбер

Функция Найти цепь: (**взвешенный граф** $(G,\ w)$, **вершина** s, **вершина** t, **целое** q) $\rightarrow$ **цепь**
  Для всех i **от** 0 **до** q
    $S_i \leftarrow \varnothing$.
  Для всех u **из** $V(G)$
    Для всех i **от** 0 **до** q
      $L_i(u) \leftarrow +\infty$.

Для всех i **от** 1 **до** q
    $L_i(s) \leftarrow 0$.

$R \leftarrow \{(s,\ 0)\}$.
Пока $R \neq \varnothing$
    $\ell \leftarrow +\infty$.
    Для всех $(v,\ j)$ **из** R
        Если $L_j(v) < \ell$ **или** $(L_j(v) = \ell$ **и** $j < i)$
            $u \leftarrow v$.
            $i \leftarrow j$.
            $\ell \leftarrow L_i(u)$.
    $R \leftarrow R \setminus \{(u,\ i)\}$.
    $S_i \leftarrow S_i \cup \{u\}$.

    Если $i < q$
        Для всех v **из** $N(u) \setminus S_{i+1}$
            Если $L_{i+1}(v) > L_i(u) + w(uv)$
                $R \leftarrow R \cup \{(v,\ i+1)\}$.
            Для всех j **от** $i+1$ **до** q
                Если $L_j(v) > L_{j-1}(u) + w(uv)$
                    $L_j(v) \leftarrow L_{j-1}(u) + w(uv)$.
                    $P_j(v) \leftarrow u$.
$p_1 \leftarrow t$.
$k \leftarrow 1$.
$u \leftarrow t$.
$i \leftarrow q$.

Пока $u \neq s$
    $u \leftarrow P_i(u)$.
    $i \leftarrow i - 1$.
    $k \leftarrow k + 1$.
    $p_k \leftarrow u$.

Вернуть $(p_i)_{i=1}^{k}$.

Сложность рассматриваемого алгоритма не более чем в q раз больше, чем сложность базового алгоритма Дейкстры, а значит, не превосходит $\mathrm{O}(qn^2)$.

3.4.3. Первые k кратчайших простых цепей

В данном подразделе разберём задачу построения первых k кратчайших простых (s,t)-цепей для взвешенного графа (G,w).

Рассмотрим последовательность вершин некоторой простой цепи C между заданной парой вершин s и t:

$$s = v_0, v_1, v_2, \ldots, v_k, v_{k+1} = t.$$

Для цепи C обозначим через $C(v_i, v_j)$ подцепь из вершины v_i в вершину v_j при $0 \leqslant i \leqslant j \leqslant k+1$:

$$v_i, v_{i+1}, \ldots, v_{j-1}, v_j.$$

В процессе работы алгоритма построения первых k кратчайших простых (s,t)-цепей будут сформированы следующие множества:

1) S — множество пар (Q,u), где Q — некоторая сгенерированная алгоритмом простая (s,t)-цепь с *точкой* (вершиной) *отклонения* u, т. е. в множестве S существует простая (s,t)-цепь Q', в которой последовательности вершин цепей Q и Q' от вершины s до вершины u совпадают:

$$Q(s,u) = Q'(s,u),$$

а в вершине u пути Q и Q' отклоняются друг от друга.

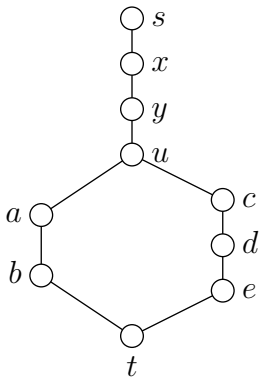


Рис. 3.49. Вершина отклонения u

На рис. 3.49 изображены две простые цепи:

$$Q' = s - x - y - u - c - d - e - t,$$
$$Q = s - x - y - u - a - b - t,$$

для которых $Q(s,u) = Q'(s,u)$, а в вершине u эти цепи отклоняются друг от друга;

2) X — множество возможных кратчайших простых (s,t)-цепей (при этом для каждой цепи хранится её точка отклонения);

3) K_{sc} — множество первых k кратчайших простых (s,t)-цепей.

Алгоритм начинает свою работу с построения в исходном взвешенном графе (G,w) кратчайшей простой (s,t)-цепи C, используя, например, алгоритм Дейкстры. Построенная кратчайшая простая (s,t)-цепь C с точкой отклонения s добавляется в множества S и X. Затем пока не будет построено требуемое множество первых k кратчайших простых (s,t)-цепей или множество возможных кратчайших простых (s,t)-цепей X не станет пустым, выполняются следующие шаги алгоритма:

- из множества X возможных кратчайших простых (s,t)-цепей удаляется (s,t)-цепь C минимальной длины (предположим, что точка отклонения цепи C равна u);
- (s,t)-цепь C добавляется к множеству K_{sc};
- по паре (C,u) генерируются возможные кратчайшие простые (s,t)-цепи и добавляются к множествам S и X.

Алгоритм построения
первых k кратчайших простых (s,t)-цепей

Функция Найти кратчайшие цепи:
(**взвешенный граф** $(G,\ w)$, **вершина** s,
вершина t, **целое** k) $\rightarrow$ **множество цепей**

$c \leftarrow 1$.
$P =$ Найти кратчайшую цепь $((G,\ w),\ s,\ t)$.
$S \leftarrow \{(P,\ s)\}$.
$X \leftarrow \{(P,\ s)\}$.
$K_{\text{sc}} \leftarrow \{P\}$.

Пока $c < k$ **и** $X \neq \varnothing$
$X \leftarrow X \setminus \{(P,\ u)\}$.
$Y \leftarrow$ Найти цепи$((G,\ w),\ P,\ s,\ t,\ u,\ S)$.
$X \leftarrow X \cup Y$.
$S \leftarrow S \cup Y$.
$(P,\ u) \leftarrow (Q,\ v)$ из X, для которой
длина $Q \leqslant$ длины R
для всех $(R,\ x)$ из X.
$K_{\text{sc}} \leftarrow K_{\text{sc}} \cup \{P\}$.
$c \leftarrow c + 1$.
Вернуть K_{sc}.

// Строит простые (s,t)-цепи
// по заданной простой (s,t)-цепи C,
// имеющей точку отклонения u.
Функция Найти цепи: (**взвешенный граф** $(G,\ w)$, **цепь** P,
вершина s, **вершина** t, **вершина** u,
множество цепей S) $\rightarrow$ **множество цепей**
$\overline{S} \leftarrow \varnothing$.

Для всех v **из** $V(P(u,\ t)) \setminus \{t\}$
$(G',\ w') \leftarrow$ Изменить граф$((G,\ w),\ P,\ v,\ S \cup \overline{S})$.
$\overline{Q} \leftarrow$ Найти кратчайшую цепь$((G',\ w'),\ v,\ t)$.
$Q \leftarrow P(s,\ v)\overline{Q}$.
$\overline{S} \leftarrow \overline{S} \cup \{(Q,\ v)\}$.
Вернуть $\overline{S}$.

Рассмотрим более подробно функцию «`Найти цепи`». Пусть P — некоторая кратчайшая простая (s,t)-цепь с точкой отклонения u, которая была получена на предыдущей итерации алгоритма (рис. 3.50).

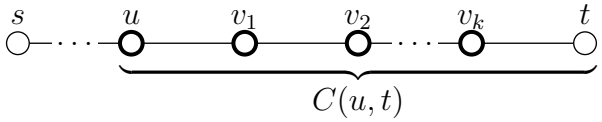


Рис. 3.50. Простая цепь P между парой вершин s и t с точкой отклонения u

Последовательно просматриваем все вершины цепи P от вершины u к вершине t, исключая вершину t (на рис. 3.50 просматриваемые вершины цепи выделены), и для каждой такой вершины v выполняем следующие действия:

1. Строим модифицированный граф $G'_{P,v}$ по правилам (функция «`Изменить граф`»):

1.1. удаляем в графе G все вершины подцепи P от вершины s до вершины v, кроме вершины v, и рёбра, инцидентные удалённым вершинам (на рис. 3.51 в качестве вершины v взята вершина v_i);

1.2. удаляем ребро (s,t)-цепи P, инцидентное вершине v, в направлении к вершине t (на рис. 3.51 жирным выделены вершины и рёбра графа G, оставшиеся после удаления);

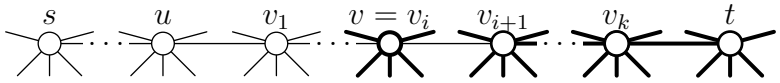


Рис. 3.51. Модифицированный граф $G'_{P,v}$

1.3. просматриваем множество S найденных простых (s,t)-цепей, и если для некоторой (s,t)-цепи Q' выполняется равенство

$$Q'(s,v) = P(s,v),$$

то удаляем ребро (s,t)-цепи Q', инцидентное вершине v в направлении вершины t.

2. В модифицированном графе $G'_{P,v}$ находим кратчайшую простую (v,t)-цепь $\overline{Q}(v,t)$.

3. Сцепляем подцепь $P(s,v)$ и цепь $\overline{Q}(v,t)$, получая при этом новую (s,t)-цепь Q. Следует отметить, что правило построения графа $G'_{P,v}$ гарантирует, что полученная (s,t)-цепь Q является простой и ранее не была сгенерирована этим алгоритмом. Добавляем пару (Q,v) к множествам S и X.

Если для поиска кратчайшей простой цепи на итерациях алгоритма использовать алгоритм Дейкстры (трудоёмкость алгоритма Дейкстры $\mathrm{O}(n^2)$), то трудоёмкость описанного алгоритма построения первых k кратчайших простых (s,t)-цепей есть $\mathrm{O}(kn^3)$.

Пример 3.11. Для взвешенного графа (G,w), приведённого на рис. 3.52, найдём первые k кратчайших простых (s,t)-цепей для $k=3$.

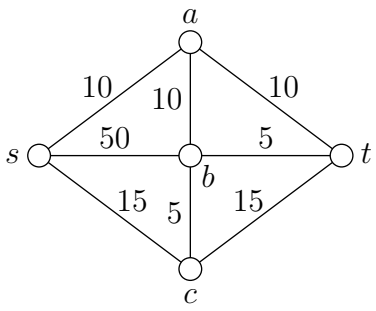


Рис. 3.52. Граф (G, w)

Итерация 1. В графе (G, w), изображённом на рис. 3.52, находим кратчайшую простую (s, t)-цепь

$$Q_1 = s - a - t$$

длины $w(Q_1) = 20$.

Формируем множества S, X и K_{sc} (в дальнейшем точку отклонения цепи будем подчеркивать):

$$\begin{aligned} S =& \big\{(Q_1 = \underline{s} - a - t,\ s)\big\}, \\ X =& \big\{(Q_1 = \underline{s} - a - t,\ s)\big\},\ w(Q_1) = 20, \\ K_{\text{sc}} =& \big\{Q_1 = s - a - t\big\},\ c = 1. \end{aligned}$$

Итерация 2. Удаляем цепь $Q_1 = \underline{s} - a - t$ минимальной длины из множества X. Точкой отклонения цепи Q_1 является вершина s. Выполняем ветвление цепи Q_1 в вершинах s и a.

При ветвлении цепи Q_1 в вершине s получим модифицированный граф $G'_{Q_1,s}$, изображённый на рис. 3.53. Находим в графе $G'_{Q_1,s}$ кратчайшую простую (s, t)-цепь

$$\overline{Q}(s, t) = \underline{s} - c - b - t$$

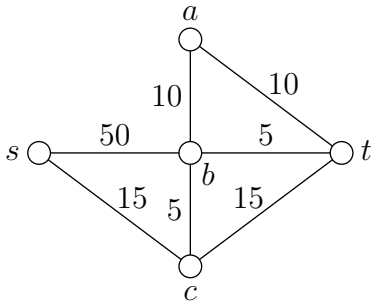


Рис. 3.53. Граф $G'_{Q_1,s}$

и, объединяя её с подцепью $Q_1(s,s)$, получаем простую (s,t)-цепь

$$Q_2 = \underline{s} - c - b - t$$

длины $w(Q_2) = 25$.

При ветвлении цепи Q_1 в вершине a получим модифицированный граф $G'_{Q_1,a}$, изображённый на рис. 3.54.

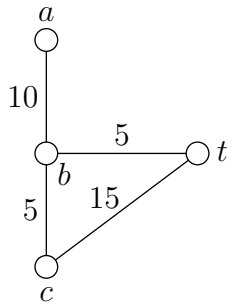


Рис. 3.54. Граф $G'_{Q_1,a}$

Находим в графе $G'_{Q_1,a}$ кратчайшую простую (a,t)-цепь

$$\overline{Q}(a,t) = a - b - t$$

и, объединяя её с подцепью $Q_1(s,a)$, получаем простую (s,t)-цепь

$$Q_3 = s - \underline{a} - b - t$$

длины $w(Q_3) = 25$.

Процедура генерирования простых (s,t)-цепей по цепи Q_1 завершена. Полученные цепи Q_2 и Q_3 добавляем в множества S и X:

$$\begin{gathered} S = \big\{(Q_1 = \underline{s} - a - t, s),\ (Q_2 = \underline{s} - c - b - t, s), \\ (Q_3 = s - \underline{a} - b - t, a)\big\}, \\ X = \big\{(Q_2 = \underline{s} - c - b - t, s),\ (Q_3 = s - \underline{a} - b - t, a)\big\}, \\ w(Q_2) = 25,\ w(Q_3) = 25. \end{gathered}$$

Находим в множестве X цепь минимальной длины

$$Q_2 = \underline{s} - c - b - t,$$

удаляем её из множества X и добавляем в множество K_{sc}. В результате описанных действий получаем новые множества:

$$X = \big\{(Q_3 = s - \underline{a} - b - t,\ a)\big\},\ w(Q_3) = 25,$$
$$K_{\text{sc}} = \big\{Q_1 = s - a - t,\ Q_2 = s - c - b - t\big\},\ c = 2.$$

Итерация 3. По цепи Q_2 генерируем простые (s,t)-цепи. Точкой отклонения цепи Q_2 является вершина s. Выполняем ветвление цепи Q_2 в вершинах s, c и b.

При ветвлении цепи Q_2 в вершине s получим модифицированный граф $G'_{Q_2,s}$, изображённый на рис. 3.55. Находим в графе $G'_{Q_2,s}$ кратчайшую простую (s,t)-цепь

$$\overline{Q}(s,t) = \underline{s} - b - t$$

и, объединяя её с подцепью $Q_2(s,s)$, получаем простую (s,t)-цепь

$$Q_4 = \underline{s} - b - t$$

длины $w(Q_4) = 55$.

При ветвлении цепи Q_2 в вершине c получим модифицированный граф $G'_{Q_2,c}$, изображённый на рис. 3.56. Находим в графе $G'_{Q_2,c}$ кратчайшую простую (c,t)-цепь

$$\overline{Q}(c,t) = c - t$$

и, объединяя её с подцепью $Q_2(s,c)$, получаем простую (s,t)-цепь

$$Q_5 = s - \underline{c} - t$$

длины $c(Q_5) = 30$.

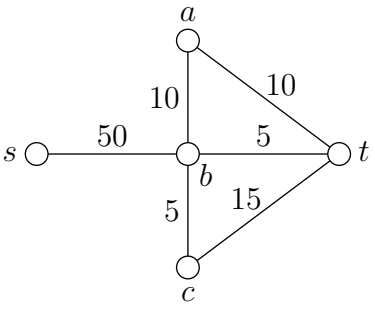


Рис. 3.55. Граф $G'_{Q_2,s}$

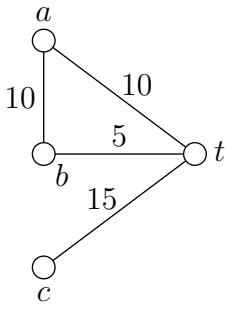


Рис. 3.56. Граф $G'_{Q_2,c}$

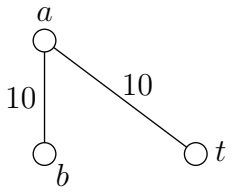


Рис. 3.57. Граф $G'_{Q_2,b}$

При ветвлении цепи Q_2 в вершине b получим модифицированный граф $G'_{Q_2,b}$, изображённый на рис. 3.57. Находим в графе $G'_{Q_2,c}$ кратчайшую простую (b,t)-цепь

$$\overline{Q}(b,t) = b - a - t$$

и, объединяя её с подцепью $Q_2(s,c)$, получаем простую (s,t)-цепь

$$Q_6 = s - c - \underline{b} - a - t$$

длины $w(Q_6) = 40$.

Процедура генерирования (s,t)-цепей по цепи Q_2 завершена. Полученные цепи Q_4, Q_5 и Q_6 добавляем в множества S и X:

$$\begin{gathered}
S = \big\{(Q_1 = \underline{s} - a - t,\ s),\ (Q_2 = \underline{s} - c - b - t,\ s),\\
(Q_3 = s - \underline{a} - b - t,\ a),\ (Q_4 = \underline{s} - b - t,\ s),\\
(Q_5 = s - \underline{c} - t,\ c),\ (Q_6 = s - c - \underline{b} - a - t,\ b)\big\},\\
X = \big\{(Q_3 = s - \underline{a} - b - t,\ a),\ (Q_4 = \underline{s} - b - t,\ s),\\
(Q_5 = s - \underline{c} - t,\ c),\ (Q_6 = s - c - \underline{b} - a - t,\ b)\big\},\\
w(Q_3) = 25,\ w(Q_4) = 55,\ w(Q_5) = 30,\ w(Q_6) = 40.
\end{gathered}$$

Находим в множестве X цепь

$$Q_3 = s - \underline{a} - b - t,$$

минимальной длины, удаляем её из множества X

$$X = \big\{Q_4, Q_5, Q_6\big\}$$

и добавляем в множество K_{sc}

$$\begin{gathered}
K_{\text{sc}} = \big\{Q_1 = s - a - t,\ Q_2 = s - c - b - t,\\
Q_3 = s - a - b - t\big\},\ c = 3,\\
w(Q_1) = 20,\ w(Q_2) = 25,\ w(Q_3) = 25.
\end{gathered}$$

Алгоритм завершает свою работу, так как $c = k = 3$.

Пример 3.12. Для взвешенного графа (G, w), приведённого на рис. 3.58, найдём первые k кратчайших простых (s,t)-цепей для $k = 3$.

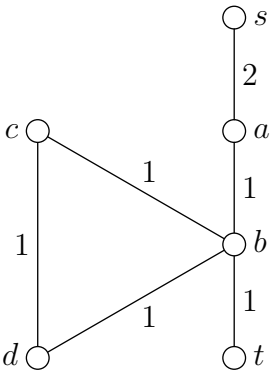


Рис. 3.58. Граф (G, w)

Итерация 1. В графе (G, w), изображённом на рис. 3.58, находим кратчайшую простую (s,t)-цепь $Q_1 = s{-}a{-}b{-}t$ длины $w(Q_1) = 4$. Формируем множества:

$$S = \big\{(Q_1 = \underline{s} - a - b - t,\ s)\big\},$$
$$X = \big\{(Q_1 = \underline{s} - a - b - t,\ s)\big\},\ w(Q_1) = 4,$$
$$K_{\text{sc}} = \big\{Q_1 = s - a - b - t\big\},\ c = 1.$$

Итерация 2. Удаляем из множества X цепь Q_1 минимальной длины. Точкой отклонения данной цепи является вершина s. Выполняем ветвление цепи Q_1 в вершинах s, a и b. При ветвлении цепи Q_1 в вершине s получим модифицированный граф $G'_{Q_1,s}$, изображённый на рис. 3.59.

В графе $G'_{Q_1,s}$ простой (s,t)-цепи не существует.

При ветвлении цепи Q_1 в вершине a получим модифицированный граф $G'_{Q_1,a}$, изображённый на рис. 3.60. В графе $G'_{Q_1,a}$ простой (a,t)-цепи не существует.

При ветвлении цепи Q_1 в вершине b получим модифицированный граф $G'_{Q_1,b}$, изображённый на рис. 3.61. В графе $G'_{Q_1,b}$ простой (b,t)-цепи не существует.

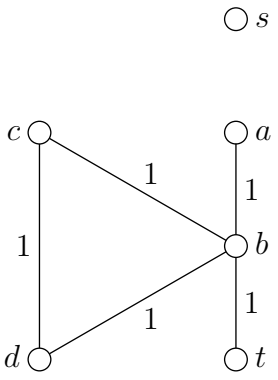


Рис. 3.59. Граф $G'_{Q_1,s}$

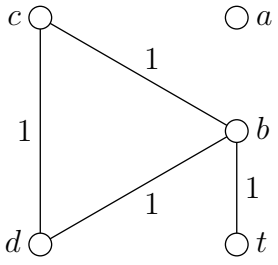


Рис. 3.60. Граф $G'_{Q_1,a}$

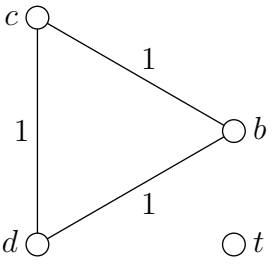


Рис. 3.61. Граф $G'_{Q_1,b}$

Процедура генерирования простых (s,t)-цепей по цепи Q_1 завершена. Текущие множества S и X:

$$S = \big\{(Q_1 = \underline{s} - a - b - t,\ s)\big\},$$
$$X = \varnothing,$$
$$K_{\text{sc}} = \big\{Q_1 = s - a - b - t\big\},\ w(Q_1) = 4.$$

Поскольку X — пустое множество, то алгоритм завершает свою работу. Таким образом, для взвешенного графа, приведённого на рис. 3.58, существует единственная простая (s,t)-цепь длины 4:

$$Q_1 = s - a - b - t.$$

Замечание 3.6. Пусть в графе (G,w) длина кратчайшей простой (s,t)-цепи равна p. Для графа (G,w) задача нахождения всех простых (s,t)-цепей длины p называется *задачей построения всех оптимальных простых цепей*. Алгоритм построения первых k кратчайших простых (s,t)-цепей можно использовать для решения данной задачи: к концу каждой итерации алгоритм добавляет к множеству K_{sc} ровно одну новую простую (s,t)-цепь, и необходимо завершить работу алгоритма, как только будет сделана попытка добавления к множеству K_{sc} цепи, длина которой больше, чем величина p.

3.4.4. Первые k кратчайших маршрутов

Рассмотрим задачу построения первых k кратчайших (s,t)-маршрутов во взвешенном графе (G,w). Заметим, что для маршрута допустимо, чтобы в нём некоторые вершины (рёбра) повторялись.

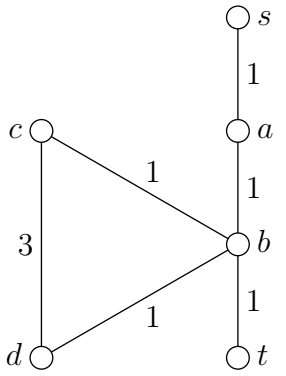


Рис. 3.62. Граф (G,w)

В качестве примера рассмотрим граф, изображённый на рис. 3.62. Пусть $k = 3$, тогда первый кратчайший (s,t)-маршрут будет иметь длину 3 и проходить по вершинам

$$s - a - b - t,$$

второй — длину 5 и проходить, например, по вершинам

$$s - a - b - c - b - t,$$

а третий — длину 5 и проходить, например, по вершинам

$$s - a - b - a - b - t.$$

Заметим, что второй и третий (s,t)-маршруты не являются цепями.

Для решения задачи построения первых k кратчайших (s,t)-маршрутов модифицируем реализацию алгоритма Дейкстры с использованием структуры данных куча, которая была приведена в подразделе 3.4.1. В частности, вершину u будем считать просмотренной только после её k-го удаления из кучи. В этом случае метка, с которой вершина u первый раз удаляется из кучи, равна длине первого кратчайшего (s,u)-маршрута; метка, с которой вершина u второй раз удаляется из кучи, равна длине второго кратчайшего (s,u)-маршрута и т. д. Алгоритм завершает свою работу, как только конечная вершина t будет удалена из кучи k-й по счёту раз (метка, с которой вершина t удаляется из кучи k-й раз, равна длине k-го кратчайшего (s,t)-маршрута).

Время работы описанной модификации алгоритма Дейкстры составляет $O(k(n + m\log m))$.

Пример 3.13. Для графа (G,w), изображённого на рис. 3.62, построим два первых кратчайших (s,t)-маршрута (т. е. $k = 2$).

Состояние кучи на каждой итерации алгоритма выделено в табл. 3.1 отдельным блоком, каждый элемент кучи состоит из следующих полей:

- имя вершины, например v;
- приоритет (временная метка), с которым вершина v была занесена в структуру;
- предок — имя вершины, например u ($uv \in E$), из которой вершина v была занесена в структуру (нижний индекс у предка u в табл. 3.1 показывает, какой раз по счёту вершина u была удалена из кучи).

Таблица 3.1

Куча

Удаление из кучи	Вершина	Приоритет	Предок
Удаление s (1-е удаление)	s	0	—
Удаление a (1-е удаление)	a	1	s_1
Удаление b (1-е удаление)	s	2	a_1
	b	2	a_1

Продолжение табл. 3.1

Удаление из кучи	Вершина	Приоритет	Предок
Удаление s (2-е удаление)	s	2	a_1
	a	3	b_1
	t	3	b_1
	c	3	b_1
	d	3	b_1
Удаление a (2-е удаление)	a	3	b_1
	t	3	b_1
	c	3	b_1
	d	3	b_1
	a	3	s_2
Удаление t (1-е удаление)	t	3	b_1
	c	3	b_1
	d	3	b_1
	a	3	s_2
	b	4	a_2
Удаление c (1-е удаление)	c	3	b_1
	d	3	b_1
	a	3	s_2
	b	4	a_2
	b	4	t_1
Удаление d (1-е удаление)	d	3	b_1
	a	3	s_2
	b	4	a_2
	b	4	t_1
	d	6	c_1
	b	4	c_1
Удаление a (уже просмотрена)	a	3	s_2
Удаление b (2-е удаление)	b	4	a_2
	b	4	t_1
	d	6	c_1
	b	4	c_1
	c	6	d_1
	b	4	d_1
Удаление b (уже просмотрена)	b	4	t_1
Удаление b (уже просмотрена)	d	6	c_1
Удаление b (уже просмотрена)	b	4	c_1
Удаление c (2-е удаление)	c	6	d_1
	b	4	d_1
	c	5	b_2
	d	5	b_2
	t	5	b_2

Окончание табл. 3.1

Удаление из кучи	Вершина	Приоритет	Предок
Удаление d (2-е удаление)	d	6	c_1
	c	6	d_1
	d	5	b_2
	t	5	b_2
	d	8	c_2
Удаление t (2-е удаление)	d	6	c_1
	c	6	d_1
	t	5	b_2
	d	8	c_2

В результате работы алгоритма получена вся необходимая информация о первых двух кратчайших (s,t)-маршрутах в графе (G,w): потенциалы (постоянные метки) вершин при их первом и втором удалении из кучи, для каждой вершины номер вершины (предок), из которой данная вершина получила свой потенциал (нижний индекс указывает порядок удаления предка из кучи и необходим для восстановления последовательности вершин кратчайшего маршрута). Информация сгруппирована в табл. 3.2.

Таблица 3.2

Метки при удалении вершин из кучи

Вершина	s	a	b	c	d	t
первое удаление вершин из кучи						
Приоритет	0	1	2	3	3	3
Предок	–	s_1	a_1	b_1	b_1	b_1
второе удаление вершин из кучи						
Приоритет	2	3	4	5	5	5
Предок	a_1	b_1	a_2	b_2	b_2	b_2

Используя данные табл. 3.2, можно определить, что первый кратчайший (s,t)-маршрут имеет длину 3 и проходит по вершинам

$$s-a-b-t,$$

второй кратчайший (s,t)-маршрут имеет длину 5 и проходит по вершинам

$$s-a-b-a-b-t.$$

3.4.5. Оптимальное k-множество непересекающихся цепей

В данном подразделе для взвешенного графа (G, w) и пары его вершин s и t будет рассмотрена задача нахождения оптимального k-множества простых рёберно-непересекающихся (простых вершинно-непересекающихся за исключением вершин s и t) (s,t)-цепей. В дальнейшем под термином «цепь» будем понимать простую цепь.

Оптимальное k-множество рёберно-непересекающихся цепей. Для взвешенного графа (G, w) и заданной пары его вершин s и t сначала рассмотрим задачу поиска оптимальной пары рёберно-непересекающихся (s,t)-цепей. На первый взгляд может показаться, что данная задача решается достаточно просто:

• найдём в графе (G, w) кратчайшую (s,t)-цепь P;

• построим модифицированный граф G', удалив из графа (G, w) рёбра цепи P и положив длину ребра $e \in E(G')$ равной $w(e)$;

• в графе G' найдём кратчайшую (s,t)-цепь P'; цепи P и P' образуют оптимальную пару рёберно-непересекающихся (s,t)-цепей в графе (G, w).

Действительно, для графа (G, w), приведённого на рис. 3.63, сначала найдём кратчайшую (s,t)-цепь P

$$P = s - a - b - t,\ w(P) = 3.$$

Затем удалим из графа (G, w) рёбра (s,t)-цепи P и в полученном

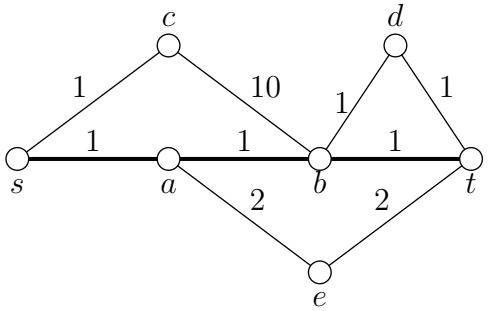


Рис. 3.63. Граф (G, w) и кратчайшая (s,t)-цепь P

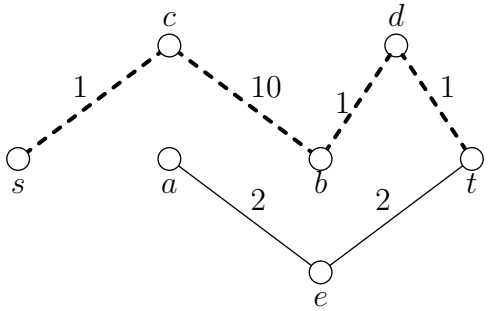


Рис. 3.64. Граф G' и кратчайшая (s,t)-цепь P'

графе G' найдём кратчайшую (s,t)-цепь P' (на рис. 3.64 рёбра цепи P' выделены пунктиром)

$$P' = s - c - b - d - t,\ w(P') = 13.$$

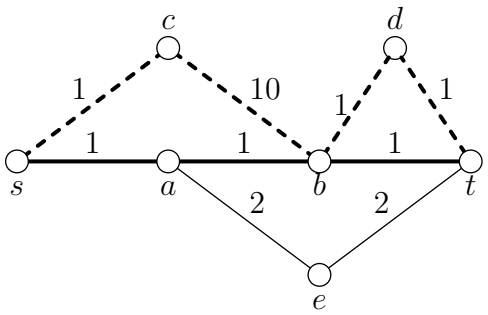


Рис. 3.65. Кратчайшая пара рёберно-непересекающихся (s,t)-цепей

Объединим (s,t)-цепи P и P'. Для графа, приведённого на рис. 3.63, две построенные (s,t)-цепи (на рис. 3.65 рёбра построенных цепей выделены жирными и пунктирными линиями соответственно) образуют оптимальную пару рёберно-непересекающихся (s,t)-цепей.

Однако для графа (G,w), приведённого на рис. 3.66, данная задача не может быть решена описанным выше способом. Сначала найдём в графе (G,w) кратчайшую (s,t)-цепь

$$P = s - a - b - t,\ w(P) = 3.$$

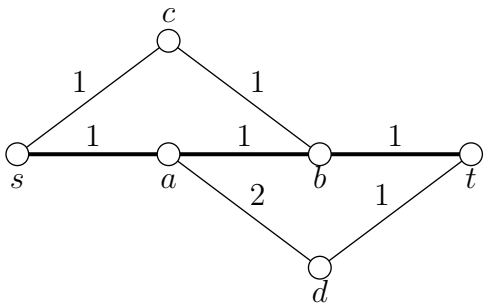


Рис. 3.66. Граф (G,w) и кратчайшая (s,t)-цепь P

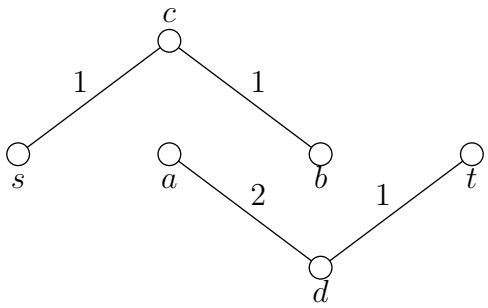


Рис. 3.67. Граф G'

Затем удалим в графе (G,w) все рёбра, входящие в цепь P. В результате выполненных действий в графе G' вершины s и t будут принадлежать разным компонентам связности (рис. 3.67). Таким образом будет сделан ошибочный вывод о том, что в графе (G,w) нет пары рёберно-непересекающихся (s,t)-цепей. Покажем, что данное утверждение является ошибочным. Действительно, на рис. 3.68 выделена оптимальная пара рёберно-непересекающихся (s,t)-цепей $s - a - d - t$ и $s - c - b - t$ (суммарная длина построенной пары цепей равна 7).

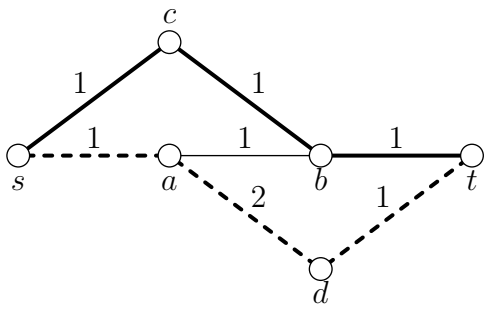


Рис. 3.68. Оптимальная пара рёберно-непересекающихся (s,t)-цепей

Рассмотрим алгоритм, приведённый в [9], который для взвешенного графа (G,w) корректно решает задачу нахождения оптимальной пары рёберно-непересекающихся (s,t)-цепей.

Алгоритм построения оптимальной пары рёберно-непересекающихся (s,t)-цепей

1. Находим в графе (G,w) кратчайшую (s,t)-цепь P.

2. По графу (G,w) и цепи P строим смешанный граф G' по следующим правилам:

- рёбра кратчайшей (s,t)-цепи P заменяем дугами; направление дуг противоположно направлению, если двигаться вдоль цепи P из вершины s в вершину t;
- если (u,v) — дуга графа G', то полагаем $w'(u,v) = -w(uv)$;
- если uv — ребро графа G', то полагаем $w'(uv) = w(uv)$.

3. Находим кратчайший (s,t)-путь Q в графе G'. По построению, в графе G' есть дуги отрицательной длины, но нет контуров отрицательной длины. Более того, отрицательные дуги графа G' составляют единую группу дуг, образующих (t,s)-путь в графе G', и если эти дуги заменить рёбрами и обратить знак их длины, то цепь, образованная из этих рёбер, будет кратчайшей (s,t)-цепью в исходном графе (G,w). Поэтому для поиска кратчайшего (s,t)-пути Q можно использовать не только алгоритм Форда–Беллмана, который за время $O(nm)$ строит кратчайший путь в графе с отрицательными длинами дуг, в предположении, что нет контуров отрицательной длины, но и описанные в подразделе 3.4.1 модифицированный алгоритм Дейкстры и модифицированный поиск в ширину.

4. Оптимальная пара рёберно-непересекающихся (s,t)-цепей получается из цепи P и пути Q по следующему правилу: заменяем дуги пути Q рёбрами и объединяем цепи P и Q, удаляя общие рёбра.

В общем случае, если нужно найти оптимальное k-множество рёберно-непересекающихся (s,t)-цепей, то нужно выполнить k шагов описанного выше алгоритма. Пусть H_2 — множество рёбер, входящих в оптимальную пару ($k=2$) рёберно-непересекающихся (s,t)-цепей. Тогда чтобы получить оптимальную тройку H_3 ($k=3$), полагаем $P=H_2$ и повторяем ещё раз алгоритм с пункта 2. Итерируя алгоритм, далее аналогичным образом мы сможем получить H_4, H_5 и т. д.

Для более простой реализации алгоритма построения оптимального k-множества можно сначала преобразовать граф (G,w) в орграф $(\overline{G},w)$ по следующему правилу: каждому ребру vu графа G в орграфе $\overline{G}$ будут соответствовать две противоположно направленные дуги (v,u) и (u,v), длина каждой из которых равна длине исходного ребра vu, т. е. $w(v,u)=w(u,v)=w(vu)$. Полагаем путь $Q=\varnothing$.

Теперь на i-м шаге алгоритма ($i=0,1,\ldots,k$) на основании пути Q орграф $\overline{G}$ перестраивается по следующему правилу: дуги пути Q заменяются противоположно направленными дугами и инвертируется знак их длины (в дальнейшем будем говорить, что дуги, которые доступны после i-го шага алгоритма, — это дуги перестроенного орграфа $\overline{G}$). В результате таких преобразований множество дуг орграфа $\overline{G}$, которые имеют отрицательную длину, соответствует множеству рёбер оптимального i-множества. Если $i=k$, то алгоритм завершает свою работу, в противном случае в орграфе $\overline{G}$ строим кратчайший (s,t)-путь Q и переходим к следующему шагу алгоритма (т. е. $i=i+1$).

Следует отметить, что в процессе работы алгоритма после i-го шага в преобразованном орграфе $\overline{G}$ ребру uv, которое принадлежит i-оптимальному множеству, соответствуют две кратные дуги (v,u) длины $w(vu)$ и $-w(vu)$ соответственно; кроме того, дуги (v,u) будут ориентирована в направлении движения из t в s вдоль цепи, которой в i-оптимальном множестве принадлежит ребро uv. В свою очередь, после i-го шага в преобразованном орграфе $\overline{G}$ ребру vu, которое не принадлежит i-оптимальному множеству, соответствуют две противоположно направленные дуги (v,u) и (u,v) положительной длины $w(vu)$.

Докажем корректность приведённого алгоритма. Для этого сначала рассмотрим операцию *наложения двух контуров* $C_1 + C_2$, которые имеют противоположные дуги (две дуги (v, u) и (u, v) называются противоположными, если $w(v, u) = -w(u, v)$). Операция наложения контуров $C_1 + C_2$ заключается в объединении всех дуг контуров C_1 и C_2 с последующим удалением противоположно направленных дуг. В результате таких преобразований для каждой вершины количество входящих дуг равно количеству выходящих дуг, а следовательно, при наложении контуров также образуются контуры. Кроме того, суммарная длина получаемых после наложения контуров равна сумме длин контуров C_1 и C_2.

Теперь, используя операцию наложения контуров, докажем, что после каждого шага алгоритма в орграфе $\overline{G}$ будут отсутствовать контуры отрицательной длины. Доказательство проведём по индукции.

База индукции — 0 шагов. В исходном орграфе $\overline{G}$ длины всех дуг неотрицательны, следовательно, все контуры имеют неотрицательную длину.

Шаг индукции: предположим, что сделано i шагов и в орграфе $\overline{G}$ нет контуров отрицательной длины. Покажем, что и на $(i + 1)$ шаге контуры отрицательной длины не появятся. Доказательство проведём от противного. Предположим, что после $(i + 1)$-го шага в орграфе $\overline{G}$ появился контур D отрицательной длины. Добавим к (s, t)-пути Q, построенном на шаге i, дугу (t, s) нулевой длины, преобразуя данный путь в контур C. Выполним процедуру наложения двух контуров $D + C$. Справедливо следующее неравенство:

$$w(Q) = w(C) > w(C) + w(D).$$

В результате наложения получим множество контуров $\{C_1, C_2, \ldots, \ldots, C_q\}$, состоящих только из дуг, которые были доступны после i-го шага алгоритма. Кроме того, среди этих контуров будет контур, которому принадлежат вершины s и t (предположим, что это контур C_{st}). Поскольку после i-го шага алгоритма контуры отрицательной длины отсутствовали, то справедливы следующие неравенства:

$$w(Q) > w(C) + w(D) = w(C_1) + w(C_2) + \ldots + w(C_{st}) \geqslant w(C_{st}).$$

Из последнего неравенства следует, что $w(Q) > w(C_{st})$, а это означает, что из множества дуг, доступных после i-го шага алгоритма, построен (s,t)-путь (из контура C_{st} удаляем дугу (t,s) нулевой длины) длина которого меньше, чем длина кратчайшего (s,t)-пути Q, построенного i-м шаге алгоритма. Противоречие. Следовательно, после $(i+1)$-го шага контуры отрицательной длины не появятся, что и требовалось доказать.

Теперь, опираясь на тот факт, что после каждого шага алгоритма в орграфе отсутствуют контуры отрицательной длины, докажем утверждение о том, что алгоритм строит оптимальное k-множество. Предположим, что алгоритм построил следующее k-множество рёберно-непересекающихся (s,t)-цепей: $\{P_1, P_2, \ldots, P_k\}$, однако существует лучшее k-множество, т. е. множество рёберно-непересекающихся (s,t)-цепей $\{P'_1, P'_2, \ldots, P'_k\}$ с меньшей суммарной длиной входящих в него рёбер:

$$w(P_1) + w(P_2) + \ldots + w(P_k) > w(P'_1) + w(P'_2) + \ldots + w(P_k)'.$$

Ориентируем все цепи P_i дугами по направлению от t к s и инвертируем знак их длины (т. е. дуги будут иметь отрицательную длину), а все цепи P'_i ориентируем дугами по направлению от s к t и оставим знак их длины положительным. Составим из пар путей P_i и P'_i контуры и выполним процедуру наложения полученных k контуров. В результате данной процедуры наложения получим множество контуров $C_1, C_2, \ldots, C_q$, состоящих только из дуг, которые были доступны после k-го шага алгоритма (все дуги пути P'_i, которые были недоступны после k-го шага алгоритма, «уничтожились» противоположно направленными дугами пути P_i). Так как после k-го шага алгоритма контуры отрицательной длины отсутствуют, то $w(C_i) \geqslant 0$ и общая суммарная длина всех рёбер после наложения равна

$$\sum_{i=1}^{k} w(P'_i) - \sum_{i=1}^{k} w(P_i) = \sum_{i=1}^{q} w(C_i) \geqslant 0.$$

Из последнего неравенства следует, что

$$\sum_{i=1}^{k} w(P'_i) \geqslant \sum_{i=1}^{k} w(P_i),$$

поэтому получаем противоречие с тем, что существует лучшее k-множество, чем то, которое построено алгоритмом.

Сейчас покажем, как на q-м шаге алгоритма восстановить q-множество рёберно-непересекающихся (s,t)-цепей. Предположим, что (s,t)-цепь Q задаётся следующей последовательностью вершин (цепь получается из кратчайшего (s,t)-пути Q, полученного на $(q-1)$-м шаге алгоритма, путём замены дуг пути Q на рёбра):

$$Q = \{s, v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_{i+z}, \ldots, t\}.$$

Обозначим через $Q(v_i, v_j)$ множество рёбер (v_i, v_j)-подцепи цепи Q.

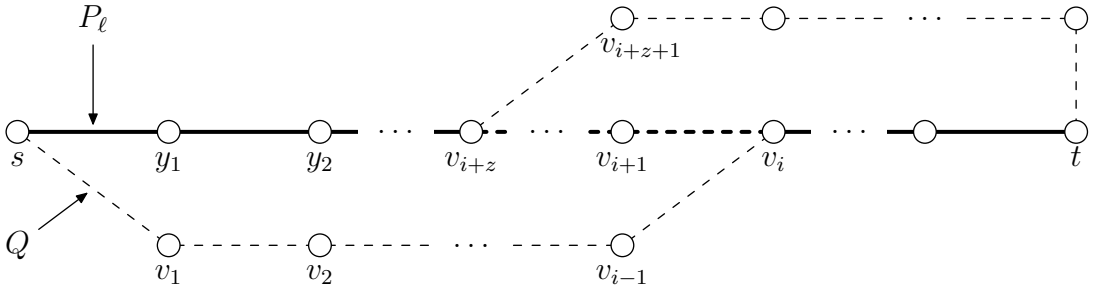


Рис. 3.69. Цепи Q и P_ℓ

Будем последовательно просматривать рёбра цепи Q, двигаясь вдоль цепи по направлению от вершины s к вершине t до тех пор, пока не встретим ребро v_iv_{i+1}, которое принадлежит некоторой цепи из множества $P^q_{s,t}$ (такому ребру соответствовала бы дуга пути Q отрицательной длины), либо не придём в конечную вершину t (на рис. 3.69 рёбра цепи Q выделены пунктирными линиями).

Предположим, что ребро v_iv_{i+1} принадлежит цепи $P_\ell \in P^{q-1}_{s,t}$

$$P_\ell = \{s, y_1, y_2, \ldots, y_m = v_{i+z}, \ldots, y_{m+z-1} = v_{i+1}y_{m+z} = v_i, y_{m+z+1}, \ldots, t\}$$

и выполняются следующие соотношения:

$$Q(v_i, v_{i+z}) = P_\ell(v_i, v_{i+z}),\ v_{i+z}v_{i+z+1} \in E(Q),\ v_{i+z}v_{i+z+1} \notin E(P_\ell),$$

т. е. вершина v_{i+z+1} цепи Q уже не лежит на цепи P_ℓ (на рис. 3.69 рёбра цепи P_ℓ выделены жирными линиями). Тогда перестраиваем цепи P_ℓ и Q по следующему правилу:

$$Q' = P_\ell(s, v_{i+z}) \cup Q(v_{i+z}, t),$$

$$P'_\ell = Q(s, v_i) \cup P_\ell(v_i, t).$$

По построению, P'_ℓ является простой (s,t)-цепью, которая не имеет общих рёбер с цепями множества $P_{s,t}^{q-1} \setminus \{P_\ell\}$, поэтому можно перестроить множество $P_{s,t}^{q-1}$:

$$P_{s,t}^{q-1} = P_{s,t}^{q-1} \setminus \{P_\ell\} \cup \{P'_\ell\}.$$

По построению, Q' является (s,t)-цепью, причём $Q'(s, v_{i+z})$ не имеет общих рёбер с цепями множества $P_{s,t}^{q-1}$ (рис. 3.70).

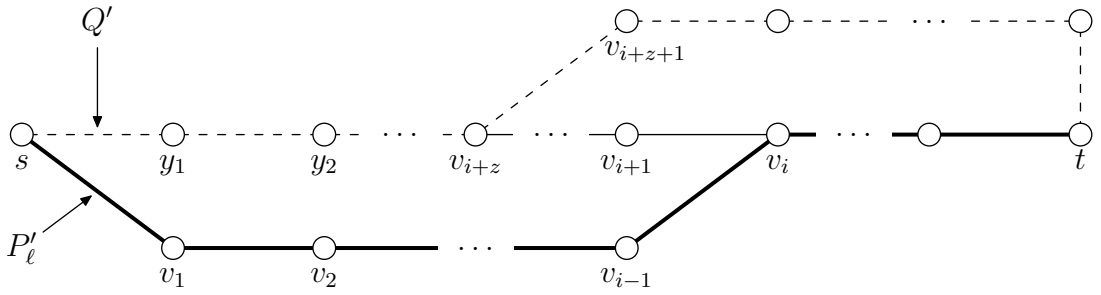


Рис. 3.70. Цепи Q' и P'_ℓ

Полагаем $Q = Q'$ и продолжаем двигаться вдоль цепи Q, анализируя в качестве текущего рёбра цепи Q ребро $v_{i+z}v_{i+z+1}$. Отметим, что если в некоторый момент времени мы опять попадём по текущему ребру $v_r v_{r+1}$ на цепь P_ℓ, то будет выполняться неравенство $r+1 > i$, а если $Q(v_r, v_{r+u}) = P_\ell(v_r, v_{r+u})$, то $r + u > i$.

В случае, когда, двигаясь по рёбрам цепи Q, мы пришли в конечную вершину t, полагаем

$$P_{s,t}^{q} = P_{s,t}^{q-1} \cup \{Q\},$$

получая при этом q-множество рёберно-непересекающихся (s,t)-цепей, построенное на основании множества $P_{s,t}^{q-1}$ и цепи Q.

Трудоёмкость приведённого алгоритма нахождения оптимального k-множества рёберно-непересекающихся (s,t)-цепей равна $\mathrm{O}(knm)$, если для поиска кратчайшего пути в орграфе (содержащем дуги отрицательной длины, но при отсутствии контуров отрицательной длины) использовать, например, алгоритм Форда – Беллмана.

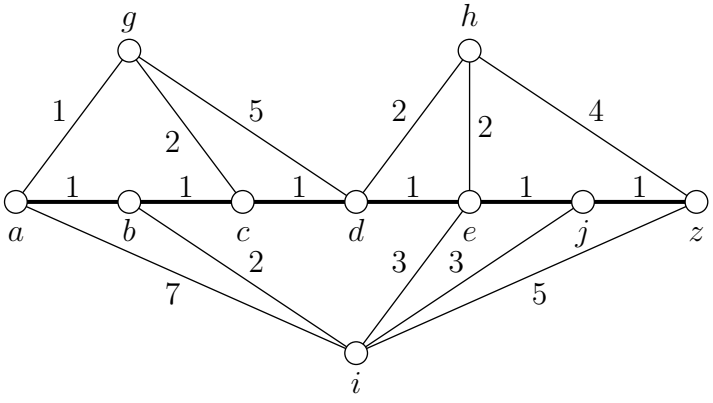


Рис. 3.71. Граф (G,w)

Пример 3.14. Для графа (G,w), изображённого на рис. 3.71, найдём оптимальное k-множество рёберно-непересекающихся (a,z)-цепей для $k=3$.

Кратчайшей (a,z)-цепью P для графа (G,w) является цепь

$$P = a-b-c-d-e-j-z,\ w(P)=6,$$

на рис. 3.71 рёбра кратчайшей цепи P выделены жирными линиями.

По графу (G,w) и (a,z)-цепи P строим смешанный граф G'. Находим в смешанном графе G' кратчайший (a,z)-путь Q (на рис. 3.72

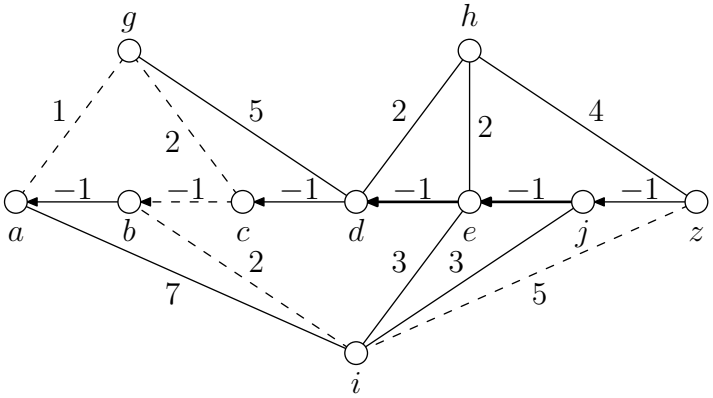


Рис. 3.72. Кратчайший путь Q в графе G'

рёбра и дуги пути Q выделены пунктирными линиями)

$$Q = a - g - c \to b - i - z,\ w(Q) = 9.$$

Заменяем дуги пути Q рёбрами, объединяем рёбра цепей P и Q, удаляя общее ребро bc, и получаем оптимальную пару рёберно-непересекающихся (a, z)-цепей (на рис. 3.73 рёбра оптимальной пары цепей выделены жирными линиями)

$$P_1 = a - g - c - d - e - j - z,$$
$$P_2 = a - b - i - z,$$
$$w(P_1) + w(P_2) = 15.$$

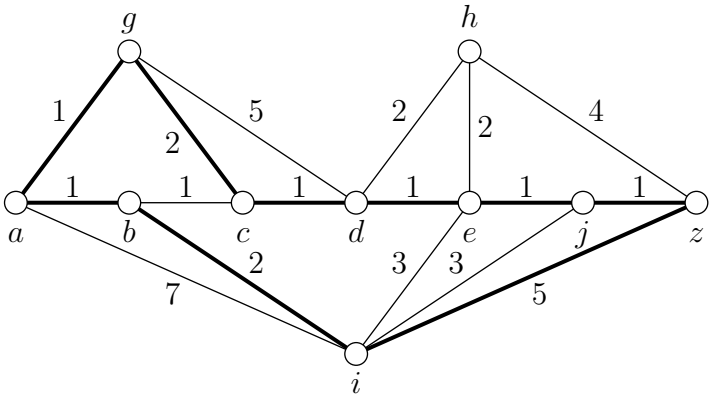


Рис. 3.73. Оптимальная пара рёберно-непересекающихся (a, z)-цепей

Полагаем $P = P_1 \cup P_2$ и строим оптимальное k-множество ($k = 3$) рёберно-непересекающихся (a, z)-цепей. Для этого по взвешенному графу (G, w) и множеству P строим смешанный граф G'' (рис. 3.74).

Находим в графе G'' кратчайший (a, z)-путь Q (на рис. 3.74 рёбра и дуги пути Q выделены пунктирными линиями)

$$Q = a - i - j \to e \to d - h - z,\ w(Q) = 14.$$

Заменяем дуги пути Q рёбрами, объединяем рёбра из множеств P и Q, удаляя общие рёбра de и ej, и получаем для $k = 3$ оптимальное k-множество рёберно-непересекающихся (a, z)-цепей (на рис. 3.75 цепи

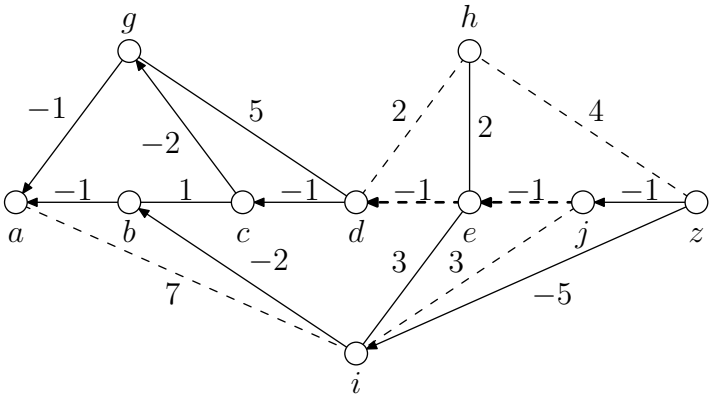


Рис. 3.74. Кратчайший маршрут Q
в смешанном графе G''

построенного оптимального множества выделены различными жирными линиями):

$$P_1 = a - g - c - d - h - z,$$
$$P_2 = a - b - i - j - z,$$
$$P_3 = a - i - z.$$

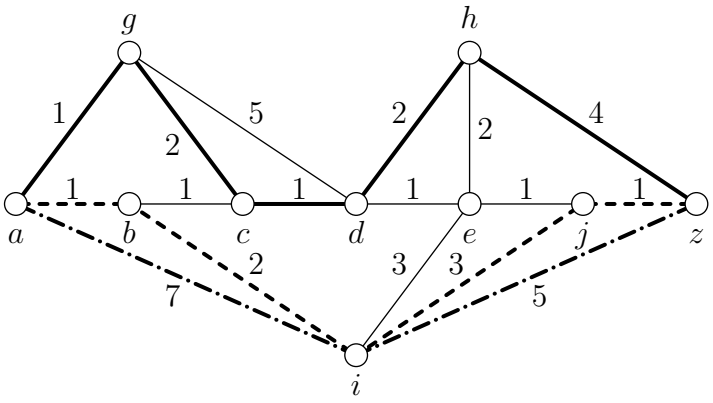


Рис. 3.75. Оптимальное множество из трёх
рёберно-непересекающихся (a, z)-цепей

Длина оптимального множества равна

$$w(P_1) + w(P_2) + w(P_3) = 29.$$

Ещё одним из алгоритмов, который во взвешенном графе (G, w) корректно находит оптимальную пару рёберно-непересекающихся (s, t)-

цепей, является *метод Сурбалле* (*Suurballe method*) [9]. В этом алгоритме граф (G, w) сначала преобразуется в орграф (G', w') по следующим правилам:

• каждое ребро uv графа (G, w) заменяется двумя противоположно направленными дугами (u, v) и (v, u);

• длины дуг (x, y) определяются по правилу

$$w'(x, y) = L(x) + w(xy) - L(y),$$

где $L(x)$ — длина кратчайшей простой (s, x)-цепи в графе (G, w).

Поскольку в орграфе (G', w') длина любого (s, t)-пути уменьшается на величину $L(t)$, то задача нахождения кратчайшей простой (s, t)-цепи в графе (G, w) и задача нахождения кратчайшего (s, t)-пути в орграфе (G', w') эквивалентны. Сначала найдём кратчайшую (s, t)-цепь P в графе (G, w), а затем определим кратчайший (s, t)-путь Q в орграфе (G'', w'), который получается из орграфа (G', w'), если удалить из него дуги маршрута P, которые направлены к источнику s, и изменить направление оставшихся дуг маршрута P на противоположное (так как длины оставшихся дуг маршрута P равны 0, то обращать знак длины этих дуг в орграфе G'' не надо). Поскольку в орграфе (G'', w') длины всех дуг остались неотрицательными, то для поиска кратчайшего пути Q можно использовать, например, базовый алгоритм Дейкстры. После того как будет построен кратчайший (s, t)-путь Q, делаем обратное преобразование орграфа (G'', w') к графу (G, w) и удаляем все рёбра, которые попали в обе цепи P и Q либо ни в одну из них. Оставшиеся рёбра образуют требуемую оптимальную пару рёберно-непересекающихся (s, t)-цепей в графе (G, w).

Трудоёмкость метода Сурбалле нахождения оптимальной пары рёберно-непересекающихся (s, t)-цепей в графе (G, w) равна:

• $O(n^2)$, если в алгоритме использовать базовый алгоритм Дейкстры;

• $O(m \log n)$, если в алгоритме использовать реализацию алгоритма Дейкстры с применением структуры данных бинарная куча [2];

• $O(m + n \log n)$, если в алгоритме использовать реализацию алгоритма Дейкстры с применением структуры данных Фибоначчиева куча [2].

Пример 3.15. Найдём оптимальную пару рёберно-непересекающихся (s,t)-цепей в графе (G,w), изображённом на рис. 3.76, используя метод Сурбалле.

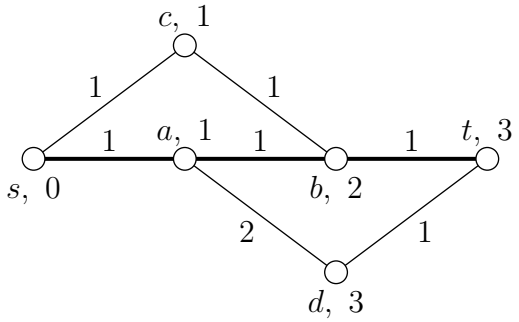


Рис. 3.76. Кратчайшая (s,t)-цепь P в графе (G,w)

Сначала определим длины кратчайших цепей, соединяющих вершину s со всеми достижимыми из неё вершинами (на рис. 3.76 после метки вершины v через запятую указана длина кратчайшей (s,v)-цепи). Кратчайшая (s,t)-цепь P задаётся следующей последовательностью вершин (на рис. 3.76 рёбра цепи P выделены жирными линиями):

$$P = s - a - b - t,\ w(P) = 3.$$

Построим орграф (G'', w') и найдём в нём кратчайший (s,t)-путь

$$Q = s \to c \to b \to a \to d \to t,\ w(Q) = 1,$$

(на рис. 3.77 кратчайший путь Q выделен пунктирными линиями).

Делаем обратное преобразование орграфа (G'', w') к графу (G,w) и удаляем все рёбра, которые попали в обе цепи P и Q (удаляется ребро ab) либо ни в одну из них. Оставшиеся в графе (G,w) рёбра обра-

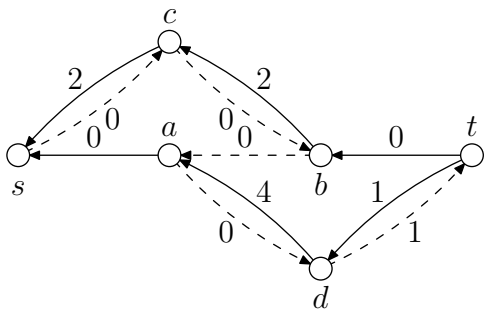


Рис. 3.77. Взвешенный орграф (G'', w') и кратчайший путь Q в нём

зуют требуемую оптимальную пару рёберно-непересекающихся (s,t)-цепей:

$$P_1 = s - c - b - t,$$
$$P_2 = s - a - d - t,$$
$$w(P_1) + w(P_2) = w(P) + w(Q) + L(t).$$

На рис. 3.78 оптимальная пара рёберно-непересекающихся (s,t)-цепей выделена пунктирными и жирными линиями соответственно.

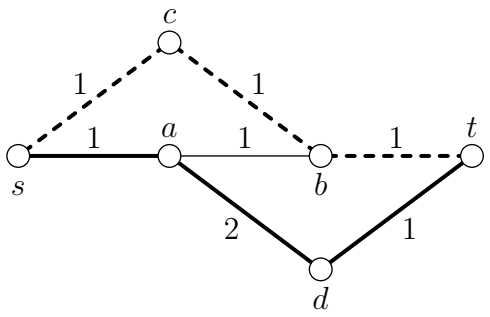


Рис. 3.78. Оптимальная пара рёберно-непересекающихся (s,t)-цепей

Для того чтобы восстановить (s,t)-цепи, принадлежащие построенному оптимальному k-множеству, можно выполнить k раз поиск в глубину из вершины s по рёбрам, принадлежащим оптимальному k-множеству. Трудоёмкость процедуры восстановления k маршрутов равна $\mathrm{O}(k(n+m))$.

Оптимальное k-множество вершинно-непересекающихся цепей. Для взвешенного графа (G,w) и заданной пары вершин s и t опишем алгоритм построения оптимальной пары ($k=2$) вершинно-непересекающихся (s,t)-цепей (общими вершинами для этой пары цепей являются только вершины s и t) и проиллюстрируем его работу для взвешенного графа (G,w), приведённого на рис. 3.79.

Алгоритм нахождения оптимальной пары вершинно-непересекающихся (s,t)-цепей

1. Найти кратчайшую (s,t)-цепь P в графе (G,w) (на рис. 3.79 рёбра кратчайшей (s,t)-цепи выделены жирными линиями).

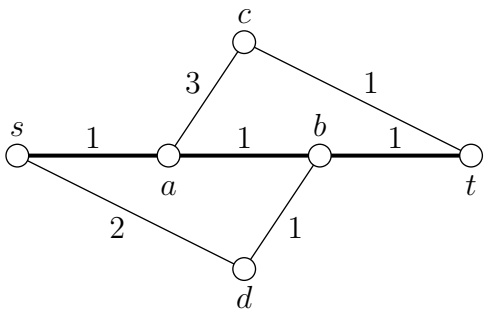


Рис. 3.79. Взвешенный граф (G, w)

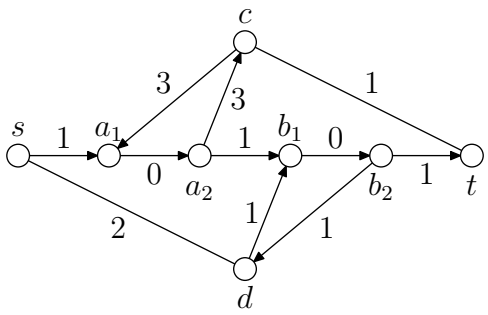


Рис. 3.80. Граф (G', w')

2. Построить смешанный взвешенный граф (G', w') по следующим правилам:

2.1. заменить каждое ребро цепи P дугой, направленной в сторону конечной вершины t;

2.2. выполнить раздвоение каждой промежуточной вершины v цепи P на две вершины v_1 и v_2, соединённые дугой нулевой длины, направленной в сторону конечной вершины t. Каждое ребро, не входящее в цепь P, но инцидентное какой-либо вершине v цепи P (исключая вершины s и t), заменить на две противоположно направленные дуги (длины, равной исходной длине ребра): одна из них должна входить в v_1, а другая — исходить из v_2 таким образом, что эти три дуги (входящая в v_1, идущая из в v_1 в v_2 и исходящая из v_2) образуют контур (рис. 3.80);

2.3. обратить направление дуг, входящих в цепь P, и заменить их длину на противоположную по знаку (рис. 3.81).

3. Найти кратчайший (s, t)-путь Q в графе (G', w') (на рис. 3.81 дуги пути Q, заданного последовательностью вершин

$$s - d \to b_1 \to a_2 \to c \to t,$$

выделены пунктирными линиями).

4. Для получения оптимальной пары вершинно-непересекающихся (s, t)-цепей выполнить обратное преобразование графа (G', w') к графу (G, w) (сжать раздвоенные вершины и заменить дуги рёбрами), удалить из графа (G, w) все рёбра, которые попали в обе (s, t)-цепи P и Q либо ни в одну из них (рис. 3.82).

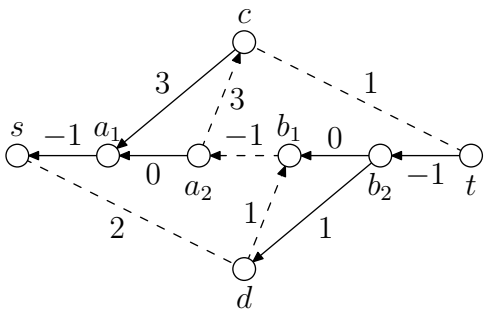


Рис. 3.81. Кратчайший путь Q в графе (G', w')

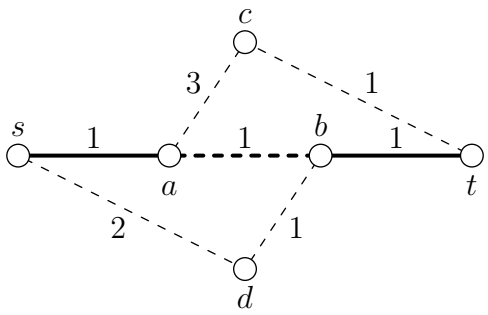


Рис. 3.82. Удаление общего ребра a, b

На рис. 3.83 рёбра оптимальной пары вершинно-непересекающихся (s,t)-цепей

$$P_1 = s - d - b - t$$

и

$$P_2 = s - a - c - t$$

выделены жирными и пунктирными линиями соответственно.

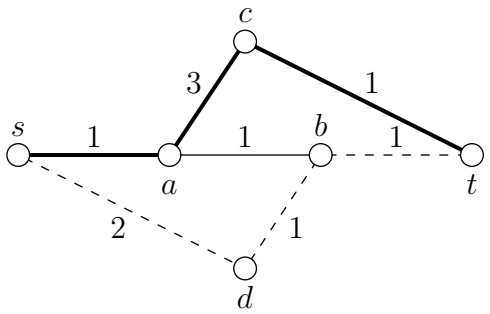


Рис. 3.83. Кратчайшая пара вершинно-непересекающихся (s,t)-цепей

Обобщение алгоритма на случай $k > 2$ аналогично обобщению на этот случай алгоритма построения оптимального k-множества рёберно-непересекающихся (s,t)-цепей. Пусть построено оптимальное $(\ell-1)$-множество вершинно-непересекающихся (s,t)-цепей $(\ell - 1 < k)$, тогда полагаем

$$P = P_1 \cup P_2 \cup \ldots \cup P_{\ell-1}$$

и повторяем приведённый выше алгоритм, находя при этом оптимальное ℓ-множество вершинно-непересекающихся (s,t)-цепей (либо делаем вывод о том, что оптимального ℓ-множества не существует).

Трудоёмкость алгоритма нахождения оптимального k-множества вершинно-непересекающихся (s,t)-цепей равна $O(knm)$, если для поиска кратчайшего (s,t)-пути в смешанном графе (G', w') использовать, например, алгоритм Форда – Беллмана.

Для того чтобы восстановить простые (s,t)-цепи, принадлежащие построенному оптимальному k-множеству, можно запустить k раз поиск в глубину из вершины s по рёбрам, принадлежащим оптимальному k-множеству. Трудоёмкость процедуры восстановления всех маршрутов из оптимального k-множества равна $O(k(n+m))$.

3.5. ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

1. Скрудж Мак-Дак. Скрудж Мак-Дак решил сделать прибор для управления самолетом. Как известно, положение штурвала зависит от состояния входных датчиков, но эта функция довольно сложна. Его механик спроектировал устройство, вычисляющее функцию за несколько этапов с использованием промежуточной памяти и вспомогательных функций. Для подсчёта значений каждой из функций требуется, чтобы в ячейках памяти уже находились полученные ранее необходимые параметры (которые являются значениями функций). Вычисление функции без параметров может производиться в любое время. После получения значения фукнции ячейки могут быть использованы повторно (хотя бы для записи нового результата). Структура вызова функций такова, что каждая функция вычисляется не более одного раза и любой параметр используется не более одного раза. Любой параметр есть имя функции. Так как Скрудж не хочет тратить лишних денег на микросхемы, он поставил задачу минимизировать память прибора. По заданной структуре вызовов функций необходимо определить минимально возможный размер памяти прибора и указать последовательность вычисления функций.

2. Сборка прибора. Прибор спроектирован таким образом, что его будут собирать из отдельных узлов, причём каждый узел уникален. В свою очередь, сами узлы могут требовать предварительной сборки. Для сборки каждого узла необходимо, чтобы все узлы, комплектую-

щие его, были уже собраны. Узлы, не требующие сборки, обязательно тестируются на работоспособность. Собираемые узлы тестировать не требуется. На сборку одного узла или его тестирование тратится один день. Готовый узел должен быть помещён на склад и может быть взят со склада только тогда, когда он необходим для сборки очередного узла или самого прибора. Хранение узла на складе в течение одного дня требует оплаты в размере одной условной денежной единицы. Необходимо организовать сборку таким образом, чтобы плата за аренду склада была минимальной.

3. Станки. В конвейер связаны один за другим N различных станков. Имеется N рабочих. Задана матрица C размера $N \times N$, где элемент матрицы $c_{i,j}$ задаёт производительность i-го рабочего на j-м станке. Необходимо определить, каким должно быть распределение рабочих по станкам (каждый рабочий может быть назначен только на один станок; на каждом станке может работать только один рабочий), чтобы производительность всего конвейера была максимальной. Производительность конвейера при некотором распределении рабочих по станкам равна минимальной из производительностей рабочих на назначенных им на конвейере станках. Если решение не единственно, вывести решение, первое в лексикографическом порядке среди всех решений.

4. Открытки и конверты. Имеется N прямоугольных конвертов и N прямоугольных открыток различных размеров. Необходимо определить, можно ли разложить все открытки по конвертам так, чтобы в каждом конверте было по одной открытке. Открытки нельзя складывать, сгибать и т. п., но можно помещать в конверт под углом. Например, открытка с размерами сторон $5 : 1$ помещается в конверты с размерами $5 : 1$, $6 : 3$, $4{,}3 : 4{,}3$, но не входит в конверты с размерами $4 : 1$, $10 : 0{,}5$, $4{,}2 : 4{,}2$.

5. Янка. Янка положил на стол N ($1 < N \leqslant 500$) выпуклых K-гранников ($1 < K \leqslant 100$) и N различных типов наклеек. Ночью кто-то наклеил наклейки на грани, по одной на грань (на одном и том же многограннике могло оказаться несколько наклеек одного типа). Янке необходимо расставить многогранники так, чтобы наклейка каждого типа была видна ровно $K - 1$ раз.

6. Прогулка. Хозяин вышел на прогулку с собакой. Известно, что путь хозяина представляет собой ломаную линию, координаты вершин ломаной (x_i, y_i) для $i = 1, \ldots, n$. Точка (x_1, y_1) — начальная точка ломаной линии, а точка (x_n, y_n) — конечная точка ломаной. Хозяин двигается во время прогулки от стартовой точки, далее — по отрезкам ломаной линии и заканчивает путь в конечной точке ломаной. У собаки есть свои любимые места с координатами $(\hat{x}_j, \hat{y}_j)$ для $j = 1, \ldots, m$, которые она хотела бы посетить. В то время пока хозяин проходит один отрезок ломаной линии, собака может посетить только одно из своих любимых мест. В начальной и конечной координате каждого отрезка ломаной собака обязана подбежать к хозяину. Известно, что скорость собаки в 2 раза выше скорости хозяина. Необходимо определить, какое наибольшее количество своих любимых мест и в какой последовательности сможет посетить собака за время прогулки.

7. Отрезки. Пусть на плоскости с Евклидовой метрикой задано n красных и n синих точек (красные точки нумеруются числами от 1 до n, а синие — от $n+1$ до $2n$). Имеется n отрезков, соединяющих эти точки таким образом, что каждый отрезок соединяет точки различного цвета и каждая точка является концевой точкой ровно одного отрезка. Необходимо определить, является ли заданное соединение минимальным по длине (сумма длин всех отрезков) среди всех возможных соединений, удовлетворяющих заданным свойствам.

8. Железные и шоссейные дороги. Города пронумерованы числами от 1 до n, где n — натуральное число. Некоторые из них соединены двусторонними дорогами, пересекающимися только в городах. Дороги бывают двух типов — шоссейные и железные. Для каждой дороги известна базовая стоимость проезда, зависящая от набора дорог, по которым вы проезжаете, и способа проезда. Так, если вы подъехали к городу C по шоссейной (железной) дороге $X - C$ и хотите ехать дальше по дороге $C - Y$ того же типа, то нужно оплатить только базовую стоимость проезда по дороге $C - Y$. Если тип дороги $C - Y$ отличен от типа дороги $X - C$, то вы должны оплатить базовую стоимость проезда по дороге $C - Y$ плюс 10 % базовой стоимости проезда по этой дороге (страховой взнос). При выезде из города A страховой

взнос платится всегда. Необходимо определить самый дешёвый маршрут проезда из города A в город B в виде последовательности городов, а также вычислить стоимость проезда по этому маршруту.

9. Заправочные станции. Существует n городов, соединённых двусторонними дорогами с правосторонним движением. Для каждой дороги задана её протяженность. Машина может поворачивать (изменять направление движения) только в городах. Бак машины вмещает z литров бензина, а двигатель расходует x литров на один километр. В некоторых городах находятся заправочные станции. Для каждой заправочной станции определена своя цена за 1 литр бензина. Машина сможет заправиться только в том случае, если её бак заполнен менее чем наполовину. Необходимо определить самый дешёвый маршрут из города A в город B.

10. Пирамида Хеопса. Внутри пирамиды Хеопса есть n комнат, в которых установлено $2m$ модулей, составляющих m устройств. Каждое устройство состоит из двух модулей, располагающихся в разных комнатах, и предназначено для перемещения между парой комнат, в которых они установлены. Перемещение происходит за 0,5 единицы времени. В начальный момент времени модули всех устройств переходят в подготовительный режим. Каждый из модулей имеет свой некоторый целочисленный период времени, в течение которого он находится в подготовительном режиме. По истечении этого времени модуль мгновенно включается, после чего опять переходит в подготовительный режим. Устройством можно воспользоваться только в тот момент, когда одновременно включаются оба его модуля. Преступник сумел проникнуть в гробницу фараона. Обследовав её, он включил устройства и собрался уходить, но в этот момент проснулся охранник. Теперь преступнику необходимо как можно быстрее попасть из комнаты номер 1 в комнату с номером n, в которой находится выход из пирамиды. При этом перемещаться из комнаты в комнату он может только при помощи устройств, так как проснувшийся охранник закрыл все двери в комнатах пирамиды. Необходимо разработать алгоритм, который по описанию расположения устройств и их характеристик получает значение оптимального времени и последовательность устройств, предна-

значенных для того, чтобы попасть из комнаты номер 1 в комнату номер n за это время.

11. Таксист. Имеется n городов, связанных m дорогами (нумерация дорог и городов начинается с 1). Движение по дорогам осуществляется только в одном направлении, и дороги пересекаются только в городах. Известна длина каждой дороги. Необходимо найти все дороги, по которым потенциально может проехать таксист таким образом, чтобы длина его маршрута отличалась не более чем на величину k от длины минимального маршрута из города 1 в город n.

12. Сеть дорог. Сеть дорог определяется следующим образом. Имеется n перекрёстков и k дорог, связывающих перекрёстки. Каждая дорога определяется тройкой чисел: двумя номерами перекрёстков и временем, требующимся на проезд по этой дороге. Необходимо найти кратчайший маршрут машины от перекрёстка i до перекрёстка j, содержащий не более чем i перекрёстков, с учётом того что на перекрёстке машина должна совершить остановку на время, равное числу пересекающихся дорог. Движение по дороге возможно в обоих направлениях (в начальном и конечном городах кратчайшего пути ожидание не требуется).

13. Светофоры. Дорожное движение в городе Дингилвилле устроено необычным образом. В городе есть перекрёстки и дороги, которыми перекрёстки связаны между собой. Два любых перекрёстка могут быть связаны не более чем одной дорогой. Не существует дорог, соединяющих один и тот же перекрёсток с самим собой. Время проезда по дороге в обоих направлениях одинаково. На каждом перекрёстке находится один светофор, который в любой момент времени может быть либо голубым, либо красным. Цвет светофора изменяется периодически: в течение некоторого интервала времени он голубой, а затем, в течение некоторого другого интервала, — красный. Движение по дороге между любыми двумя перекрёстками разрешено тогда и только тогда, когда светофоры на обоих перекрёстках этой дороги имеют один и тот же цвет в момент въезда на эту дорогу. Если транспортное средство прибывает на перекрёсток в момент переключения светофора, то его движение будет определяться новым цветом светофора. Транспорт-

ные средства могут находиться в состоянии ожидания на перекрёстках. У вас есть карта города, которая показывает:

- время прохождения каждой дороги (целые числа);
- длительность горения каждого цвета для каждого светофора (целые числа);
- начальный цвет и оставшееся время горения этого цвета (целые числа) для каждого светофора.

Необходимо определить путь между двумя заданными перекрёстками, позволяющий транспортному средству проехать от начального перекрёстка к конечному за минимальное время с момента старта. Если существуют несколько таких путей, то нужно вывести только один из них.

14. Инкассаторы. Когда Макс Крейзи закончил финансовый колледж, он стал управляющим городского банка. Уже с первых дней работы Макс столкнулся с одной неразрешимой для него проблемой. В стране, где живет Макс, есть n городов. Некоторые из них связаны двусторонними дорогами, пересекающимися только в городах. Раз в месяц инкассаторы Макса должны доставлять деньги в k сберкасс, которые находятся в различных городах. Пока банк Макса небогат и имеет всего одну машину для перевозки денег. Необходимо составить маршрут, начинающийся в городе ℓ (в котором располагается банк Макса), проходящий по всем k городам, где находятся нужные сберкассы, и заканчивающийся также в городе ℓ. Маршрут должен иметь минимальную длину (длиной маршрута назовём сумму длин всех дорог, входящих в него).

15. Надёжная сеть. Акула хочет быть в курсе всех дел Квадратного Бизнесмена. Для этого он дал Фитцджеральду новое задание: связать в сеть компьютеры Акулы и Квадратного Бизнесмена, используя для этого промежуточные компьютеры, стоящие в цехах «Сыр Индастриз». Также Акуле известно, что Квадратный Бизнесмен обладает достаточным могуществом, чтобы отключить от сети одновременно до k промежуточных компьютеров, но на отключение $(k+1)$-го компьютера его власти уже не хватает. Квадратный Бизнесмен никогда не выключает свой компьютер и не может отключить компьютер Акулы. Акула

хочет, чтобы сеть была надёжной, т. е. чтобы связь между компьютерами Акулы и Квадратного Бизнесмена не пропадала ни при каких действиях последнего. Все компьютеры пронумерованы уникальными натуральными числами от 1 до n. Компьютер Акулы имеет номер A, а компьютер Квадратного Бизнесмена — номер B. Фитц уже определил, какие пары компьютеров можно связать между собой непосредственно и сколько метров провода понадобится для этого. Необходимо посчитать, можно ли построить надёжную сеть. Если это возможно, то узнать наименьшую суммарную длину проводов (в метрах), необходимую для этого, если нет — определить минимальное количество компьютеров, которое нужно отключить Квадратному Бизнесмену, чтобы Акула не мог быть в курсе его дел.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. Алгоритмы: построение и анализ / Т. Кормен [и др.]. М., 2005.

2. *Котов В. М.*, *Соболевская Е. П.*, *Толстиков А. А.* Алгоритмы и структуры данных : учеб. пособие. Минск, 2011.

3. *Препарата Ф.*, *Шеймос М.* Вычислительная геометрия: Введение : пер. с англ. М., 1989.

4. *Хопкрофт Дж. Е.*, *Тарьян Р. Е.* Изоморфизм планарных графов // Кибернетический сборник. Новая серия. М., 1975. Вып. 12. С. 39–61.

5. *Гэри М.*, *Джонсон Д.* Вычислительные машины и труднорешаемые задачи. М., 1982.

6. *Ловас Л.*, *Пламмер М.* Прикладные задачи теории графов. Теория паросочетаний в математике, физике, химии. М., 1998.

7. Лекции по теории графов / В. А. Емеличев [и др.]. М., 1990.

8. *Pióro M.*, *Medhi D.* Routing, Flow, and Capacity Design in Communication and Computer Networks. Kansas City, 2004.

9. *Bhandari R.* Survivable networks: Algorithms for Diverse Routing. Norwell, 1999.

Учебное издание

Иржавский Павел Александрович
Котов Владимир Михайлович
Лобанов Алексей Юрьевич и др.

ТЕОРИЯ АЛГОРИТМОВ

Учебное пособие

Редактор *А. Г. Терехова*
Художник обложки *Т. Ю. Таран*
Технический редактор *Т. К. Раманович*
Компьютерная вёрстка *П. А. Иржавского*
Корректор *С. А. Бондаренко*

Подписано в печать 06.11.2013. Формат 60×84/16. Бумага офсетная.
Ризография. Усл. печ. л. 9,30. Уч.-изд. л. 8,9
Тираж 100 экз. Заказ ...

Белорусский государственный университет.
ЛИ № 02330/0494425 от 08.04.2009.
Пр. Независимости, 4, 220030, Минск.

Республиканское унитарное предприятие
«Издательский центр Белорусского государственного университета».
ЛП № 02330/0494178 от 03.04.2009.
Ул. Красноармейская, 6, 220030, Минск.