Алгоритмы и структуры данных III

СПбГУ, третий семестр, 2020 год

Алексей Гордеев

Оглавление

Быстрая длинная арифметика

1. Быстрое умножение

Умножение чисел сведём к умножению многочленов: пусть даны два числа, записанные в r-ичной системе счисления —

$$a = \sum_{i=0}^{k} a_i r^i,\ b = \sum_{i=0}^{m} b_i r^i,$$

где $0 \leqslant a_i, b_i < r$, тогда рассмотрим многочлены

$$P(x) = \sum_{i=0}^{k} a_i x^i,\ Q(x) = \sum_{i=0}^{m} b_i x^i.$$

Произведение ab — это значение произведения многочленов $P \cdot Q$ в точке r:

$$ab = (P \cdot Q)(r) = \sum_{l=0}^{k+m} \left(\sum_{i+j=l} a_i b_j \right) \cdot r^l.$$

Тогда достаточно найти коэффициенты произведения $P \cdot Q$, после чего сделать переносы в разрядах, чтобы получить запись ab в r-ичной системе счисления.

Осталось научиться быстро перемножать многочлены. Дальше нам придётся работать с вещественными числами (даже если коэффициенты многочленов изначально целые), но пока для удобства будем считать, что все арифметические операции над коэффициентами можно осуществлять без погрешности за $O(1)$.

1.1 Альтернативное представление многочленов

Из курса алгебры вы знаете, что любой многочлен степени не больше d однозначно задаётся своими значениями в любых $d+1$ различных точках. Зафиксируем произвольные попарно различные точки $x_0, x_1, \ldots, x_d$, тогда любой многочлен $P(x)$, $\deg(P) \leqslant d$, можно задать двумя способами:

- коэффициентами $a_0, \ldots, a_d$: $P(x) = a_0 + a_1 x + \cdots + a_d x^d$;
- значениями в точках: $y_0 = P(x_0), \ldots, y_d = P(x_d)$.

Два многочлена, заданных вторым способом, легко перемножить за линейное время (если степень их произведения не превосходит d, то есть если произведение тоже однозначно задаётся таким способом): $(P \cdot Q)(x_i) = P(x_i) \cdot Q(x_i)$.

Обычно мы всё-таки работаем с многочленами, заданными набором коэффициентов, поэтому перемножать многочлены мы будем в три шага: пусть даны многочлены $P(x)$, $Q(x)$, $\deg(P) = k$, $\deg(Q) = m$, заданные наборами коэффициентов $\{a_i\}_{i=0}^{k}$ и $\{b_i\}_{i=0}^{m}$. Мы выберем $n \geqslant k + m + 1$ различных точек $x_0, \ldots, x_{n-1}$, после чего

1. Вычислим значения многочленов в точках: $P(x_0), \ldots, P(x_{n-1})$ и $Q(x_0), \ldots, Q(x_{n-1})$;
2. Вычислим значения произведения многочленов в точках:

$$(P \cdot Q)(x_0) = P(x_0) \cdot Q(x_0), \ldots, (P \cdot Q)(x_{n-1}) = P(x_{n-1}) \cdot Q(x_{n-1});$$

3. Восстановим коэффициенты $\{c_i\}_{i=0}^{n-1}$ произведения $(P \cdot Q)(x) = c_0 + \cdots + c_{n-1}x^{n-1}$ по значениям в точках.

Пока мы умеем осуществлять быстро лишь второй шаг. Самый простой способ выполнить первый шаг — вычислить значение многочлена в каждой точке отдельно, суммарно потратив $\Theta(n^2)$ времени. Оказывается, если подобрать точки аккуратно, то одни и те же вычисления можно переиспользовать для разных точек; это позволяет осуществлять первый этап за $\Theta(n \log n)$.

1.2 Быстрое преобразование Фурье

Итак, мы хотим вычислить значения многочлена $P(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ в n различных точках, при этом набор точек мы можем взять какой угодно. Для удобства будем считать, что n — степень двойки (увеличим n до ближайшей степени двойки и, если надо, добавим нужное количество нулевых коэффициентов a_j, оценка времени работы не изменится, так как n увеличилось не более, чем в два раза).

Попробуем воспользоваться методом "разделяй и властвуй": пусть n точек разбиты на пары $\pm x_0, \ldots, \pm x_{n/2-1}$. Обозначим

$$P_0(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{\frac{n-2}{2}},$$

$$P_1(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{\frac{n-2}{2}},$$

тогда

$$P(x) = P_0(x^2) + x \cdot P_1(x^2).$$

Таким образом, достаточно вычислить значения многочленов P_0, P_1 степени $\frac{n}{2}$ в $\frac{n}{2}$ точках $x_0^2, \ldots, x_{n/2-1}^2$, после чего значения многочлена P в исходных точках можно будет восстановить за линейное время:

$$P(x_j) = P_0(x_j^2) + x_j \cdot P_1(x_j^2),\ P(-x_j) = P_0(x_j^2) - x_j \cdot P_1(x_j^2).$$

Проблема в том, что для любых вещественных x_j их квадраты x_j^2 будут неотрицательны, поэтому рекурсивно применить тот же трюк не получится. Решение проблемы — вычислять значения многочлена не в вещественных, а в комплексных точках. Возьмём в качестве точек комплексные корни из единицы n-й степени (здесь и далее $i = \sqrt{-1}$):

$$x_j = \omega_n^j, 0 \leqslant j < n, \text{ где } \omega_n = e^{2\pi i/n} = \cos(2\pi/n) + i\sin(2\pi/n).$$

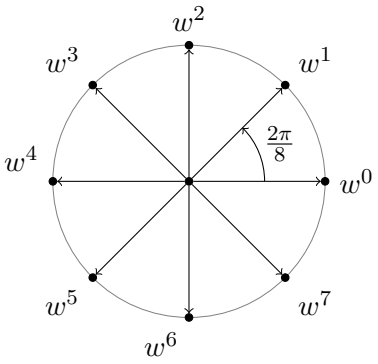

Рис. 1.1: Комплексные корни 8-й степени из единицы

Если n чётно, то для $0 \leqslant j < \frac{n}{2}$ верно

$$x_{j+n/2} = e^{2\pi i(j+n/2)/n} = e^{2\pi ij/n} \cdot e^{\pi i} = -e^{2\pi ij/n} = -x_j,$$

то есть точки разбиваются на пары противоположных по знаку. При этом для любого $0 \leqslant j < \frac{n}{2}$

$$x_j^2 = e^{2\pi i \cdot 2j/n} = e^{2\pi ij/(n/2)},$$

то есть x_j^2 являются корнями из единицы степени $\frac{n}{2}$. Значит, значения в таких точках можно продолжать рекурсивно вычислять тем же способом. Получаем рекуррентное соотношение $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$, откуда $T(n) = \Theta(n \log n)$.

```
# в результате выполнения функции на место коэффициентов записываются значения в точках
FFT(a, n):
  if n == 1:
    return # значение в точке совпадает с единственным коэффициентом
  a --> aOdd, aEven # разбиваем коэффициенты на чётные и нечётные
  FFT(aEven, n / 2), FFT(aOdd, n / 2)
  w = exp(2 * pi * i / n) # первообразный корень из единицы n-й степени
  x = 1
  for j = 0..(n - 1):
    a[j] = aEven[j % (n / 2)] + x * aOdd[j % (n / 2)]
    x *= w
```

Получившийся алгоритм по n-элементной последовательности чисел $a_0, \ldots, a_{n-1}$ вычисляет последовательность $A_0, \ldots, A_{n-1}$ по правилу

$$A_j = \sum_{k=0}^{n-1} a_k \cdot e^{2\pi ijk/n}.$$

Последовательность $A_0, \ldots, A_{n-1}$ называют *дискретным преобразованием Фурье (discrete Fourier transform, DFT)* последовательности $a_0, \ldots, a_{n-1}$. Изученный нами алгоритм, соответственно, называют *быстрым преобразованием Фурье (fast Fourier transform, FFT)*.

R Вообще говоря, стандартное определение дискретного преобразования Фурье получится, если вместо $e^{2\pi ij/n}$ подставить $e^{-2\pi ij/n}$, но сути это не меняет ($e^{2\pi ij/n}$ и $e^{-2\pi ij/n}$ при $0 \leqslant j < n$ пробегают один и тот же набор точек).

Помимо умножения многочленов и чисел, дискретное преобразование Фурье играет важную роль в цифровой обработке сигналов и применяется, например, в алгоритмах сжатия изображений и звуковых файлов.

1.3 Обратное преобразование

Осталось решить обратную задачу: по значениям многочлена в точках восстановить его коэффициенты. Оказывается, эта задача решается практически тем же самым алгоритмом. Чтобы понять это, запишем формулы перехода от коэффициентов многочлена к значениям в точках в матричном виде:

$$\begin{pmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_0^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Матрица справа называется *матрицей Вандермонда* и обозначается $V(x_0, \ldots, x_{n-1})$.

R Несложно доказать по индукции, что определитель матрицы Вандермонда равен $\prod_{j<i}(x_i - x_j)$, поэтому она обратима для любого набора попарно различных x_i.

Чтобы по значениям в точках получить коэффициенты, нужно умножить вектор значений в точках слева на обратную матрицу. В случае, когда x_i — комплексные корни из единицы n-й степени, обратная матрица выглядит очень просто. Для краткости обозначим $V(1, x, x^2, \ldots, x^{n-1}) = V_n(x)$.

Теорема 1.3.1 Пусть $\omega_n = e^{2\pi i/n}$, тогда

$$V_n(\omega_n)^{-1} = \frac{1}{n} V_n(\omega_n^{-1}).$$

Доказательство. Нужно показать, что

$$\sum_{k=0}^{n-1} \left(V_n(\omega_n)\right)_{j,k} \left(V_n(\omega_n^{-1})\right)_{k,l} = \sum_{k=0}^{n-1} \omega_n^{jk} \omega_n^{-kl} = \sum_{k=0}^{n-1} \omega_n^{k(j-l)} = \begin{cases} 0, & \text{если } j \neq l, \\ n, & \text{если } j = l. \end{cases}$$

При $j = l$ получаем

$$\sum_{k=0}^{n-1} \omega_n^0 = \sum_{k=0}^{n-1} 1 = n.$$

При $j \neq l$ получаем

$$\sum_{k=0}^{n-1} \omega_n^{k(j-l)} = \frac{1 - \omega_n^{n(j-l)}}{1 - \omega_n^{j-l}} = \frac{1 - 1}{1 - \omega_n^{j-l}} = 0.$$

■

Значит, для того, чтобы восстановить коэффициенты произведения многочленов, достаточно запустить тот же алгоритм, но на каждом шаге использовать ω_n^{-1} вместо ω_n.

```
# в результате выполнения функции на место коэффициентов записываются значения в точках
FFT(a, n, isInverse):
  if n == 1:
    return # значение в точке совпадает с единственным коэффициентом
  a --> aOdd, aEven # разбиваем коэффициенты на чётные и нечётные
  FFT(aEven, n / 2, isInverse), FFT(aOdd, n / 2, isInverse)
  if isInverse:
    w = exp(-2 * pi * i / n)
  else:
    w = exp(2 * pi * i / n)
  x = 1
  for j = 0..(n - 1):
    a[j] = aEven[j % (n / 2)] + x * aOdd[j % (n / 2)]
    x *= w

multiply(a, b, c): # результат умножения записывается в c
  k = len(a), m = len(b)
  n = 1
  for (; n < k + m + 1; n *= 2)
  # здесь надо дополнить a и b ведущими нулями
  FFT(a, n, False), FFT(b, n, False)
  for i = 0..(n - 1):
    c[i] = a[i] * b[i]
  FFT(c, n, True)
  for i = 0..(n - 1):
    c[i] /= n
```

Всё вместе работает за $2\Theta(n \log n) + \Theta(n) + \Theta(n \log n) = \Theta(n \log n)$.

1.4 Нерекурсивная версия алгоритма

Текущая версия алгоритма делает много рекурсивных вызовов и использует много дополнительной памяти, поэтому на практике её не используют. Попробуем избавиться от этих недостатков.

Избавляемся от дополнительной памяти

Чтобы не использовать дополнительную память, будем переставлять коэффициенты многочлена внутри массива: в первую половину переставим все чётные коэффициенты, во вторую — все нечётные. Тогда можно не использовать дополнительные массивы, а делать рекурсивные запуски от отрезков текущего.

Тут возникает небольшая проблема с тем, что после рекурсивных запусков нужно пересчитать ответ и записать его туда же, где хранятся результаты рекурсивных запусков. Заметим, однако, что для $0 \leqslant j < \frac{n}{2}$ значения

$$P(\omega_n^j) = P_0(\omega_n^{2j}) + \omega_n^j \cdot P_1(\omega_n^{2j}),$$

$$P(\omega_n^{j+n/2}) = P(-\omega_n^j) = P_0(\omega_n^{2j}) - \omega_n^j \cdot P_1(\omega_n^{2j})$$

зависят только от значений $P_0(\omega_n^{2j})$, $P_1(\omega_n^{2j})$, которые хранятся ровно в тех ячейках массива, в которые надо записать $P(\omega_n^j)$ и $P(\omega_n^{j+n/2})$. Значит, для каждого j значения $P(\omega_n^j)$ и $P(\omega_n^{j+n/2})$ можно пересчитать, используя $O(1)$ дополнительной памяти.

Избавляемся от рекурсии

Заметим, что теперь перед рекурсивными запусками мы просто переставляем местами элементы массива, и только перед выходом из рекурсии делаем какие-то полезные действия. Давайте поймём, в каком порядке будут стоять элементы на самом глубоком уровне рекурсии, тогда от неё получится избавиться.

После всех перестановок элементы будут упорядочены по развёрнутой битовой записи их начального индекса (например, при $n = 8$ на самом глубоком уровне рекурсии элементы будут идти в порядке $a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7$). Действительно, в первую половину массива попали элементы с чётными индексами (то есть с нулевым младшим битом индекса), в первую четверть — с нулевыми двумя младшими битами индекса, и так далее.

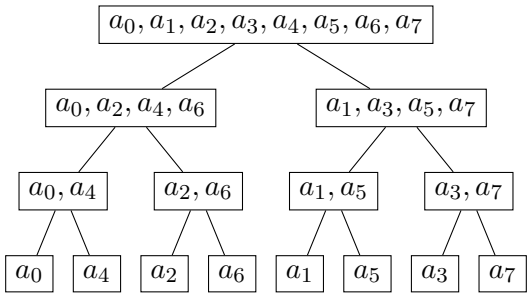

Рис. 1.2: Дерево рекурсивных вызовов при $n = 8$

Развёрнутую битовую запись числа легко посчитать динамикой: пусть $n = 2^q$, тогда

$$rev(i) = \lfloor rev(\lfloor i/2 \rfloor)/2 \rfloor + (i \bmod 2) \cdot 2^{q-1}.$$

Если не хочется использовать дополнительный массив, развёрнутую битовую запись можно посчитать для каждого числа отдельно за $O(q) = O(\log n)$, это не ухудшит оценку сложности.

$rev(rev(i)) = i$, поэтому все индексы разбиваются на пары, и надо просто поменять элементы в каждой паре местами.

При реализации алгоритма в C++ можно использовать встроенные комплексные числа: `std::complex`.

```
# переставляет элементы a в порядке развёрнутой битовой записи, n = 2 ** q
bitReversePermute(a, n, q):
  int rev[n]
  rev[0] = 0
  for j = 1..(n - 1):
    rev[j] = ((rev[j >> 1]) >> 1) ^ ((j & 1) << (q - 1))
  for j = 0..(n - 1):
    if j < rev[j]:
      swap(a[j], a[rev[j]])

# в результате выполнения функции на место коэффициентов записываются значения в точках
FFT(a, n, q, isInverse):
  bitReversePermute(a, n, q)
  for (len = 1; len < n; len *= 2):
    # моделируем рекурсивные запуски на отрезках массива длины 2 * len
    complex<double> w(cos(pi / len), sin(pi / len))
    # w = exp(2 * pi * i / (2 * len))
    if isInverse:
      w.imag = -w.imag
      # w = exp(-2 * pi * i / (2 * len))
    for (i = 0; i < n; i += 2 * len)
      complex<double> x(1, 0)
      for j = 0..(len - 1):
        complex<double> tmp = a[i + len + j] * x
        a[i + len + j] = a[i + j] - tmp
        a[i + j] = a[i + j] + tmp
        x *= w

multiply(a, b, c): # результат умножения записывается в c
  k = len(a), m = len(b)
  n = 1, q = 0
  for (; n < k + m + 1; n *= 2, q += 1)
  # здесь надо дополнить a и b ведущими нулями
  FFT(a, n, q, False), FFT(b, n, q, False)
  for i = 0..(n - 1):
    c[i] = a[i] * b[i]
  FFT(c, n, q, True)
  for i = 0..(n - 1):
    c[i] /= n
```

1.5 Оценка времени работы с учётом погрешности вычислений

На самом деле мы работаем с вещественными числами, которые хранятся в памяти компьютера не абсолютно точно. Можно пользоваться стандартными типами данных (например, `double` в C++), или реализовать вещественные числа с нужной нам точностью. Во втором случае время работы арифметических операций над числами уже будет зависеть от количества хранящихся бит, и используемого алгоритма (в том числе, для операций умножения можно рекурсивно использовать алгоритм FFT!)

Пусть мы перемножаем два числа, имеющих n знаков в десятичной записи. За $O(n)$ их

легко перевести в систему счисления с основанием $B = 10^k$ (конкретное k выберем позже). Получились два многочлена степени $m = \frac{n}{k}$ с целыми коэффициентами, по модулю меньшими B. Коэффициенты произведения многочленов по модулю не превосходят mB^2. При этом мы знаем, что эти коэффициенты окажутся целыми, поэтому достаточно вычислить их с погрешностью меньше 0.5, тогда мы сможем округлить их до ближайшего целого и получить правильный ответ. При использовании типа `double` так получится различать лишь целые числа примерно до 10^{15} (так как мантисса состоит из 52 бит), поэтому должно выполняться неравенство $mB^2 \leqslant 10^{15}$.

Мы пока не учли, что операции над вещественными числами тоже производятся с погрешностью. На практике FFT обладает хорошей вычислительной устойчивостью, поэтому для оценки погрешности вычислений часто используют просто $O(mB^2e)$, где e — точность вычислений (для `double` $e \approx 10^{-15}$). Скажем, если $n = 10^6$, то неравенство $\frac{10^6}{k}10^{2k} \leqslant 10^{15}$ выполняется при $k \leqslant 4$.

Можно формально доказать, что погрешность вычислений не превосходит

$$6m^2B^2 \log m \cdot e = O(m^2B^2 \log m \cdot e).$$

Таким образом, в общем случае нужно делать все промежуточные вычисления с использованием $\Theta(\log m + \log B + \log\log m) = \Theta(\log m + \log B)$ бит, чтобы в конце удалось с помощью округления получить правильный ответ.

Если умножать коэффициенты за квадратичное от числа бит время, получаем оценку времени работы всего алгоритма $O(m \log m(\log m + \log B)^2)$. Если взять $k \approx \log n$, получим оценку

$$O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\left(\log\left(\frac{n}{\log n}\right) + \log\left(10^{\log n}\right)\right)^2\right) = O(n \log^2 n).$$

Можно умножать коэффициенты рекурсивным запуском алгоритма FFT. Если при этом в рекурсивном запуске использовать квадратичные алгоритмы умножения для промежуточных вычислений, получаем оценку

$$O(n \log n(\log\log n)^2).$$

Полная рекурсивная версия имеет оценку

$$O(n \log n \log\log n \log\log\log n \cdots).$$

1.6 Алгоритмы с лучшей асимптотической оценкой

Алгоритм Шёнхаге — Штрассена

Заметим, что всё, что нам нужно было от комплексных чисел — наличие 2^k корней 2^k-й степени из единицы для достаточно большого k. Если для простого p порядок первообразного корня $\varphi(p) = p - 1$ делится на достаточно большую степень двойки, и $mB^2 < p$, то те же вычисления можно произвести по модулю p, то есть вместо поля комплексных чисел использовать поле вычетов по модулю p. При этом все вычисления будут точными, поэтому не будет проблем с погрешностью, которые возникают при использовании комплексных чисел.

На самом деле даже не обязательно использовать модуль, остатки по которому образуют поле. Достаточно, чтобы в кольце остатков было достаточно корней из единицы нужной степени.

Алгоритм Шёнхаге-Штрассена (Schönhage, Strassen, 1971, [25]) вычисляет произведение ab по модулю $2^n + 1$, где n тоже является степенью двойки. Если длина битовой записи ab не превосходит n, то $ab \bmod (2^n + 1) = ab$.

В кольце остатков по модулю $2^n + 1$ есть n корней из единицы степени n: это $2^{2j} \bmod (2^n + 1)$, $0 \leqslant j < n$ (то, что они различны и в степени n дают единицу, следует из того, что $2^n \equiv -1 \pmod{2^n + 1}$). Удобно, что умножение на такие корни из единицы, а также взятие по модулю $2^n + 1$ легко осуществляются битовыми сдвигами и сложениями/вычитаниями.

Алгоритм делит числа a, b на $\sqrt{n}$ блоков по $\sqrt{n}$ бит; вычисляет от получившихся последовательностей преобразование Фурье, пользуясь вышеописанными корнями из единицы; перемножает получившиеся последовательности рекурсивными запусками алгоритма; после чего вычисляет обратное преобразование Фурье. Мы не будем вдаваться в детали, отметим лишь, что, поскольку длина битовой записи числа при каждом рекурсивном запуске уменьшается с n до $\sqrt{n}$, глубина рекурсии имеет порядок $O(\log \log n)$. Время работы всего алгоритма оценивается как $O(n \log n \log \log n)$.

В статье [25], где был описан алгоритм, авторы высказали предположение, что существует алгоритм с честной оценкой времени работы $O(n \log n)$.

Алгоритм Фюрера

Алгоритм Фюрера (Fürer, 2007, [13]) приблизился к этой оценке — время его работы оценивается как $O(n \log n \cdot 2^{O(\log^* n)})$. Этот алгоритм производит преобразование Фурье над кольцом $\mathbb{C}[x]/(x^p + 1)$ для некоторого $p \approx \log n$.

Алгоритм Харви-Ван-дер-Хэвена

Наконец, совсем недавно был опубликован алгоритм с оценкой времени работы $O(n \log n)$ (Harvey, van der Hoeven, 2019, [16]). В алгоритме вычисления производятся над факторкольцом комплексных многочленов от нескольких переменных.

Вопрос, существуют ли более быстрые алгоритмы, остаётся открытым: в общем случае никакой нижней оценки на время умножения лучше $\Omega(n)$ не известно.

Практическая применимость

Не смотря на худшую асимптотическую оценку, на практике используют изученный нами алгоритм и алгоритм Шёнхаге-Штрассена, поскольку за счёт простоты и меньшей константы в оценке времени работы они оказываются быстрее на диапазонах чисел, с которыми работают на практике ($\log \log n \leqslant 6$ для всех $n \leqslant 2^{64}$, поэтому на практике разница в асимптотических оценках куда менее важна, чем константа в оценке времени работы).

2. Быстрое деление

2.1 Метод Ньютона

Метод Ньютона позволяет последовательными приближениями найти корень функции $f(x)$ с заданной точностью. Выбирается начальное приближение x_0, после чего x_{k+1} получается пересечением прямой $y = 0$ и касательной прямой к графику функции в точке x_k. Поскольку касательная прямая задаётся уравнением $y = f'(x_k)(x - x_k) + f(x_k)$, точку x_{k+1} можно найти, решив уравнение $0 = f'(x_k)(x_{k+1} - x_k) + f(x_k)$, то есть по формуле

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Известно, что если начальное приближение x_0 достаточно близко к корню функции $f(\alpha) = 0$, в окрестности α вторая производная $f''(x)$ непрерывна, а первая производная $f'(x)$ не равна нулю, то скорость сходимости будет квадратична: если $\varepsilon_k = x_k - \alpha$, то $|\varepsilon_{k+1}| = O(\varepsilon_k^2)$.

Метод Ньютона используют, помимо быстрого деления, также, например, для быстрого вычисления корня из числа.

2.2 Деление методом Ньютона

Пусть мы хотим поделить n-битное число a на n-битное число b (и a, и b, возможно, с ведущими нулями), то есть найти такие q, r, что $a = qb + r$, $0 \leqslant r < b$.

Сведение к вычислению обратного

Вычислим n-битное приближение $\frac{1}{b}$, то есть такое z с n битами после запятой, что $|z - \frac{1}{b}| < 2^{-n}$. Тогда $|za - \frac{a}{b}| < |a| \cdot 2^{-n} < 1$. Значит, $|za - q| < 2$. Тогда q и r можно найти точно, например, так: вычислим $a - \lfloor za \rfloor b$ и прибавим или вычтем b не более двух раз так, чтобы получить число из диапазона $[0, b)$, то есть r; q получается соответствующим изменением младших бит в $\lfloor za \rfloor$.

Пусть $M(n)$ — сложность умножения двух n-битных чисел. Мы свели деление a на b к нахождению z — n-битного приближения $\frac{1}{b}$, плюс ещё $O(M(n))$ операциям. Покажем, что z тоже можно найти за время $O(M(n))$, тогда получим алгоритм деления, имеющий ту же асимптотическую оценку, что и умножение.

Вычисление обратного методом Ньютона

Искать z будем методом Ньютона: будем приближать корень функции $f(x) = \frac{1}{x} - b$. По рекуррентному соотношению

$$z_{k+1} = z_k - \frac{1/z_k - b}{-1/z_k^2} = z_k + z_k - bz_k^2 = 2z_k - bz_k^2$$

z_{k+1} можно вычислить по z_k, используя только умножения и вычитания. Будем последовательно вычислять приближения следующим образом:

- Считаем, что двоичная запись b имеет вид $0.v_1 \dots v_n$, где $v_1 = 1$ (произвольное b приводится к такому виду битовым сдвигом, к получившемуся в результате z нужно будет применить такой же битовый сдвиг).
- В качестве начального приближения возьмём

$$z_0 = \frac{1}{4} \left\lfloor \frac{32}{4v_1 + 2v_2 + v_3} \right\rfloor.$$

- В начале k-й итерации мы имеем z_k, имеющее $2^k + 1$ бит после точки. Вычислим

$$z_{k+1} = 2z_k - b_k z_k^2 + r,$$

  где b_k имеет двоичную запись $0.v_1 \dots v_{2^{k+1}+3}$; $0 \leqslant r < 2^{-2^{k+1}-1}$ — такое, что z_{k+1} имеет ровно $2^{k+1}+1$ бит после точки (то есть вычислим точно $2z_k - b_k z_k^2$ и округлим).
- Остановимся, когда $2^k + 1 \geqslant n$.

Можно формально показать, что на каждом шаге выполняются неравенства

$$z_k \leqslant 2, \ \left| z_k - \frac{1}{b} \right| \leqslant 2^{-2^k}.$$

Мы ограничимся тем, что покажем идею доказательства: если бы вычисления производились по формуле $z_{k+1} = 2z_k - bz_k^2$, то, обозначив $\varepsilon_k = \frac{1}{b} - z_k$, получаем

$$\varepsilon_{k+1} = \frac{1}{b} - z_{k+1} = \frac{1}{b} - z_k - z_k(1 - bz_k) = \varepsilon_k - z_k \varepsilon_k b = \varepsilon_k - \left(\frac{1}{b} - \varepsilon_k \right) \varepsilon_k b = b\varepsilon_k^2 < \varepsilon_k^2,$$

поскольку $b < 1$. Значит, если $|\varepsilon_k| < 2^{-2^k}$, то $|\varepsilon_{k+1}| < 2^{-2^{k+1}}$.

Для того, чтобы из этого рассуждения получить формальное доказательство (которое можно найти в разделе 4.3.3 в [2]), нужно аккуратно проследить за тем, что использование b_k вместо b и округление на каждом шаге не испортят оценок.

Оценка времени работы

На k-м шаге число $b_k z_k^2$ имеет $2 \cdot (2^k + 1) + 2^{k+1} + 3 = 4 \cdot 2^k + 5$ знаков. Значит, суммарно алгоритм совершает

$$2M(4n) + 2M(2n) + 2M(n) + \dots + O(n)$$

действий. Если $M(n) = O(n \cdot g(n))$, где $g(\cdot)$ — монотонно возрастающая функция (это верно для всех известных алгоритмов умножения), то

$$M(4n) + M(2n) + M(n) + \dots < (4n + 2n + n + \dots) \cdot g(8n) \leqslant M(8n),$$

значит, время работы алгоритма оценивается как $O(M(n))$.

II Алгоритмы на строках

3. Поиск подстроки в строке

Строка — последовательность символов из некоторого *алфавита* Σ. *Подстрока* строки s — любая подпоследовательность подряд идущих символов строки s, то есть $s[l, r)$ для некоторых $0 \leqslant l \leqslant r \leqslant |s|$. *Префикс* строки s — подстрока s, являющаяся "началом" s, то есть $s[0, i)$ для некоторого $i \geqslant 0$. *Суффикс* строки s — подстрока s, являющаяся "концом" s, то есть $s[i, |s|)$ для некоторого $i \leqslant |s|$. *Собственный* суффикс или префикс — тот, что не совпадает со всей строкой.

Вхождение строки s в строку t — это такая позиция i, что строка s совпадает с подстрокой t, начинающейся с позиции i и имеющей длину $|s|$: $s = t[i, i + |s|)$. Вхождением, в зависимости от контекста, будем называть как позицию i, так и саму подстроку, совпадающую с s.

В C++ есть два стандартных способа работы со строками: `char*` и `std::string`. В зависимости от конкретной задачи бывает удобно пользоваться как первым, так и вторым способом.

Задача поиска подстроки в строке формулируется следующим образом: даны строка t (*текст*) и строка s (*образец*), нужно проверить, входит ли s в t как подстрока; или, в более общем случае, нужно найти все вхождения s в t.

3.1 Наивный алгоритм

Самый простой способ решения задачи: проверить для каждой подстроки t, имеющей длину $|s|$, не совпадает ли она с s.

```
# Ищем строку s длины n в строке t длины m
for i = 0..(m - n):
  isEqual = True
  for k = 0..(n - 1):
    if s[k] != t[i + k]:
      isEqual = False
      break
  if isEqual:
    ... # i - позиция вхождения
```

R Строки типа `std::string` можно проверять на равенство встроенным оператором, для сравнения `char*` можно использовать `std::strcmp` и `std::strncmp` (во всех случаях строки длины n сравниваются за $O(n)$).

Получаем время работы $O((|t| - |s| + 1) \cdot |s|) = O(|t| \cdot |s|)$, что приемлемо лишь для достаточно коротких текстов.

3.2 Z-функция

Определение 3.2.1 Для строки s и любого $0 < i < |s|$, $Z_i(s)$ — длина наибольшей подстроки, начинающейся с i-й позиции и совпадающей с префиксом s.

s	a	b	a	c	a	b	a	c	a	b	a	d	a	a	b
$Z(s)$		0	1	0	7	0	1	0	3	0	1	0	1	2	1

Рис. 3.1: Строка и её Z-функция

Если посчитать Z-функцию для строки $s\$t$, где $\$$ — символ, не встречающийся в s и t, то вхождения s в t будут соответствовать таким позициям i, что $Z_i(s\$t) = |s|$: $Z_i(s\$t) = |s|$ тогда и только тогда, когда $s = t[i - |s| - 1, i - 1)$.

При этом Z-функцию строки можно вычислить за линейное от длины строки время.

Лемма 3.2.1 Пусть $l > 0$, $r = l + Z_l(s)$, тогда для любого $l < i \leqslant r$ верно

$$Z_i(s) \geqslant \min(r - i, Z_{i-l}(s)).$$

Доказательство. По определению $Z_l(s)$, $s[l, r) = s[0, Z_l(s))$, тогда $s[i, r) = s[i - l, Z_l(s))$. Если $h = \min(r - i, Z_{i-l}(s))$, то $s[i, i + h) = s[i - l, i - l + h) = s[0, h)$ по определению $Z_{i-l}(s)$. Значит, $Z_i(s) \geqslant h$. ■

Будем вычислять $Z_i(s)$ последовательно, поддерживать пару вида $(l, r) = (l, l + Z_l(s))$ с максимальным r и пользоваться леммой при вычислении очередного значения $Z_i(s)$.

```
# вычислим Z-функцию от строки s длины n
l = r = 0
for i = 1..(n - 1):
  z[i] = max(0, min(r - i, z[i - l]))
  while i + z[i] < n and s[i + z[i]] == s[z[i]]:
    z[i] += 1
  if i + z[i] > r:
    l = i, r = i + z[i]
```

Предложение 3.2.2 Время работы получившегося алгоритма — $O(n)$.

Доказательство. Заметим, что если $Z_{i-l}(s) < r - i$, то

$$s[i + Z_{i-l}(s)] = s[i - l + Z_{i-l}(s)] \neq s[Z_{i-l}(s)],$$

то есть в этом случае во внутренний цикл мы не попадём. Значит, если на i-й итерации внешнего цикла мы сделали $k > 0$ итераций внутреннего цикла, то в конце i-й итерации r тоже увеличится хотя бы на k. Поскольку в конце $r \leqslant n$, получаем суммарное время работы $O(n + n) = O(n)$. ■

Таким образом, мы научились решать задачу поиска подстроки в строке за $O(|s| + |t|)$. Отметим, что при этом можно использовать лишь $O(|s|)$ дополнительной памяти, так как $Z_i(s\$t) \leqslant |s|$, поэтому промежуточные значения Z-функции при $i \geqslant |s|$ на последующих шагах не пригодятся, и их можно не хранить.

3.3 Префикс-функция и алгоритм Кнута-Морриса-Пратта

Определение 3.3.1 Для строки s и любого $0 \leqslant i < |s|$, $\pi_i(s)$ — длина наибольшего собственного суффикса $s[0, i]$, совпадающего с префиксом s.

Алгоритм Кнута-Морриса-Пратта (Knuth, Morris, Pratt, 1977, [19]) поиска подстроки в строке очень похож на алгоритм, использующий Z-функцию, только использует вместо неё префикс-функцию. Позиции i такие, что $\pi_i(s\$t) = |s|$, снова соответствуют вхождениям s в t: $\pi_i(s\$t) = |s|$ тогда и только тогда, когда $s = t[i - 2|s|, i - |s|)$.

Осталось научиться вычислять префикс-функцию за линейное от длины строки время.

s	a	b	a	c	a	b	a	c	a	b	a	d	a	a	b
$\pi(s)$	0	0	1	0	1	2	3	4	5	6	7	0	1	1	2

Рис. 3.2: Строка и её префикс-функция

Лемма 3.3.1 Для $0 \leqslant i < |s|$ обозначим за $P_i(s)$ множество длин собственных суффиксов $s[0, i]$, совпадающих с префиксом s. Тогда $P_i(s) = \{0\}$, если $\pi_i(s) = 0$; иначе $P_i(s) = \{\pi_i(s)\} \cup P_{\pi_i(s)-1}(s)$.

Доказательство. Пусть $\pi_i(s) > 0$, $k \in P_i(s)$, $k \neq \pi_i(s)$. Тогда $k < \pi_i(s)$, и, поскольку $s(i - \pi_i(s), i] = s[0, \pi_i(s))$,

$$s[0, k) = s(i - k, i] = s[\pi_i(s) - k, \pi_i(s)),$$

значит, $k \in P_{\pi_i(s)-1}(s)$. ■

Таким образом,

$$P_i(s) = \{\pi_i(s), \pi_{\pi_i(s)-1}(s), \pi_{\pi_{\pi_i(s)-1}(s)-1}(s), \dots\}.$$

Заметим, что если $\pi_i(s) > 0$, то, поскольку $s(i - \pi_i(s), i] = s[0, \pi_i(s))$, верно и $s(i - \pi_i(s), i - 1] = s[0, \pi_i(s) - 1)$. Значит, $\pi_i(s) - 1 \in P_{i-1}(s)$.

Получаем, что либо $\pi_i(s) = 0$, либо $\pi_i(s)$ на единицу больше максимального такого $k \in P_{i-1}(s)$, что $s(i - 1 - k, i] = s[0, k]$. Поскольку для любого $k \in P_{i-1}(s)$ верно $s(i - 1 - k, i - 1] = s[0, k)$, достаточно проверять условие $s[i] = s[k]$.

```
# вычислим префикс-функцию от строки s длины n
p[0] = 0
for i = 1..(n - 1):
  k = p[i - 1]
  while k > 0 and s[i] != s[k]:
    k = p[k - 1]
  if s[i] == s[k]:
    k += 1
  p[i] = k
```

Предложение 3.3.2 Время работы получившегося алгоритма — $O(n)$.

Доказательство. На каждом шаге k увеличивается не более, чем на один, при этом на каждой итерации внутреннего цикла k строго уменьшается. В любой момент времени $k \leqslant n$, значит суммарное количество итераций внутреннего цикла не превосходит $2n$. ■

Алгоритм Кнута-Морриса-Пратта снова имеет сложность $O(|s| + |t|)$, при этом снова достаточно $O(|s|)$ дополнительной памяти (поскольку $\pi_i(s\$t) \leqslant |s|$, значения префикс-функции при $i \geqslant |s|$ можно не хранить).

3.4 Алгоритм Бойера-Мура

Алгоритм Бойера-Мура (Boyer, Moore, 1977, [6]) похож на наивный алгоритм, но использует несколько оптимизаций, благодаря которым он зачастую посещает сильно меньше, чем все $|s| + |t|$ символов строк. При поиске по тексту в браузере или в текстовом редакторе зачастую используется именно алгоритм Бойера-Мура.

Как и в наивном алгоритме, мы будем "подставлять" строку s ко всем позициям в строке t по очереди и проверять, не совпадает ли она с подстрокой, которая оказалась напротив. Оптимизации позволят нам зачастую сдвигать s вправо не на одну позицию, а сразу на несколько.

Сравнение справа налево

Первое отличие от наивного алгоритма — символы s будем сравнивать с символами подстроки t, начиная с конца, то есть не слева направо, а справа налево. Само по себе это никак не влияет на время работы, но это изменение важно для последующих оптимизаций.

Правило плохого символа (bad character rule)

Пусть мы сравнивали s с $t[i, i + |s|)$, и нашли несовпадение в k-м символе: $s[k] \neq t[i + k]$. Тогда сдвинем s вправо не на одну позицию, а сразу так, чтобы $t[i+k]$ совпало с символом s, оказавшимся напротив.

Для того, чтобы быстро вычислить требуемый сдвиг, предподсчитаем для каждого символа алфавита список всех его вхождений в s. Тогда, если $t[i + k] = y$, найдём в списке вхождений y в s ближайшее к $s[k]$ слева вхождение y, и сдвинем s так, чтобы оно оказалось напротив $t[i + k]$; если же вхождений y в s слева от $s[k]$ нет, то совпадения с $t[i + k]$ добиться уже не удастся; значит, в этом случае можно пропустить все подстроки, содержащие $t[i + k]$.

Описанную выше оптимизацию называют расширенным правилом плохого символа, так как в оригинальной версии алгоритма использовалось более простое правило: для каждого символа алфавита запоминалась лишь позиция самого правого его вхождения в s. Такая версия правила срабатывает реже (только если самое правое вхождение y в s находится левее $s[k]$), но она использует меньше дополнительной памяти и не тратит времени на поиск в списке вхождений, что в некоторых случаях более чем компенсирует то, что длинные сдвиги случаются реже.

Правило хорошего суффикса (good suffix rule)

Правило плохого символа уже сильно ускоряет алгоритм на практике, особенно при работе с текстами, написанными на естественных языках. Однако оно не очень эффективно, когда алфавит небольшой (например, при работе с двоичными строками или с последовательностями ДНК). Поэтому часто используют ещё одну оптимизацию — правило хорошего суффикса.

Пусть при сравнении s и $t[i, i+|s|)$ нашлось несовпадение в k-м символе: $s[k] \neq t[i+k]$. Тогда сдвинем s так, чтобы символы на позициях напротив $t(i + k, i + |s|)$ не поменялись (то есть чтобы сравнения на этих позициях продолжили выполняться), а символ напротив $t[i + k]$, наоборот, поменялся.

R В оригинальной версии алгоритма правило хорошего суффикса было сформулировано без последнего уточнения. Поэтому сформулированное нами правило также называют сильным правилом хорошего суффикса.

Мы опустим детали реализации этого правила; отметим лишь, что требуемый сдвиг можно вычислить за $O(1)$, если пользоваться предподсчитанной Z-функцией от строки, полученной переворотом s.

Оба правила можно применять одновременно, то есть делать максимальный сдвиг из тех двух, что предлагают правила.

Можно показать, что при использовании сильного правила хорошего суффикса (даже если при этом не используется правило плохого символа) алгоритм работает за $O(|s|+|t|)$, если s не входит в t как подстрока (доказательство можно прочитать, например, в главе 3.2 в [1]).

Если вхождения s в t существуют, то время работы алгоритма по-прежнему оценивается в худшем случае как $\Theta(|s| \cdot |t|)$ (и оценка достигается, например, на строках, являющихся

многократным повторением одного символа). Однако можно добавить ещё одну несложную оптимизацию (правило Галиля), чтобы алгоритм работал за $O(|s| + |t|)$ ($O(|s| + |t| + |\Sigma|)$ при использовании правила плохого символа) в худшем случае (доказательство можно прочитать всё в той же главе 3.2 в [1]).

Отметим, что при использовании одного из правил самого по себе алгоритм будет делать в лучшем случае $O(|t|/|s|)$ операций. На практике алгоритм действительно зачастую работает значительно быстрее алгоритмов КМП и Z-функции (всегда работающих за линейное от суммы длин строк время).

3.5 Полиномиальные хеши и алгоритм Рабина-Карпа

Вместо того, чтобы сравнивать строки напрямую, можно вычислять от них хеш-функцию и сравнивать её значения. Если при поиске подстроки s в строке t мы будем заново с нуля вычислять значение хеш-функции для каждой подстроки t длины $|s|$, то никакого выигрыша во времени по сравнению с наивным алгоритмом мы не получим. Поэтому от хеш-функции требуется не только малая вероятность коллизий, но и, например, возможность быстро пересчитывать $h(t[i+1, i+1+|s|))$ по $h(t[i, i+|s|))$. Такие хеш-функции называют *кольцевыми хешами (rolling hash)*.

Определение 3.5.1 Пусть $|s| = n$, $\Sigma \subset \{0, 1, \ldots, q-1\}$, $0 \leqslant x < q$, тогда

$$h_{q,x}(s) = (x^{n-1}s_0 + x^{n-2}s_1 + \cdots + xs_{n-2} + s_{n-1}) \bmod q$$

есть *полиномиальный хеш* строки s.

Будем опускать индексы q, x, когда речь идёт об одной конкретной хеш-функции.

Полиномиальный хеш является кольцевым: пусть известно $h(t[i, i+n))$, тогда

$$\begin{aligned} h(t[i+1, i+1+n)) &= (x^{n-1}t_{i+1} + x^{n-2}t_{i+2} \cdots + t_{i+n}) \bmod q \\ &= ((h(t[i, i+n)) - x^{n-1}t_i) \cdot x + t_{i+n}) \bmod q. \end{aligned}$$

Алгоритм Рабина-Карпа

Алгоритм Рабина-Карпа (Rabin, Karp, 1987, [17]) вычисляет хеш-функцию от шаблона s, после чего поочерёдно вычисляет хеш-функции от всех подстрок текста t длины $|s|$ и сравнивает полученные значения с $h(s)$. Если при этом использовать кольцевой хеш (например, полиномиальный хеш), время работы составит $O(|s|+|t|)$, при этом потребуется $O(1)$ дополнительной памяти, что выгодно отличает алгоритм от предыдущих. Недостаток состоит в том, что часть найденных алгоритмом позиций могут не быть вхождениями s в t, если случались коллизии хеш-функции.

```
# Ищем строку s длины n в строке t длины m
hashS = 0
for i = 0..(n - 1):
  hashS = (hashS * x + s[i]) % q

xpow = pow(x, n - 1, q) # возведение в степень по модулю
h = 0
for i = 0..(n - 1):
  h = (h * x + t[i]) % q
for i = 0..(m - n):
  if h == hashS:
    ... # нашли потенциальное вхождение
  if i < n - m:
    h = (h - xpow * t[i]) % q # в C++ эту строку надо писать аккуратнее
    h = (h * x + t[i + n]) % q
```

При совпадении значений хеш-функции можно за $O(|s|)$ проверять, действительно ли позиция является вхождением. Тогда алгоритм не будет ошибаться, но будет работать в худшем случае за $O(|s| \cdot |t|)$.

R Если действовать чуть более хитро, можно за линейное время проверить, правда ли, что все найденные позиции являются вхождениями (подробности в конце раздела 4.4.2 в [1]). Если это не так, можно перезапустить алгоритм с другой хеш-функцией. Получился алгоритм, который никогда не ошибается и работает в среднем за $O(|s| + |t|)$ (если вероятность коллизий мала).

Оценка вероятности ошибки

Для того, чтобы коллизии случались редко, будем использовать простое q, а x выберем случайно перед началом работы.

Предложение 3.5.1 Пусть $s \neq t$, $|s| = |t|$ — произвольные строки одной длины с символами из алфавита $\Sigma \subset \{0, 1, \ldots, q-1\}$, где q — простое. Тогда $h_{q,x}(s) = h_{q,x}(t)$ для не более чем $|s| - 1$ различных x.

Доказательство. Рассмотрим $g(x) = h_{q,x}(s) - h_{q,x}(t)$ — многочлен над $\mathbb{Z}_q$. Заметим, что $\deg(g) \leqslant |s|-1$, при этом $g(x)$ не является тождественным нулём, так как $s \neq t$. Поскольку q простое, $\mathbb{Z}_q$ является полем, тогда $g(x)$ имеет не более $|s| - 1$ корней. ■

Следствие 3.5.2 Пусть $s_1 \neq t_1, \ldots, s_k \neq t_k$ ($|s_1| = |t_1|, \ldots, |s_k| = |t_k|$) — произвольные строки длины не более n с символами из алфавита $\Sigma \subset \{0, 1, \ldots, q-1\}$, где q — простое. Пусть $0 \leqslant x < q$ выбрано случайно равновероятно. Тогда вероятность того, что у $h_{q,x}$ случилась коллизия хотя бы на одной паре s_i, t_i (то есть $h_{q,x}(s_i) = h_{q,x}(t_i)$ для некоторого i) не превышает $\frac{kn}{q}$.

Доказательство. Пусть P_i — вероятность коллизии на паре s_i, t_i. Из предложения 3.5.1 следует, что $P_i \leqslant \frac{n}{q}$ для любого i. Вероятность того, что коллизия произошла хотя бы на одной паре, не превышает $P_1 + \cdots + P_k \leqslant \frac{kn}{q}$. ■

R Заметим, что семейство полиномиальных хешей не является универсальным, так как вероятность коллизии на фиксированной паре строк длины n оценивается лишь как $\frac{n}{q}$, а не $\frac{1}{q}$. Тем не менее, на практике при выборе q часто пользуются оценкой вероятности коллизии $\frac{1}{q}$.

Возвращаясь к алгоритму Рабина-Карпа, если использовать $q \approx |t||s|r$, получим вероятность нахождения ложного вхождения не более $O(|t||s|/(|t||s|r)) = O(1/r)$.

Если $|t||s|r$ слишком большое (например, не помещается в машинное слово), можно использовать одновременно несколько хеш-функций с q_i меньшего размера. Пусть используются l хеш-функций с $q_i \approx q$, $1 \leqslant i \leqslant l$, тогда вероятность того, что в хотя бы в одной позиции произошла коллизия по всем хеш-функциям одновременно, есть $O(|t||s|^l/q^l)$. Для достижения оценки $O(1/r)$ достаточно уже $q_i \approx (|t|r)^{1/l}|s|$.

Если нас интересуют не все вхождения, а лишь какое-нибудь одно, то достаточно использовать одну хеш-функцию с $q \approx |t||s|$. При этом алгоритм, делающий честную проверку при совпадении хеш-функций, и останавливающийся, когда находит вхождение, сделает в среднем $O(|t|+|s|)$ шагов (так как коллизии в среднем случатся $O(|t|/|t|) = O(1)$ раз).

Наибольшая общая подстрока

С помощью полиномиальных хешей можно достаточно просто решать множество других задач. В качестве примера, научимся искать наибольшую по длине общую подстроку двух строк s и t.

Первая полезная идея: если предподсчитать значения хеш-функции от всех префиксов строки, то с их помощью можно за $O(1)$ вычислить значение хеш-функции от любой её подстроки:

$$h(s[l,r)) = (h(s[0,r)) - h(s[0,l)) \cdot x^{r-l}) \bmod q.$$

Далее, если у s и t есть общая подстрока длины k, то у них есть и общая подстрока любой меньшей длины. Значит, можно делать двоичный поиск по k.

При фиксированном k вычислим значения хеш-функции от всех подстрок s длины k и сложим их в хеш-таблицу. После этого вычислим значения хеш-функции от всех подстрок t длины k, и проверим, лежит ли хоть одно из них в хеш-таблице.

Получаем алгоритм со средним временем работы $O(n \log n)$, где $n = \max(|s|, |t|)$, использующий $O(n)$ дополнительной памяти. При этом вероятность ошибки не превосходит $O(n^3 \log n/q)$ (так как на каждом шаге двоичного поиска мы неявно сравниваем $O(n^2)$ пар строк длины $O(n)$).

Можно при нахождении подстрок с одинаковым значением хеш-функции проверять совпадение посимвольно, при этом достаточно найти одно совпадение, чтобы завершить текущий шаг двоичного поиска. При $q \approx n^3$ на каждом шаге в среднем случится $O(1)$ коллизий, поэтому даже с учётом дополнительной проверки каждый шаг будет работать в среднем линейное время. Получаем уже никогда не ошибающийся алгоритм с тем же средним временем работы $O(n \log n)$.

Поиск множества подстрок одинаковой длины в строке

Алгоритм Рабина-Карпа легко обобщается на задачу поиска в тексте t сразу нескольких (m) строк одинаковой длины n. Сложим значения хеш-функции от всех этих строк в хеш-таблицу, после чего вычислим значение хеш-функции от каждой подстроки t длины n, и проверим, лежит ли это значение в хеш-таблице. Получаем алгоритм со средним временем работы $O(|t| + mn)$, пропорциональным сумме длин всех строк. Вероятность ошибки при этом не превышает $O((|t|m + m^2)n/q)$.

R Существуют и другие алгоритмы, решающие задачу поиска подстроки в строке, в том числе:

- алгоритм Shift-And, эффективный на коротких шаблонах, и обобщающийся на поиск подстрок, *почти* совпадающих с шаблоном;
- двусторонний алгоритм (two-way string-matching, 1991, [7]), имеющий линейное от длины строк время работы и использующий $O(1)$ дополнительной памяти.

4. Поиск множества подстрок в строке

Рассмотрим более общую задачу: дан текст t и *словарь* — набор строк $s_1, \ldots, s_m$. Необходимо найти все вхождения строк из словаря в текст.

Изученные нами алгоритмы позволяют решать эту задачу за $O(mt + \sum_i |s_i|)$, применяя алгоритм для каждой строки из словаря отдельно. Алгоритм Рабина-Карпа позволяет решить задачу быстрее, но лишь в случае, когда все строки в словаре имеют одинаковую длину. Наша цель — научиться решать задачу в общем случае за $O(t + \sum_i |s_i| + k)$, где k — суммарное количество вхождений всех слов из словаря.

4.1 Бор

Бор (trie) представляет собой подвешенное дерево, на каждом ребре которого написан символ, при этом символы на любой паре рёбер, исходящих из одной вершины, различны. Некоторые вершины бора отмечены как терминальные. Каждой вершине бора соответствует строка, получающаяся выписыванием всех символов на пути от корня до этой вершины. В боре *хранится* множество строк, соответствующее терминальным вершинам.

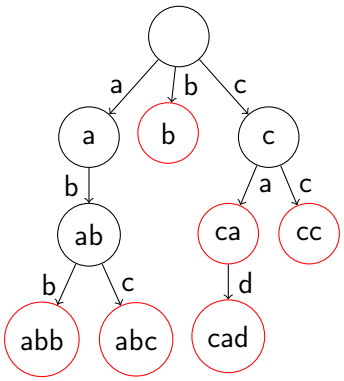

Рис. 4.1: Бор на множестве строк $\{abb, abc, b, ca, cad, cc\}$. Терминальные вершины отмечены красным цветом.

Список исходящих из вершины рёбер можно поддерживать разными способами:

- Самый простой способ — хранить в каждой вершине массив размера $|\Sigma|$. Тогда бор на n вершинах будет занимать $O(n|\Sigma|)$ памяти, при этом поиск строки s в боре можно осуществлять за $O(|s|)$, добавление новой строки s в бор — за $O(|s||\Sigma|)$ (так как в худшем случае может появиться $|s|$ новых вершин).
- Можно хранить список символов, по которым есть переходы, в двоичном дереве поиска (например, в `std::map` в C++). Такой бор будет занимать уже $O(n)$ памяти (суммарное число рёбер не превышает числа вершин), но поиск/добавление строки будет осуществляться за $O(|s| \log |\Sigma|)$.
- Наконец, вместо двоичного дерева поиска можно использовать хеш-таблицу. Получаем $O(n)$ памяти (если поддерживать хеш-таблицы размера, пропорционального

числу исходящих рёбер, и периодически перестраивать их при добавлении новых рёбер) и $O(|s|)$ в среднем на поиск и добавление (все оценки с достаточно большой константой).

При использовании первых двух способов (но не последнего) легко выписать все строки, лежащие в боре, в лексикографическом порядке: выполним обход в глубину из корня, перебирая рёбра в вершине в порядке возрастания написанных на них символов; будем поддерживать строку на пути от корня до текущей вершины в стеке и выписывать её, когда попадаем в терминальную вершину. Получаем время работы $O(|\Sigma|n + L)$ при использовании массивов и $O(n + L)$ при использовании деревьев поиска (L — суммарная длина лежащих в боре строк).

```
class Node:
  Node *next[K] # считаем, что алфавит - числа от 0 до K - 1
  bool term

class Trie:
  Node *root

  insert(s): # добавить строку s в бор
    v = root
    for i = 0..(len(s) - 1):
      if not v->next[s[i]]:
        v->next[s[i]] = new Node()
      v = v->next[s[i]]
    v->term = True

  find(s):
    v = root
    for i = 0..(len(s) - 1):
      if not v->next[s[i]]:
        return False
      v = v->next[s[i]]
    return v->term

  dfs(v, s):
    if v->term:
      ... # s - одна из строк, лежащих в боре
    for i = 0..(K - 1):
      if v->next[i]:
        s.push_back(i)
        dfs(v->next[i], s)
        s.pop_back()

  traverse():
    vector<int> s
    dfs(root, s)
```

Сортировка строк бором

С помощью бора можно отсортировать строки $s_1, \ldots, s_m$: положим их в бор и выпишем в лексикографическом порядке, получаем время $O((\sum_i |s_i|) \log |\Sigma|)$.

Поиск множества подстрок в строке

Положим все слова из словаря в бор, после чего для каждого суффикса текста попытаемся пройти по символам этого суффикса в боре. Встреченные терминальные вершины соответствуют вхождениям слов из словаря в текст. Получаем решение за $O(\sum_i |s_i| + |t| \max_i(|s_i|))$.

4.2 Алгоритм Ахо-Корасик

Начиная с этого момента, будем для упрощения анализа сложности считать, что размер алфавита $|\Sigma|$ — константа.

Следующий алгоритм (Alfred Aho, Margaret Corasick, 1975, [4]) в каком-то смысле является обобщением алгоритма Кнута-Морриса-Пратта.

Будем обозначать строку, соответствующую вершине v, как $S(v)$. Для каждой вершины v предподсчитаем *суффиксную ссылку* $suff(v)$ — ссылку на такую вершину u, что $S(u)$ — наибольший собственный суффикс $S(v)$, соответствующий какой-то вершине бора; а также *сжатую суффиксную ссылку* $up(v)$ — ссылку на такую **терминальную** вершину u, что $S(u)$ — наибольший собственный суффикс $S(v)$, соответствующий **терминальной** вершине бора; либо на корень, если такой терминальной вершины не существует. Условимся, что суффиксная и сжатая суффиксная ссылки от корня указывают на него самого.

R Для бора, построенного на одной строке, суффиксные ссылки соответствуют значениям префикс-функции от этой строки.

Для удобства будем считать, что корень — не терминальная вершина (искать вхождения пустой строки не очень интересно).

Поиск множества подстрок в строке с помощью суффиксных ссылок

С помощью суффиксных ссылок можно для каждого префикса текста $t[0, i)$ найти такую вершину v_i, что $S(v_i)$ — максимальный по длине суффикс $t[0, i)$, соответствующий вершине бора. v_0 — это просто корень бора. Пусть v_i уже известно; либо v_{i+1} — корень, либо $S(v_{i+1})$ без последнего символа является суффиксом $S(v_i)$. Во втором случае v_{i+1} можно найти, переходя по суффиксным ссылкам из v_i, пока не встретится вершина, из которой в боре есть переход по символу $t[i]$ (возможно, эта вершина — сама v_i; этот переход и будет вести в v_{i+1}).

Заметим, что $|S(v_{i+1})| \leqslant |S(v_i)| + 1 - k_i$, где k_i — число переходов по суффиксным ссылкам при вычислении v_{i+1} вышеуказанным способом. Тогда $|S(v_{|t|})| \leqslant |t| - \sum_i k_i$, откуда $\sum_i k_i \leqslant |t|$. Значит, все v_i можно вычислить за $O(|t|)$.

Остаётся заметить, что для любого вхождения слова из словаря в текст $s_j = t[i - |s_j|, i)$ является суффиксом $S(v_i)$, значит, вершина, соответствующая s_j, достижима из v_i по суффиксным ссылкам. Тогда все вхождения, заканчивающиеся в позиции $i - 1$, можно найти, переходя из v_i по сжатым суффиксным ссылкам. Получаем алгоритм, находящий все вхождения за $O(|t| + k)$, где k — суммарное число вхождений всех слов из словаря.

```
v = root
for i = 0..(len(t) - 1):
  while v != root and not v->next[t[i]]:
    v = v->suff
  if v->next[t[i]]:
    v = v->next[t[i]]
  w = v if v->term else v->up
  while w != root:
    ... # вхождение S(w) заканчивается в позиции i
    w = w->up
```

Вычисление суффиксных ссылок

Будем вычислять суффиксные ссылки в порядке обхода в ширину, начиная от корня. Пусть мы находимся в вершине v, и суффиксная ссылка от v (а также от всех вершин, находящихся на меньшем расстоянии от корня) уже посчитана. Найдём суффиксные

ссылки от всех вершин, в которые можно прийти из v, перейдя по ребру бора. Пусть w — такая вершина, то есть $S(w) = \overline{S(v)x}$ для некоторого символа x (под $\overline{ab}$ здесь и далее будем иметь в виду строку, полученную *конкатенацией* ("склеиванием") строк/символов a и b).

Пусть суффиксная ссылка от w должна вести в вершину q. Если q — не корень, то рассмотрим предыдущую вершину r на пути от корня к q ($\overline{S(r)x} = S(q)$), и заметим, что, поскольку $S(q)$ — собственный суффикс $S(w)$, $S(r)$ — собственный суффикс $S(v)$. Значит, q можно найти, переходя из $suff(v)$ по суффиксным ссылкам, пока не встретим вершину, из которой есть переход по символу x.

Осталось вычислить сжатую суффиксную ссылку от w; либо $up(w) = suff(w)$ (если $suff(w)$ — терминальная), либо $up(w) = up(suff(w))$.

```
root->suff = root->up = root
q <-- root
while not q.empty():
  q --> v
  for x = 0..(K - 1): # считаем, что алфавит - числа от 0 до K - 1
    if not v->next[x]:
      continue
    w = v->next[x]
    r = v->suff
    while r != root and not r->next[x]:
      r = r->suff
    if r->next[x] and r->next[x] != w:
      w->suff = r->next[x]
    else:
      w->suff = root
    w->up = w->suff if w->suff->term else w->suff->up
```

Пусть $v_0, v_1, \ldots, v_{|s_j|}$ — вершины на пути от корня до вершины, соответствующей слову s_j из словаря. Оценим суммарное время вычисления суффиксных ссылок от этих вершин. Пусть $suff(v_i) = w_i$. Заметим, что $|S(w_{i+1})| \leqslant |S(w_i)| + 1 - k_i$, где k_i — число переходов по суффиксным ссылкам (не считая перехода от v_i к w_i) при вычислении суффиксной ссылки от v_{i+1}. Тогда $|S(w_{|s_j|})| \leqslant |s_j| - \sum_i k_i$, откуда $\sum_i k_i \leqslant |s_j|$. Значит, суммарное время вычисления суффиксных ссылок от вершин $v_0, \ldots, v_{|s_j|}$ линейно зависит от $|s_j|$.

Поскольку любая вершина бора лежит на пути до хотя бы одной вершины, соответствующей слову из словаря, суммарное время вычисления всех суффиксных ссылок можно оценить сверху суммарной длиной слов в словаре.

R Заметим, что время вычисления суффиксных ссылок не обязательно линейно зависит от числа вершин в боре. Например, рассмотрим бор, построенный на m строках длины n таких, что первые $n-1$ символов каждой из них равны одному и тому же символу x, а последние символы этих строк попарно различны. В таком боре $n+m$ вершин, но суммарное время вычисления всех суффиксных ссылок составит $\Theta(nm)$.

Получился алгоритм со временем работы $O(\sum_i |s_i| + |t| + k)$ (где k — суммарное число вхождений), использующий $O(\sum_i |s_i|)$ дополнительной памяти (обе оценки при условии, что размер алфавита — константа).

Ленивое вычисление суффиксных ссылок

Можно вычислять суффиксные ссылки по мере надобности: пусть нам понадобилась суффиксная ссылка от вершины w, тогда вычислим её, пользуясь теми же рассуждениями, что и выше. Часть суффиксных ссылок, используемых в рассуждениях (например,

суффиксная ссылка от родителя w в боре) также могут быть ещё не вычислены; если это так, вычислим их рекурсивно. При этом бесконечного цикла не случится, так как для вычисления суффиксной ссылки от w нам понадобятся лишь вершины, находящиеся ближе к корню, чем w.

Суммарное время работы оценивается так же, как и раньше; недостаток этого способа в том, что для каждой вершины нужно дополнительно хранить ссылку на родителя и последний символ строки, соответствующей вершине (то есть символ, по которому в вершину можно прийти из родителя).

Построение автомата переходов

Если мы можем позволить себе использовать $O((\sum_i |s_i|)|\Sigma|)$ дополнительной памяти (то есть хранить переходы в боре в виде массивов), можно превратить бор в автомат, в котором из каждой вершины будет переход по каждому символу. В исходном боре каждой вершине v соответствовала строка $S(v)$. В автомате переход из вершины v по символу x будет вести в такую вершину $go(v, x) = u$, что $S(u)$ — наибольший суффикс строки $\overline{S(v)x}$, соответствующий какой-то вершине бора.

Вычислять переходы автомата будем одновременно с суффиксными ссылками. Заметим, что если в боре был переход из вершины v по символу x, то в автомате он не поменяется. В противном случае, $go(v, x) = go(suff(v), x)$. При этом уже посчитанные переходы упрощают вычисление суффиксных ссылок: если $S(w) = \overline{S(v)x}$, то $suff(w) = go(suff(v), x)$ (за исключением случая, когда v — корень; в этом случае $suff(w) = v$).

```
root->suff = root->up = root
for x = 0..(K - 1):
  root->go[x] = root->next[x] if root->next[x] else root
q <-- root
while not q.empty():
  q --> v
  for x = 0..(K - 1):
    if not v->next[x]:
      continue
    w = v->next[x]
    r = v->suff
    w->suff = r->go[x] if r->go[x] != w else root
    w->up = w->suff if w->suff->term else w->suff->up
    for y = 0..(K - 1):
      w->go[y] = w->next[y] if w->next[y] else w->suff->go[y]
```

Точно так же упрощается поиск вхождений в тексте t (при этом уменьшается константа во времени работы): теперь $v_{i+1} = go(v_i, t[i])$.

```
v = root
for i = 0..(len(t) - 1):
  v = v->go[t[i]]
  w = v if v->term else v->up
  while w != root:
    ... # вхождение S(w) заканчивается в позиции i
    w = w->up
```

Переходы автомата, как и суффиксные ссылки, тоже можно вычислять лениво по мере надобности.

Поиск первых вхождений

Если для каждого слова из словаря нас интересуют не все его вхождения, а лишь какое-нибудь одно (например, первое), то можно избавиться от суммарного количества вхождений

в оценке времени работы.

Будем хранить в каждой терминальной вершине пометку, были ли мы в ней до этого. В тот момент, когда мы перемещаемся по сжатым суффиксным ссылкам и отмечаем вхождения, остановимся, если попали в уже помеченную вершину v. Все вершины, достижимые из v по сжатым суффиксным ссылкам, были помечены на том же шаге, что и v, поэтому мы не пропустили первых вхождений никаких слов из словаря. Теперь каждый прыжок по сжатой суффиксной ссылке соответствует первому вхождению какого-то слова из словаря, поэтому суммарное число таких прыжков не превосходит количества слов в словаре. Получаем время работы $O(\sum_i |s_i| + |t|)$.

Вместо хранения пометок можно просто перенаправлять сжатые суффиксные ссылки посещённых вершин в корень.

5. Суффиксный массив

Во всех предыдущих алгоритмах мы предварительно обрабатывали шаблоны, которые потом искали в тексте. Суффиксный массив, наоборот, представляет собой предварительную обработку текста. Он оказывается намного более мощным инструментом, позволяющим решать куда более широкий круг задач.

Определение 5.0.1 *Суффиксный массив* (Manber, Mayers, 1990, [20]; Gonnet, 1987, [15]) строки t — это упорядоченный в лексикографическом порядке список суффиксов t. Поскольку каждый суффикс задаётся позицией его начала в строке, суффиксный массив можно считать перестановкой чисел от 0 до $|t|-1$.

6	aab
7	ab
4	abaab
0	abacabaab
2	acabaab
8	b
5	baab
1	bacabaab
3	cabaab

Рис. 5.1: Суффиксный массив строки “abacabaab”

Здесь и далее по умолчанию говорим о суффиксном массиве строки t длины n. Будем использовать обозначение $suf(t,i) = t[i,n)$.

Самый простой способ построить суффиксный массив — просто отсортировать все суффиксы. Если сравнивать суффиксы посимвольно за $O(n)$, получаем время построения $O(n^2 \log n)$.

Предподсчитаем полиномиальный хеш от всех префиксов строки t, тогда, как мы уже знаем, мы можем вычислить хеш от любой подстроки t за $O(1)$. Теперь можно сравнить два суффикса $suf(t,i)$ и $suf(t,j)$ за $O(\log n)$ (в следующих рассуждениях считаем, что коллизий не произошло): двоичным поиском найдём максимальное l такое, что $h(t[i,i+l)) = h(t[j,j+l))$, после чего сравним символы $t[i+l]$ и $t[j+l]$, и поймём, какой из суффиксов меньше (если $i+l=n$ либо $j+l=n$, то соответствующий суффикс меньше). Получаем алгоритм построения суффиксного массива за $O(n \log^2 n)$ с некоторой вероятностью ошибки.

5.1 Построение суффиксного массива за $O(n \log n)$

Будем считать, что последний символ строки t — это $\$$ — символ, меньший всех остальных. К произвольной строке можно добавить $\$$ в конец, при этом порядок суффиксов в суффиксном массиве не поменяется, только в начало добавится суффикс из последнего символа.

Циклической подстрокой строки t длины k, начинающейся в позиции i, будем называть обычную подстроку $t[i, i+k)$, если $i+k \leqslant n$, и конкатенацию строк $t[i, n)$ и $t[0, k-(n-i))$ иначе. Для удобства при обсуждении циклических подстрок будем использовать обозначение $t[i, i+k)$ в обоих случаях. *Циклическим сдвигом* строки t, начинающимся в позиции i, будем называть циклическую подстроку длины n, начинающуюся в этой позиции.

Для строки, заканчивающейся символом \$, лексикографический порядок циклических сдвигов совпадает с лексикографическим порядком суффиксов, поскольку при сравнении любой пары циклических сдвигов первое несовпадение символов случится не позже первого встреченного символа \$. Вместо суффиксов дальше будем работать с циклическими сдвигами.

Алгоритм сортировки циклических сдвигов будет состоять из $\lceil \log n \rceil + 1$ шагов: на нулевом шаге отсортируем все циклические подстроки длины 1, на первом — длины 2, на k-м — длины 2^k. При $2^k \geqslant n$ отсортированными окажутся циклические сдвиги.

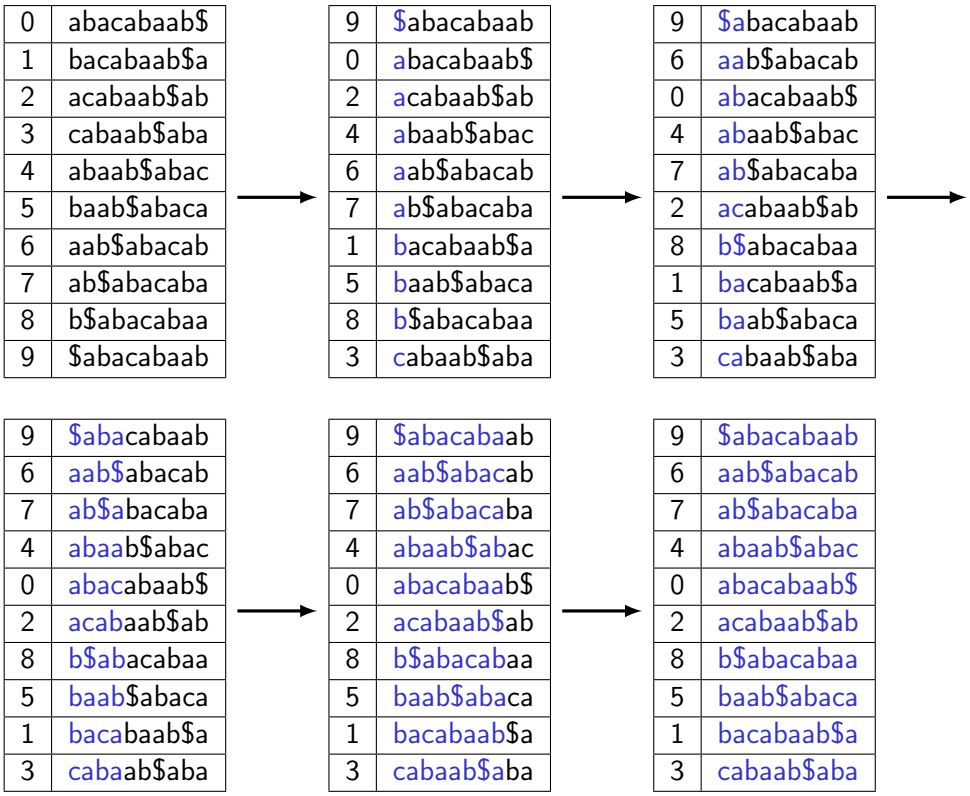

Рис. 5.2: Сортировка циклических сдвигов строки “abacabaab\$”

На нулевом шаге нужно просто отсортировать все символы строки. Это, в зависимости от размера алфавита, можно сделать за $O(n \log n)$ либо за $O(n + |\Sigma|)$ сортировкой подсчётом. После этого пройдём по отсортированному списку циклических подстрок длины один и разобьём их на классы эквивалентности — группы равных. Пронумеруем классы эквивалентности в порядке возрастания их представителей. Пусть $c[i]$ — номер класса эквивалентности $t[i, i+1)$.

На k-м шаге у нас есть p — отсортированный список циклических подстрок длины 2^{k-1}, также для каждой такой циклической подстроки $t[i, i+2^{k-1})$ известен $c[i]$ — номер её класса эквивалентности. При этом $t[i, i+2^{k-1}) < t[j, j+2^{k-1})$ тогда и только тогда, когда $c[i] < c[j]$. Тогда для того, чтобы отсортировать циклические подстроки длины 2^k, достаточно отсортировать пары $(c[i], c[(i+2^{k-1}) \bmod n])$. Пусть $q[i] = (p[i] - 2^{k-1}) \bmod n$,

тогда q — это список пар, отсортированных по вторым элементам. Остаётся стабильно отсортировать этот список по первым элементам, что можно сделать за $O(n)$ сортировкой подсчётом; после чего пройти по отсортированному списку и найти новые классы эквивалентности.

```
# для удобства считаем, что символы - числа от 0 до n - 1
vector<int> cnt(n), c(n), p(n)
fill(cnt, 0)
for i = 0..(n - 1): # сортируем символы подсчётом и записываем в p
  cnt[t[i]] += 1
for i = 1..(n - 1):
  cnt[i] += cnt[i - 1]
for i = (n - 1)..0:
  cnt[t[i]] -= 1
  p[cnt[t[i]]] = i
  c[i] = t[i] # в качестве номеров классов эквивалентности
              # символов можно использовать сами символы

for (len = 1; len < n; len *= 2): # сортируем подстроки длины 2 * len
  vector<int> q(n), cNew(n)
  for i = 0..(n - 1): # сортируем по второй половине
    q[i] = (p[i] - len) % n # в C++ надо быть аккуратнее
  fill(cnt, 0)
  for i = 0..(n - 1): # стабильно сортируем по первой половине
    cnt[c[i]] += 1
  for i = 1..(n - 1):
    cnt[i] += cnt[i - 1]
  for i = (n - 1)..0:
    cnt[c[q[i]]] -= 1
    p[cnt[c[q[i]]]] = q[i]
  cNew[p[0]] = 0 # вычисляем новые классы эквивалентности
  for i = 1..(n - 1):
    a = p[i], b = p[i - 1]
    cNew[p[i]] = cNew[p[i - 1]]
    if c[a] != c[b] or c[(a + len) % n] != c[(b + len) % n]:
      cNew[p[i]] += 1
  swap(c, cNew)
```

R Заметим, что суффиксный массив является, в том числе, отсортированным списком символов строки, поэтому в случае произвольного алфавита построить суффиксный массив быстрее, чем за $\Theta(n \log n)$, не получится. Существуют алгоритмы, позволяющие строить суффиксный массив за линейное время в предположении $|\Sigma| = O(n)$; описание одного из них можно найти в следующем разделе (в список вопросов на экзамене этот алгоритм не войдёт).

5.2 Поиск подстроки в строке с помощью суффиксного массива

Построим суффиксный массив на тексте t. Заметим, что все вхождения шаблона s в t являются префиксами суффиксов t. Поэтому для того, чтобы найти все вхождения, достаточно найти все суффиксы t, чьи префиксы длины $|s|$ совпадают с s. Такие суффиксы образуют подотрезок суффиксного массива. Левая граница этого (возможно, пустого) отрезка — минимальное i такое, что $suf(t, p[i]) \geqslant s$. Правая граница — максимальное i такое, что $t[p[i], \min(n, p[i] + |s|)) \leqslant s$. Обе границы можно найти двоичным поиском по суффиксному массиву.

Если сравнивать строки посимвольно, получаем время работы $O(|s| \log |t|)$ (если нужно выписать все вхождения, а не какое-то одно, то $O(|s| \log |t| + k)$, где k — количество

вхождений). Можно сравнивать строки за $O(\log |s|)$ (с некоторой вероятностью ошибки) с помощью полиномиальных хешей — тогда получаем время работы $O(|s| + \log |s| \log |t|)$.

5.3 Массив lcp и его вычисление за линейное время

Пусть p — суффиксный массив строки t длины n, p^{-1} — обратная к нему перестановка, то есть $p^{-1}[i]$ — позиция $suf(t, i)$ в суффиксном массиве.

За $lcp(a, b)$ будем обозначать длину *наибольшего общего префикса (largest common prefix)* строк a и b. Для краткости будем в аргументах lcp вместо суффикса строки t писать его номер: $lcp(a, i) = lcp(a, suf(t, i))$. Предподсчитаем для всех пар соседних суффиксов в суффиксном массиве длину их наибольшего общего префикса:

$$lcp[i] = lcp(p[i], p[i + 1]), \quad 0 \leqslant i < n - 1.$$

Будем для удобства считать, что $lcp[n - 1] = 0$.

p		lcp
6	aab	1
7	ab	2
4	abaab	3
0	abacabaab	1
2	acabaab	0
8	b	1
5	baab	2
1	bacabaab	0
3	cabaab	0

Рис. 5.3: Суффиксный массив и массив lcp строки “abacabaab”

Массив lcp оказывается полезным при решении множества различных задач; в том числе, он позволяет быстрее решать задачу поиска подстроки в строке.

Вычисление значений lcp по определению потребует $O(n^2)$ времени. Следующий алгоритм (Kasai, Lee, Arimura, Arikawa, Park, 2001, [18]) позволяет вычислить массив lcp за $O(n)$ (если суффиксный массив уже построен): будем вычислять значения lcp в порядке уменьшения длины суффиксов, то есть в порядке $lcp[p^{-1}[0]], lcp[p^{-1}[1]], \ldots, lcp[p^{-1}[n-1]]$. Обозначим $next[i] = p[p^{-1}[i] + 1]$ (не определено, если $p^{-1}[i] = n - 1$), тогда, если $p^{-1}[i] \neq n - 1$, то

$$lcp[p^{-1}[i]] = lcp(i, next[i]).$$

Лемма 5.3.1 Пусть $0 \leqslant i < n - 1$, тогда

$$lcp[p^{-1}[i + 1]] \geqslant lcp[p^{-1}[i]] - 1.$$

Доказательство. Пусть $k = lcp[p^{-1}[i]]$. Можно считать, что $k > 1$ (иначе доказывать нечего). Первые k символов $suf(t, i)$ и $suf(t, next[i])$ совпадают. Поскольку $k > 0$, суффиксы, полученные из этих удалением первого символа, будут идти в том же лексикографическом порядке и будут иметь общий префикс длины $k - 1$: $suf(t, i + 1) < suf(t, next[i] + 1)$ и

$$lcp(i + 1, next[i] + 1) \geqslant k - 1 > 0.$$

В частности, отсюда следует, что $p^{-1}[i + 1] \neq n - 1$, то есть $next[i + 1]$ определено. Заметим, что $suf(t, i + 1) < suf(t, next[i + 1]) \leqslant suf(t, next[i] + 1)$, так как $next[i + 1]$ идёт в суффиксном массиве сразу за $i + 1$. Тогда

$$lcp[p^{-1}[i + 1]] = lcp(i + 1, next[i + 1]) \geqslant lcp(i + 1, next[i] + 1) = k - 1.$$

■

Таким образом, при вычислении $lcp[p^{-1}[i+1]]$ можно не проверять на равенство первые $\max(0, lcp[p^{-1}[i]]-1)$ символов суффиксов. Тогда при вычислении $lcp[p^{-1}[i+1]]$ мы сделаем не более $lcp[p^{-1}[i+1]] - lcp[p^{-1}[i]] + 1$ сравнений символов, то есть на вычисление всего массива lcp мы потратим $O(lcp[p^{-1}[n-1]] - lcp[p^{-1}[0]] + n) = O(n)$ времени.

```
for i = 0..(n - 1):
  pinv[p[i]] = i
k = 0
for i = 0..(n - 1):
  if pinv[i] == n - 1:
    lcp[n - 1] = k = 0
    continue
  j = p[pinv[i] + 1]
  while max(i + k, j + k) < n and t[i + k] == t[j + k]:
    k += 1
  lcp[pinv[i]] = k
  k = max(0, k - 1)
```

Заметим, что

$$lcp(p[i], p[j]) = \min(lcp[i], lcp[i+1], \ldots, lcp[j-1]).$$

Если построить на массиве lcp структуру RMQ, то мы сможем быстро узнавать длину наибольшего общего префикса любых двух суффиксов t.

5.4 Поиск подстроки в строке с помощью суффиксного массива и lcp

Снова будем делать двоичный поиск по суффиксному массиву. Пусть, для определённости, мы ищем минимальный суффикс, больше или равный s (поиск правой границы отрезка вхождений осуществляется аналогично).

Начнём с того, что сравним s с $suf(t, p[0])$ и $suf(t, p[n-1])$ за $O(|s|)$, в процессе вычислив $lcp(s, p[0])$ и $lcp(s, p[n-1])$. Если $lcp(s, p[0]) = |s|$, то интересующий нас суффикс — $p[0]$; если $lcp(s, p[0]) < |s|$ и $s < suf(t, p[0])$, то вхождений s в t нет; то же верно, если $s > suf(t, p[n-1])$.

Пусть L, R — текущие границы двоичного поиска (вначале $L = 0$, $R = n-1$), известны $l = lcp(s, p[L])$ и $r = lcp(s, p[R])$. При этом выполняются условия

$$l < |s|,\ s > suf(t, p[L]),\ s \leqslant suf(t, p[R]).$$

Пусть $M = \lfloor \frac{L+R}{2} \rfloor$. Мы хотим сдвинуть одну из границ так, чтобы условия на границы всё ещё выполнялись.

Рассмотрим сначала случай $l \geqslant r$. Обозначим $k = lcp(p[L], p[M])$.

- Если $k < l$, то $s[0, k) = t[p[L], p[L]+k) = t[p[M], p[M]+k)$, но $s[k] = t[p[L]+k] < t[p[M]+k]$, откуда $s < suf(t, p[M])$. Значит, можно сдвинуть правую границу: $R = M$.
- Пусть теперь $k \geqslant l$. Тогда $lcp(s, p[M]) \geqslant l$. Найдём $lcp(s, p[M])$, проверяя символы на совпадение, пропустив первые l (они точно совпадают). Если $lcp(s, p[M]) = |s|$, то можно сдвинуть правую границу: $R = M$. Если $lcp(s, p[M]) < |s|$, то либо $s < suf(t, p[M])$ (тогда сдвинем правую границу), либо $s > suf(t, p[M])$ (тогда сдвинем левую границу).

Случай $l < r$ разбирается похожим образом: обозначим $k = lcp(p[R], p[M])$.

- Если $k < r$, то $lcp(s, p[M]) \leqslant k < |s|$ и $s > suf(t, p[M])$, то есть можно сдвинуть левую границу.
- Если $k \geqslant r$, то $lcp(s, p[M]) \geqslant r$. Тогда найдём $lcp(s, p[M])$, пропустив первые r сравнений символов, после чего сдвинем одну из границ по тем же правилам, что и в случае $l \geqslant r$.

В конце $R - L = 1$, при этом $suf(t, p[L]) < s \leqslant suf(t, p[R])$, кроме того, известно значение $r = lcp(s, p[R])$. Если $r < |s|$, то вхождений s в t нет, если $r = |s|$, то искомый суффикс — $p[R]$.

Оценка времени работы

Заметим, что $\max(l, r)$ в ходе работы алгоритма не уменьшается. При этом, если во время поиска $lcp(s, p[M])$ случилось q успешных сравнений символов, то $\max(l, r)$ на следующем шаге увеличится на q (так как M станет одной из границ). Значит, за всё время работы алгоритма было проведено $O(|s|)$ удачных сравнений символов, плюс не более одного неудачного на каждом шаге двоичного поиска. Получаем время работы $O(|s| + \log|t|)$ (при условии, что используется либо структура RMQ, умеющая отвечать на запросы за $O(1)$, либо дерево отрезков, к которому применена оптимизация двоичного поиска со вложенным спуском по дереву).

Поиск множества подстрок в строке

Поскольку мы не проводим никакой предварительной обработки шаблона, тот же алгоритм можно применять для поиска сразу множества слов из словаря в тексте: получим время работы $O(|t| + \sum_i |s_i| + m \log|t|)$, если словарь состоит из слов $s_1, \ldots, s_m$; или $O(|t| + \sum_i |s_i| + m \log|t| + k)$, где k — суммарное число вхождений, если нужно выписать все вхождения каждого слова, а не какое-нибудь одно. При этом используется $O(|t|)$ дополнительной памяти.

Преимущество перед алгоритмом Ахо-Корасик в том, что на запросы к фиксированному тексту можно отвечать "онлайн": можно добавлять в словарь новые слова и сразу же искать их в тексте; алгоритму Ахо-Корасик же при добавлении новых слов потребуется заново пройти по всему тексту за $O(|t|)$. Недостаток — в дополнительном слагаемом $m \log|t|$ в оценке времени работы.

R Если помимо массива lcp хранить ещё один массив с некоторой информацией о суффиксах, от этого слагаемого можно избавиться, улучшив время поиска одного шаблона s в тексте до $O(|s|)$.

5.5 Количество различных подстрок

Зная массив lcp, несложно посчитать количество различных подстрок в тексте t: любая подстрока является префиксом некоторого суффикса t; префиксы суффикса $suf(t, p[i])$, которые не встречались в предыдущих суффиксах (в порядке суффиксного массива) — это все префиксы длины больше $lcp[i-1]$ (или просто все префиксы при $i = 0$).

Тогда всего различных подстрок (положительной длины) в тексте

$$n - p[0] + \sum_{i=1}^{n-1}(n - p[i] - lcp[i-1]) = \sum_{i=1}^{n} i - \sum_{i=1}^{n-1} lcp[i-1] = \frac{n(n+1)}{2} - \sum_{i=0}^{n-2} lcp[i].$$

5.6 Наибольшая общая подстрока

Мы умеем решать задачу поиска наибольшей общей подстроки двух строк за $O(n \log n)$ с помощью полиномиальных хешей (с некоторой вероятностью ошибки). С помощью суффиксного массива эту задачу можно решить за линейное время.

Пусть мы хотим найти наибольшую общую подстроку строк s и t. Построим суффиксный массив p и массив lcp на строке $w = s\$_1 t\$_2$. Наибольшая подстрока s и t является общим префиксом двух суффиксов $suf(w, i)$ и $suf(w, j)$ для некоторых $i < |s|$, $j > |s|$. Значит, нам нужно найти такие $i < |s| < j$, что $lcp(suf(w, i), suf(w, j))$ максимально.

Пусть этот максимум достигается на $p[a] = i < |s| < j = p[b]$, при этом $|a - b|$ минимально возможное. Тогда $|a - b| = 1$: если это не так, то одно из $p[a]$, $p[b]$ можно заменить на $p[\lfloor (a + b)/2 \rfloor]$, так что lcp не уменьшится. Значит, ответ хранится в $lcp[\min(a, b)]$.

Получаем следующий алгоритм: нужно взять максимум из $lcp[i]$ по всем i таким, что $p[i]$ и $p[i + 1]$ соответствуют разным строкам (то есть одно из них меньше $|s|$, а другое больше). Время работы — $O(|s| + |t|)$.

5.7 Преобразование Барроуза-Уилера

Пусть, как обычно, t — строка длины n, заканчивающаяся символом \$, который строго меньше остальных символов строки. Будем рассматривать циклические сдвиги t в лексикографическом порядке, и выписывать из каждого последний символ. Полученную строку обозначим за $bwt(t)$; она представляет собой перестановку символов строки t. Описанный алгоритм называют *преобразованием Барроуза-Уилера (Burrows-Wheeler transform, BWT)* строки t.

9	\$abacabaab
6	aab\$abacab
7	ab\$abacaba
4	abaab\$abac
0	abacabaab\$
2	acabaab\$ab
8	b\$abacaba
5	baab\$abaca
1	bacabaab\$a
3	cabaab\$aba

Рис. 5.4: “bbac\$baaaa” — результат преобразования Барроуза-Уилера строки “abacabaab\$”

Для строки t, заканчивающейся на \$, порядок на циклических сдвигах совпадает с порядком на суффиксах. Поэтому, имея суффиксный массив p строки t, несложно вычислить преобразование Барроуза-Уилера за линейное время:

$$bwt(t)[i] = \begin{cases} t[p[i] - 1] & \text{, если } p[i] > 0, \\ \$ & \text{, если } p[i] = 0. \end{cases}$$

Преобразование Барроуза-Уилера является обратимым: по $bwt(t)$ можно восстановить t за линейное от длины t время. Другое полезное свойство BWT состоит в том, что в $bwt(t)$, как правило, большие группы одинаковых символов идут подряд: например, если в тексте t часто встречалась подстрока “the” (что типично для текстов на английском языке), то многие из циклических сдвигов, начинающихся на “he”, будут заканчиваться

на “t”, поэтому в соответствующих позициях $bwt(t)$ будет много подотрезков из подряд идущих символов “t”.

Благодаря этим свойствам BWT применяют в алгоритмах сжатия (например, в утилите bzip2 в UNIX): вместо строки t сжимают $bwt(t)$. Многие алгоритмы сжатия зачастую оказываются более эффективны на строке $bwt(t)$, чем на исходной строке, так как группы из одинаковых символов сжимать несложно (существуют и более формальные рассуждения, которые мы приводить не будем).

Обратное преобразование

Как восстановить по $bwt(t)$ строку t? Пусть $q[i]$ — перестановка, которой соответствует $bwt(t)$: $q[i] = p[i] - 1$ при $p[i] > 0$, и $q[i] = n - 1$ иначе.

Пусть $t[i] = t[j]$ для некоторых $i \neq j$. i встречается в p раньше j тогда и только тогда, когда $suf(t, i) < suf(t, j)$. Поскольку $t[i] = t[j]$, это равносильно тому, что $suf(t, i+1) < suf(t, j+1)$, то есть тому, что i встречается раньше j в q. Значит, позиции вхождений любого символа в строку t идут в p и q в одном и том же порядке.

Тогда, зная $q[i]$, легко восстановить $LF[i]$ (last to first) — позицию, в которой $q[i]$ находится в суффиксном массиве, то есть такую, что $p[LF[i]] = q[i]$. В p первые символы суффиксов находятся в отсортированном порядке. Пусть $t[q[i]] = bwt(t)[i] = a$, тогда раньше $q[i]$ в p будут все вхождения символов, меньших a, а также столько вхождений символа a, сколько раз он встречается в $bwt(t)[0, i)$ (так как позиции вхождений символа a идут в p и q в одном порядке). Итак, $LF[i] = c[a] + left[i]$, где $c[a]$ — количество символов строки t, меньших a, а $left[i]$ — количество вхождений $a = t[q[i]]$ в $bwt(t)[0, i)$. Массивы c и $left$ предподсчитаем сортировкой подсчётом символов строки $bwt(t)$.

№	p		q	c	$left$	LF
0	9	$abacabaab	8	6	0	6
1	6	aab$abacab	5	6	1	7
2	7	ab$abacaba	6	1	0	1
3	4	abaab$abac	3	9	0	9
4	0	abacabaab$	9	0	0	0
5	2	acabaab$ab	1	6	2	8
6	8	b$abacabaa	7	1	1	2
7	5	baab$abaca	4	1	2	3
8	1	bacabaab$a	0	1	3	4
9	3	cabaab$aba	2	1	4	5

С помощью массива LF уже несложно восстановить строку t: пусть r_i — позиция, в которой i находится в суффиксном массиве, то есть такая, что $p[r_i] = i$. Всегда верно, что $r_{n-1} = 0$. Из определения массива LF следует, что $r_i = LF[r_{i+1}]$ при $i < n - 1$. Наконец, $t[n-1] = \$$; $t[i] = bwt(t)[r_{i+1}]$ при $i < n - 1$.

```
# для удобства считаем, что символы - числа от 0 до n - 1
inverseBWT(s, n): # bwt(t) = s, |s| = |t| = n
  vector<int> c(n), left(n), LF(n)
  for i = 0..(n - 1):
    left[i] = c[s[i] + 1]
    c[s[i] + 1] += 1
  for i = 1..(n - 1):
    c[i] += c[i - 1]
  for i = 0..(n - 1):
    LF[i] = left[i] + c[i]
  vector<int> t(n)
```

```
12    r = 0, t[n - 1] = 0 # t[n - 1] = $ (0 - это $)
13    for i = (n - 2)..0:
14      t[i] = s[r]
15      r = LF[r]
16    return t
```

Поиск подстроки в строке с помощью BWT

Если размер алфавита небольшой, то с помощью $bwt(t)$ и суффиксного массива на строке t можно искать подстроку s в тексте t за $O(|s|)$. Предподсчитаем для каждой позиции i в $bwt(t)$ и каждого символа алфавита a величину $LFC(a,i) = c[a] + left'(a,i)$, где $left'(a,i)$ — количество символов, равных a, в $bwt(t)[0,i)$. Это потребует $O(n \cdot |\Sigma|)$ времени и дополнительной памяти.

Теперь будем поочерёдно искать вхождения всех суффиксов строки s в порядке увеличения их длины. Пусть $s[|s|-1] = a$, тогда вхождения $suf(s, |s|-1)$ соответствуют подотрезку суффиксного массива с границами $l = c[a]$, $r = c[a+1]$ (правая граница не включена).

Далее, пусть вхождения $suf(s, k+1)$ соответствуют подоотрезку суффиксного массива с границами l', r'. Тогда вхождения $suf(s,k)$ соответствуют подотрезку суффиксного массива с границами $l = LFC(s[k], l')$ и $r = LFC(s[k], r')$.

Почему это так? Пусть $i \in [l', r')$, тогда $btw(t)[i] = s[k]$ равносильно тому, что $suf(s,k)$ является префиксом $suf(t, p[i]-1)$ ($p[i] > 0$, поскольку \$ не встречается в строке s); все вхождения $suf(s,k)$ в t соответствуют таким суффиксам t. Но такие суффиксы t лежат в суффиксном массиве ровно в позициях $[l, r)$.

R Модификация этого метода позволяет осуществлять поиск подстроки прямо в сжатом тексте t, не восстанавливая его. $bwt(t)$ делится на небольшие блоки, каждый из которых сжимается отдельно, и в каждом из которых дополнительно хранится информация о количестве вхождений каждого символа во все предыдущие блоки. Тогда каждый шаг алгоритма требует обращения лишь к двум блокам, значения LFC вычисляются при помощи линейного прохода по текущему блоку, а также информации о предыдущих блоках. Похожим образом по значениям l, r в конце восстанавливаются значения суффиксного массива на подотрезке $[l, r)$.

Такой способ имеет достаточно большую константу времени работы, но позволяет использовать меньше памяти (даже меньше, чем занимает сам текст), что бывает важно, когда текст очень большой.

6. Факультативное чтение

6.1 Построение суффиксного массива за $O(n)$: алгоритм SA-IS

Будем считать, что $|\Sigma| = O(n)$; в частности, символы строки можно отсортировать сортировкой подсчётом и перенумеровать их числами от 0 до $n-1$.

Мы изучим алгоритм SA-IS (Nong, Zhang, Chan, 2009, [22]), строящий суффиксный массив за линейное время (при указанных выше ограничениях на алфавит). Этот алгоритм, в отличие от большинства более ранних линейных алгоритмов построения суффиксного массива, одновременно имеет относительно простую реализацию и эффективен на практике.

Будем, как обычно, работать со строкой t длины n, последний символ которой равен $\$$ — символу, строго меньшему любого другого символа строки.

S-суффиксы и L-суффиксы

Определение 6.1.1 Будем говорить, что суффикс $suf(t,i)$, $0 \leqslant i < n-1$, имеет *тип S*, если $suf(t,i) < suf(t,i+1)$, и *тип L*, если $suf(t,i) > suf(t,i+1)$. Будем считать, что последний суффикс $suf(t,n-1)$ имеет тип S.

Символ строки $t[i]$ имеет тот же тип, что и суффикс $suf(t,i)$.

Определить тип каждого суффикса/символа строки можно одним проходом по строке справа налево, если воспользоваться следующим правилом: пусть $0 \leqslant i < n-1$, тогда

- $t[i]$ имеет тип S, если $t[i] < t[i+1]$;
- $t[i]$ имеет тип L, если $t[i] > t[i+1]$;
- $t[i]$ имеет тот же тип, что и $t[i+1]$, если $t[i] = t[i+1]$

(в последнем случае результат сравнения $suf(t,i)$ и $suf(t,i+1)$ совпадает с результатом сравнения $suf(t,i+1)$ и $suf(t,i+2)$).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	c	c	b	c	b	c	b	c	b	a	b	b	c	$
S	L	L	S	L	S	L	S	L	L	S	S	S	L	S

Рис. 6.1: Типы символов строки “accbcbcbcbabbc$”

Лемма 6.1.1 Пусть $suf(t,i)$ имеет тип S, $suf(t,j)$ имеет тип L, $t[i] = t[j]$. Тогда $suf(t,i) > suf(t,j)$.

Доказательство. Обозначим $a = t[i] = t[j]$. Найдём минимальное такое $k > 0$, что $t[i+k] \neq a$ или $t[j+k] \neq a$ (такое k точно найдётся, поскольку $\$$ — уникальный символ в конце строки). Заметим, что из того, что $t[i, i+k) = \overline{a \ldots a}$, а $suf(t,i)$ имеет тип S, следует, что $t[i+k] \geqslant a$. Точно так же из того, что $t[j, j+k) = \overline{a \ldots a}$, а $suf(t,j)$ имеет тип L, следует, что $t[j+k] \leqslant a$. Итак, $t[i, i+k) = t[j, j+k)$, $t[j+k] \leqslant a \leqslant t[i+k]$, при этом хотя бы одно из неравенств строгое. Значит, $suf(t,j) < suf(t,i)$. ■

Будущий суффиксный массив можно поделить на блоки, в которых все суффиксы начинаются с одного символа. Из леммы следует, что каждый из этих блоков можно поделить ещё на два: все L-суффиксы будут идти раньше всех S-суффиксов, начинающихся с того же символа; будем называть эти подблоки L-блоком и S-блоком соответствующего символа. Сортировкой подсчётом можно за линейное время вычислить размер каждого из блоков, а также его границы в суффиксном массиве.

LMS-суффиксы, "разделяй и властвуй"

Определение 6.1.2 Будем говорить, что суффикс $suf(t, i)$ (или символ строки $t[i]$), $1 \leqslant i < n$, имеет *тип LMS ("leftmost S")*, если он имеет тип S, а $t[i-1]$ имеет тип L.

На рис. 6.1 тип LMS имеют символы с номерами 3, 5, 7, 10, 14.

Заметим, что количество LMS-суффиксов не превышает $\lfloor \frac{n}{2} \rfloor$, так как никакие два соседних суффикса не могут одновременно иметь тип LMS, $suf(t, 0)$ не имеет тип LMS. Мы воспользуемся подходом "разделяй и властвуй": сначала отсортируем отдельно суффиксы типа LMS (для этого мы построим вспомогательную строку и запустим алгоритм на ней рекурсивно; обсудим это подробнее позже), после чего с помощью *индуцированной сортировки (induced sorting)* за линейное время отсортируем уже все суффиксы, пользуясь уже известным лексикографическим порядком на LMS-суффиксах.

Индуцированная сортировка

Пусть нам уже известен p_1 — отсортированный список всех LMS-суффиксов. С его помощью мы построим суффиксный массив p за линейное время следующим алгоритмом:

0. Изначально заполним все ячейки p значением -1. Сортировкой подсчётом найдём границы L-блока и S-блока каждого символа.
1. Сложим LMS-суффиксы в соответствующие им S-блоки, сохраняя лексикографический порядок, заданный p_1, внутри каждого S-блока. **Отметим, что LMS-суффиксы не обязательно попали в ячейку суффиксного массива, в которой они окажутся в конце.**
2. Пусть $pos[a]$ указывает на первый элемент L-блока символа a. Пройдём по массиву p слева направо, и каждый раз, когда встречаем $p[i] \neq -1$ такое, что $suf(t, p[i]-1)$ имеет тип L, присвоим $p[i]-1$ в $p[pos[t[p[i]-1]]]$, после чего увеличим $pos[t[p[i]-1]]$ на единицу.
3. Пусть $pos[a]$ указывает на последний элемент S-блока символа a. Пройдём по массиву p справа налево, и каждый раз, когда встречаем $p[i]$ такое, что $suf(t, p[i]-1)$ имеет тип S, присвоим $p[i]-1$ в $p[pos[t[p[i]-1]]]$, после чего уменьшим $pos[t[p[i]-1]]$ на единицу.

Предложение 6.1.2 После второго шага все L-суффиксы окажутся на корректных позициях в массиве p.

Доказательство. Покажем сначала, что каждый L-суффикс попадёт в какую-то ячейку p. Для этого заметим, что для любого L-суффикса $suf(t, j)$ найдётся такое $k > 0$, что $suf(t, j+q)$ имеет тип L для $0 \leqslant q < k$, $suf(t, j+k)$ имеет тип LMS. При этом

$$t[j] \geqslant t[j+1] \geqslant \cdots \geqslant t[j+k-1] > t[j+k].$$

В тот момент, когда мы посетим $p[i] = j+k$, мы поместим $j+k-1$ в L-блок, находящийся строго правее позиции i. Значит, после этого мы посетим $p[i] = j+k-1$, и поместим $j+k-2$ в тот же L-блок, но правее, либо в один из следующих L-блоков. Аналогично, для

всех $0 < q < k$ мы посетим $p[i] = j + q$, и в этот момент поместим $j + q - 1$ в позицию правее i.

Теперь предположим, что нашлись такие два L-суффикса $suf(t, i) > suf(t, j)$, что i находится левее j в массиве p. Из всех таких пар рассмотрим пару с максимальным $\min(i, j)$ (если таких несколько, то любую из них).

Заметим, что $i < n - 1$, $j < n - 1$, так как $suf(t, n - 1)$ имеет тип S. Кроме того, $t[i] = t[j]$, так как суффиксы из разных L-блоков точно лежат в p в корректном порядке. Тогда $suf(t, i + 1) > suf(t, j + 1)$.

Если $suf(t, i + 1)$ и $suf(t, j + 1)$ оба имеют тип S, то они имеют тип LMS. Тогда $j + 1$ находится левее $i + 1$ в массиве p. То же верно, если один из суффиксов имеет тип LMS, а другой — тип L (так как суффиксы из разных блоков всегда лежат в p в корректном порядке). Наконец, то же верно, если оба суффикса имеют тип L, так как $\min(i + 1, j + 1) > \min(i, j)$.

Итак, $j + 1$ находится левее $i + 1$ в p. Но тогда при проходе по массиву p мы встретили $j + 1$ раньше $i + 1$, то есть положили j в p раньше, чем i. Это противоречит тому, что i находится левее j в p, значит наше предположение неверно, и все пары L-суффиксов лежат в p в корректном порядке. ■

Предложение 6.1.3 После третьего шага все S-суффиксы окажутся на корректных позициях в массиве p.

Доказательство. Доказательство аналогично доказательству предыдущего предложения. Отметим лишь, что $suf(t, n - 1)$ уже оказался на корректной позиции на первом шаге, а для любого другого S-суффикса $suf(t, j)$ найдётся такое $k > 0$, что $suf(t, j + q)$ имеет тип S для $0 \leqslant q < k$, $suf(t, j + k)$ имеет тип L. При этом все L-суффиксы уже находятся на корректных позициях после второго шага. ■

LMS-подстроки и вспомогательная строка

Для того, чтобы отсортировать LMS-суффиксы, построим вспомогательную строку, на которой запустим алгоритм рекурсивно.

Определение 6.1.3 Пусть $0 < i < n - 1$, $t[i]$ — LMS-символ, $t[j]$ — следующий за ним LMS-символ. Будем называть подстроку $t[i, j]$ для таких i, j *LMS-подстрокой*. Также LMS-подстрокой будем называть подстроку $t[n - 1, n - 1]$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S	L	L	S	L	S	L	S	L	L	S	S	S	L	S
a	c	c	b	c	b	c	b	c	b	a	b	b	c	$
			b	c	b									
					b	c	b							
							b	c	b	a				
										a	b	b	c	$
														$

Рис. 6.2: Строка “accbcbcbcbabbc$” и её LMS-подстроки (LMS-символы выделены цветом)

Пусть всего в строке t есть n_1 LMS-символов. Позицию i-го слева LMS-символа (нумерация с нуля) будем обозначать за i'. Тогда LMS-суффиксы имеют вид $suf(t, i')$, $0 \leqslant i < n_1$; LMS-подстроки имеют вид $t[i', (i + 1)']$, $0 \leqslant i < n_1$ (считаем $n_1' = (n_1 - 1)'$).

Введём новое отношение порядка $\prec$ на подстроках t, отличающееся от обычного лексикографического порядка тем, что при равенстве символы сравниваются по своему

типу (при этом считаем, что тип S больше типа L). Так, на рис. 6.2 $t[3,5] < t[7,10]$, но $t[3,5] \succ t[7,10]$, поскольку $t[5]$ имеет тип S, а $t[9]$ имеет тип L.

Отсортируем LMS-подстроки, пользуясь отношением $\prec$, после чего сопоставим каждой LMS-подстроке номер её класса эквивалентности (классы эквивалентности пронумеруем в порядке возрастания их представителей). Как произвести эту сортировку за линейное время, обсудим чуть позже. Определим вспомогательную строку t_1 длины n_1 следующим образом: $t_1[i]$ — номер класса эквивалентности LMS-подстроки $t[i', (i+1)']$. Так, $t_1 = \overline{33210}$ в примере на рис. 6.2.

Отдельно отметим, что LMS-подстрока $t[n-1, n-1] = \$$ — единственная в минимальном классе эквивалентности, поэтому $t_1[n_1 - 1]$ строго меньше любого другого символа t_1 (то есть является аналогом $\$$ в исходной строке t). Значит, суффиксный массив t_1 можно найти рекурсивным запуском алгоритма.

Как суффиксный массив t_1 поможет отсортировать LMS-суффиксы? Заметим, что существует биекция между суффиксами t_1 и LMS-суффиксами t: $suf(t_1, i)$ соответствует LMS-суффиксу $suf(t, i')$. Например, в примере на рис. 6.2 $suf(t_1, 2)$ соответствует $suf(t, 7)$.

Предложение 6.1.4 Пусть $0 \leqslant i, j < n_1$, $i \neq j$, тогда $suf(t_1, i) < suf(t_1, j)$ тогда и только тогда, когда $suf(t, i') < suf(t, j')$.

Доказательство. Пусть $suf(t_1, i) < suf(t_1, j)$. Пусть $k \geqslant 0$ — минимальное такое, что $t_1[i+k] \neq t_1[j+k]$ (то есть $t_1[i+k] < t_1[j+k]$). Такое k всегда существует, поскольку $t_1[n_1 - 1] = 0$ — единственный символ t_1, равный 0.

Из $t_1[i+k] < t_1[j+k]$, следует, что $t[(i+k)', (i+k+1)'] \prec t[(j+k)', (j+k+1)']$. Пусть $l \geqslant 0$ — минимальное такое, что $t[(i+k)' + l]$ и $t[(j+k)' + l]$ не совпадают или имеют разные типы. Обозначим $(i+k)' + l = c$, $(j+k)' + l = d$.

Заметим, что из $t_1[i, i+k) = t_1[j, j+k)$ и $t[(i+k)', c) = t[(j+k)', d)$ следует, что $t[i', c) = t[j', d)$. Тогда $suf(t, i') < suf(t, j')$ тогда и только тогда, когда $suf(t, c) < suf(t, d)$.

Если $t[c] < t[d]$, то $suf(t, c) < suf(t, d)$. Если же $t[c] = t[d]$, то $t[c]$ имеет тип L, а $t[d]$ имеет тип S, то есть тоже верно $suf(t, c) < suf(t, d)$. ■

Таким образом, если p_1 — суффиксный массив t_1, то массив p'_1 ($p'_1[i] = (p_1[i])'$) — отсортированный список LMS-суффиксов.

LMS-префиксы, индуцированная сортировка LMS-префиксов

Осталось научиться сортировать LMS-подстроки относительно порядка $\prec$ за линейное время. Оказывается, это можно сделать алгоритмом, практически полностью совпадающим с описанной выше индуцированной сортировкой.

Определение 6.1.4 Пусть $0 \leqslant i < n-1$, $j > i$ — минимальное такое, что $t[j]$ — LMS-символ. Тогда $pre(t, i) = t[i, j]$ — *LMS-префикс*. Также будем называть LMS-префиксом $pre(t, n-1) = t[n-1, n-1]$.

Другими словами, если $(j-1)' \leqslant i < j'$, то $pre(t, i) = t[i, j']$. Заметим, что все LMS-подстроки являются LMS-префиксами; любой другой LMS-префикс является суффиксом одной из LMS-подстрок. Будем говорить, что LMS-префикс $pre(t, i)$ имеет тот же тип, что и $t[i]$.

Мы отсортируем относительно порядка $\prec$ не только LMS-подстроки, а вообще все LMS-префиксы следующим алгоритмом:

0. Изначально заполним все ячейки p значением -1. Сортировкой подсчётом найдём границы L-блока и S-блока каждого символа.

1. Сложим LMS-символы в соответствующие им S-блоки, порядок внутри блока значения не имеет. **Как и в прошлый раз, на последнем шаге значения в этих ячейках перезапишутся.**
2. Пусть $pos[a]$ указывает на первый элемент L-блока символа a. Пройдём по массиву p слева направо, и каждый раз, когда встречаем $p[i] \neq -1$ такое, что $pre(t, p[i]-1)$ имеет тип L, присвоим $p[i]-1$ в $p[pos[t[p[i]-1]]]$, после чего увеличим $pos[t[p[i]-1]]$ на единицу.
3. Пусть $pos[a]$ указывает на последний элемент S-блока символа a. Пройдём по массиву p справа налево, и каждый раз, когда встречаем $p[i]$ такое, что $pre(t, p[i]-1)$ имеет тип S, присвоим $p[i]-1$ в $p[pos[t[p[i]-1]]]$, после чего уменьшим $pos[t[p[i]-1]]$ на единицу.

Предложение 6.1.5 После второго шага все L-LMS-префиксы окажутся на корректных позициях в массиве p.

Доказательство. Покажем сначала, что каждый L-LMS-префикс попадёт в какую-то ячейку p. Для этого заметим, что для любого L-LMS-префикса $pre(t, j) = t[j, k']$ $((k-1)' \leqslant j < k')$ верно, что $pre(t, q)$ имеет тип L для $j \leqslant q < k'$, $t[k']$ имеет тип LMS. При этом

$$t[j] \geqslant t[j+1] \geqslant \cdots \geqslant t[k'-1] > t[k'].$$

В тот момент, когда мы посетим $p[i] = k'$, мы поместим $k'-1$ в L-блок, находящийся строго правее позиции i. Значит, после этого мы посетим $p[i] = k'-1$, и поместим $k'-2$ в тот же L-блок, но правее, либо в один из следующих L-блоков. Аналогично, для всех $j < q < k$ мы посетим $p[i] = q$, и в этот момент поместим $q-1$ в позицию правее i.

Теперь предположим, что нашлись такие два L-LMS-префикса $pre(t, i) \succ pre(t, j)$, что i находится левее j в массиве p. Из всех таких пар рассмотрим пару с максимальным $\min(i, j)$ (если таких несколько, то любую из них). Пусть $pre(t, i) = t[i, a']$, $pre(t, j) = t[j, b']$.

Заметим, что $i < n-1$, $j < n-1$, так как $pre(t, n-1)$ имеет тип S. Кроме того, $t[i] = t[j]$, так как LMS-префиксы из разных L-блоков точно лежат в p в корректном порядке. Тогда $t[i+1, a'] > t[j+1, b']$.

Если $t[i+1]$ и $t[j+1]$ оба имеют тип S, то они имеют тип LMS, $a' = i+1$, $b' = j+1$. Тогда $t[i+1] > t[j+1]$, значит, $j+1$ находится левее $i+1$ в массиве p. То же верно, если один из $t[i+1]$, $t[j+1]$ имеет тип LMS, а другой — тип L (так как LMS-символ и L-LMS-префикс принадлежат разным блокам, то есть всегда лежат в p в корректном порядке). Наконец, то же верно, если и $t[i+1]$, и $t[j+1]$ имеют тип L, так как $\min(i+1, j+1) > \min(i, j)$.

Итак, $j+1$ находится левее $i+1$ в p. Но тогда при проходе по массиву p мы встретили $j+1$ раньше $i+1$, то есть положили j в p раньше, чем i. Это противоречит тому, что i находится левее j в p, значит наше предположение неверно, и все пары L-LMS-префиксов лежат в p в корректном порядке. ■

Предложение 6.1.6 После третьего шага все S-LMS-префиксы окажутся на корректных позициях в массиве p.

Доказательство. Доказательство аналогично доказательству предыдущего предложения. Отметим лишь, что $p[0] = n-1$ уже после первого шага алгоритма, а для любого другого S-LMS-префикса $pre(t, j)$ найдётся такое $k > 0$, что $pre(t, j+q)$ имеет тип S для $0 \leqslant q < k$, $pre(t, j+k)$ имеет тип L. При этом все L-LMS-префиксы уже находятся на корректных позициях после второго шага. ■

Остаётся разбить отсортированные LMS-подстроки на классы эквивалентности. Поскольку суммарная длина LMS-подстрок не превышает $2n$ (две соседних LMS-подстроки

пересекаются по одному символу), проверять равенство соседних в отсортированном списке LMS-подстрок можно посимвольно.

Оценка времени работы

Получаем рекуррентное соотношение

$$T(n) = O(n) + T(n_1) \leqslant O(n) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right),$$

откуда $T(n) = O(n)$.

Если все символы t_1 оказываются попарно различны, можно не запускать алгоритм рекурсивно, а просто запустить сортировку подсчётом. Также при маленьких n можно использовать более простые алгоритмы с меньшей константой во времени работы.

Максимально аккуратная реализация алгоритма (подробно о которой мы говорить не будем) использует примерно столько же дополнительной памяти, сколько занимает сам построенный суффиксный массив.

7. Суффиксное дерево

Сжатый бор отличается от обычного бора тем, что на рёбрах разрешается писать строки, а не только символы, но при этом запрещаются вершины с одним исходящим ребром (кроме корня). Строки на рёбрах, исходящих из вершины, по-прежнему должны начинаться с разных символов.

Суффиксное дерево (suffix tree) (Weiner, 1973, [27]) строки t длины n представляет собой сжатый бор на множестве суффиксов строки t. Для того, чтобы каждому суффиксу соответствовал лист бора, обычно предполагают, что последний символ t равен $\$$ — символу, больше не встречающемся в строке.

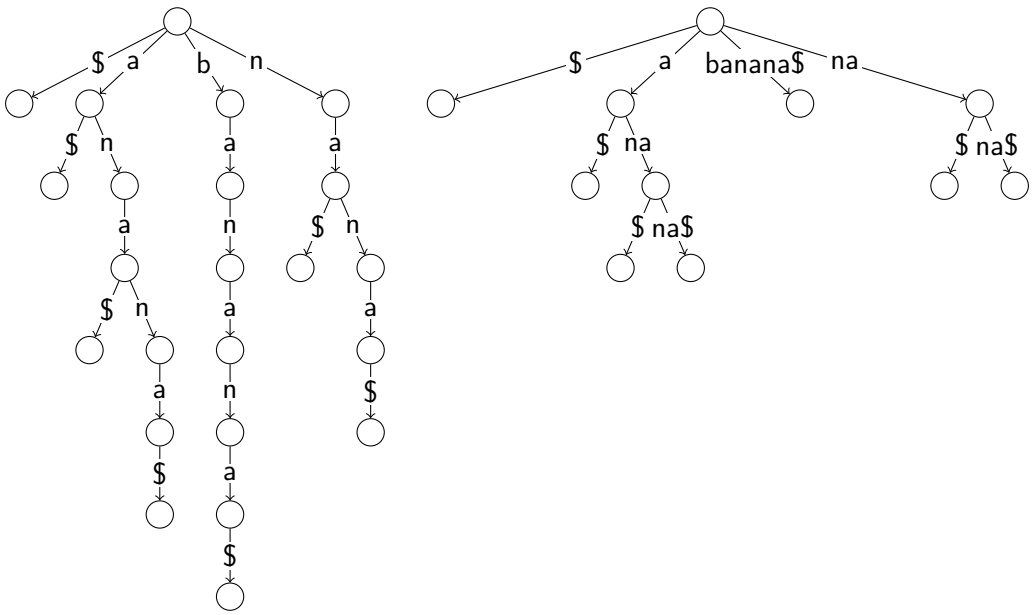

Рис. 7.1: Бор на суффиксах строки “banana$” и суффиксное дерево на той же строке

Суффиксы строки t и листья дерева находятся во взаимно-однозначном соответствии. Значит, поскольку у всех нелистовых вершин исходящая степень равна хотя бы двум, общее число вершин в суффиксном дереве не превосходит $2n - 1$.

При этом для каждого ребра вместо строки, написанной на этом ребре, будем хранить границы подстроки t, которой она соответствует. Полученное суффиксное дерево занимает $O(n)$ памяти (мы по-прежнему считаем размер алфавита константой и не учитываем в асимптотике).

R В случае алфавита константного размера существуют алгоритмы построения суффиксного дерева за $O(n)$ (например, Ukkonen, 1995, [26]). Для произвольного алфавита (как и случае суффиксного массива) строить суффиксное дерево быстрее, чем за $\Theta(n \log n)$, не умеют.

7.1 Связь с суффиксным массивом

Построение суффиксного дерева по суффиксному массиву

Имея суффиксный массив и массив lcp, несложно построить суффиксное дерево за $O(n)$: будем добавлять суффиксы в сжатый бор в лексикографическом порядке, при этом поддерживая ссылку на вершину, соответствующую последнему добавленному суффиксу. При добавлении суффикса $suf(t, p[i+1])$ нужно подняться из текущей вершины до глубины $lcp[i]$. Если при этом мы попали в "середину" ребра, "разрежем" его на два, добавив в "середину" новую вершину. После этого создадим ребро в новый лист, который будет соответствовать суффиксу $p[i+1]$.

На каждом шаге мы создали не более двух новых вершин (и рёбер). При этом по каждому ребру мы прошли не более двух раз: один раз вверх, и один раз вниз. Значит, время работы алгоритма есть $O(n)$.

Построение суффиксного массива по суффиксному дереву

Имея суффиксное дерево, несложно построить суффиксный массив и массив lcp за $O(n)$: будем просто обходить дерево поиском в глубину, перебирая рёбра в лексикографическом порядке. При этом будем поддерживать текущую глубину, а также минимальную глубину, на которую мы поднимались между двумя соседними листьями. Тогда при попадании в очередной лист текущая глубина соответствует длине очередного суффикса, а минимальная глубина на пути от предыдущего суффикса соответствует очередному значению lcp.

Сравнение структур

Алгоритмы, решающие задачи на строках с помощью суффиксного дерева, как правило, идейно устроены проще, чем алгоритмы, решающие ту же задачу с помощью суффиксного массива. Также суффиксному массиву зачастую требуется предподсчёт вспомогательных массивов (например, массива lcp), для того, чтобы решить задачу, которую суффиксное дерево способно решить само по себе.

Недостатком суффиксного дерева является большее количество используемой памяти. Суффиксные массивы появились именно как более легковесный аналог суффиксных деревьев. В 2004 году было доказано (Abouelhoda, Kurtz, Ohlebusch, [3]), что любой (с некоторыми оговорками) алгоритм на суффиксном дереве можно преобразовать в алгоритм на суффиксном массиве с одним или несколькими вспомогательными массивами. При этом новый алгоритм будет иметь то же время работы, но использовать меньше памяти.

7.2 Поиск подстроки в строке

Попытаемся пройти по строке s в суффиксном дереве. Если в какой-то момент по одному из символов не найдётся перехода, вхождений нет. Если удалось пройти по всем символам, то все достижимые из текущей вершины (или ребра, если мы оказались на "середине" ребра) листья соответствуют вхождениям s в t. Таким образом, наличие вхождения проверяется за $O(|s|)$. Количество вхождений тоже можно находить за $O(|s|)$, если предподсчитать для каждой вершины количество достижимых из неё листьев. Все вхождения можно найти, обойдя все достижимые листья. Получаем время работы $O(|s| + k)$, где k — количество вхождений (так как количество посещённых листьев не более чем в два раза больше общего количества посещённых при обходе листьев вершин).

7.3 Количество различных подстрок

Количество различных подстрок строки t равняется суммарной длине рёбер суффиксного дерева (так как каждая подстрока соответствует вершине бора на суффиксах, то есть вершине или "середине" ребра суффиксного дерева).

7.4 Наибольшая общая подстрока

Для того, чтобы найти наибольшую общую подстроку строк s и t, построим суффиксное дерево на строке $s\$_1 t\$_2$. Тогда наибольшая общая подстрока s и t соответствует максимальной по глубине вершине дерева, из которой достижимы листья, соответствующие разным строкам (то есть листья, соответствующие суффиксам длины не больше $|t|+1$ и больше $|t|+2$). Все такие вершины можно пометить поиском в глубину по дереву. Получаем время работы $O(|s|+|t|)$.

7.5 Алгоритм сжатия Зива-Лемпеля

Существует целое семейство алгоритмов сжатия LZ* (Ziv, Lempel и другие), основанных на похожих идеях. Мы изучим одну из базовых версий и её реализацию с помощью суффиксного дерева.

Пусть t — строка длины n, $0 \leqslant i < n$. Обозначим за $prior(i)$ наибольший префикс $suf(t,i)$, который встречается как подстрока левее в строке t (то есть является префиксом $suf(t,j)$ для некоторого $j < i$). Обозначим за $len(i)$ длину $prior(i)$. Если $len(i) > 0$, то за $l(i)$ обозначим позицию самого левого вхождения $prior(i)$ в t.

Пусть мы уже сжали префикс t длины i. Тогда, если $len(i) > 0$, мы можем закодировать следующие $len(i)$ символов строки t парой $(l(i), len(i))$: это соответствует равенству $t[l(i), l(i)+len(i)) = t[i, i+len(i))$, с помощью которого эти символы можно будет восстановить (даже если эти две строки пересекаются). Если $len(i) = 0$, то просто запишем символ $t[i]$ и перейдём к следующей позиции.

Например, в результате применения алгоритма к строке $abacabadabacaba$ получится $ab(0,1)c(0,3)d(0,7)$; к строке $abacababababababa$ — $ab(0,1)c(0,3)(5,8)$. Разумеется, можно заменять символы строки на пару чисел не при всех $len(i) > 0$, а лишь когда $len(i)$ достаточно большое, то есть когда это действительно уменьшает длину строки.

С помощью суффиксного дерева алгоритм можно реализовать за линейное время. Для этого построим суффиксное дерево на строке t, и для каждой вершины дерева v предподсчитаем l_v — номер суффикса $suf(t, l_v)$, соответствующего самому глубокому листу в поддереве v. Все значения l_v можно вычислить за один обход в глубину.

Теперь для того, чтобы найти $len(i)$ и $l(i)$, просто будем идти из корня по символам $suf(t,i)$. Остановимся, если по следующему символу нет перехода, или если при переходе по символу мы попали бы на ребро, ведущее в такую вершину v, что $l_v \geqslant i$. Тогда в момент, когда мы остановились, $len(i)$ — это просто текущая глубина. $l(i)$ — это l_v, где v — текущая вершина (или конец текущего ребра).

III Паросочетания

8. Максимальное паросочетание

Напомним, что *паросочетание (matching)* — множество попарно несмежных рёбер в неориентированном графе. Первая задача, которой мы займёмся — поиск максимального (по размеру) паросочетания.

8.1 Лемма Бержа

Пусть M — некоторое фиксированное паросочетание. Будем называть простой (без повторяющихся вершин) путь в графе *чередующимся (alternating)*, если рёбра, принадлежащие M, чередуются в нём с рёбрами, не принадлежащими M. Будем называть чередующийся путь *дополняющим (augmenting)*, если первая и последняя вершины на пути не покрыты паросочетанием M.

Лемма 8.1.1 — Лемма о дополняющем пути; лемма Бержа [5]. Паросочетание M в графе G является максимальным (по размеру) тогда и только тогда, когда G не содержит дополняющего пути (относительно M).

Доказательство. Пусть в графе есть дополняющий путь P, тогда рассмотрим симметрическую разность M и P как множеств рёбер: $M \oplus P = (P \setminus M) \cup (M \setminus P)$. Заметим, что $|M \oplus P| = |M| + 1$, поскольку в P рёбер, не лежащих в M, было на одно больше, чем лежащих. При этом $M \oplus P$ — паросочетание. Значит, M — не максимальное.

Докажем теперь следствие в обратную сторону: пусть M — не максимальное по размеру паросочетание, то есть существует паросочетание N, $|N| > |M|$. Рассмотрим подграф, индуцированный множеством рёбер $S = M \oplus N$. Степень любой вершины в этом подграфе не превосходит двух, значит, каждая компонента связности этого подграфа является путём либо циклом. При этом все циклы имеют чётную длину (так как рёбра из M и N в них чередуются), значит, найдётся хотя бы один путь $P \subset S$ такой, что в нём рёбер из N больше, чем из M; в P рёбра из M и N тоже чередуются, значит, P — дополняющий. ■

Заметим, что в доказательстве выше описан возможный план действий при поиске максимального паросочетания: пусть мы как-нибудь умеем находить дополняющий путь, либо понимать, что его в графе нет. Тогда будем много раз искать дополняющий путь, и, пока он находится, увеличивать паросочетание "инвертированием" рёбер в найденном пути.

8.2 Поиск максимального паросочетания в двудольном графе

Начиная с этого момента будем работать с двудольными графами.

Пусть $G = (V, E)$ — двудольный граф с долями A, B; M — паросочетание в нём. Рассмотрим ориентированный граф $G' = (V, E')$, где

$$E' = \{a \to b \mid a \in A, b \in B, (a, b) \in E\} \cup \{b \to a \mid a \in A, b \in B, (a, b) \in M\};$$

то есть в G' из A в B ведут все рёбра G, а из B в A — только рёбра паросочетания. Заметим, что существует биекция между простыми путями в G' и чередующимися путями

в G: действительно, каждое второе ребро простого пути P в G' направлено из B в A, то есть соответствует ребру паросочетания; тогда остальные рёбра P в паросочетании не лежат, так как иначе какая-то вершина встретилась бы в пути несколько раз.

Дополняющие пути в G соответствуют простым путям в G', концы которых не покрыты паросочетанием. Искать такой простой путь в G' можно алгоритмом поиска в глубину. Для каждой вершины из A будем хранить список исходящих из неё рёбер. Для вершины $b \in B$ достаточно хранить ссылку на второй конец ребра паросочетания, покрывающего b (если такое ребро есть).

```
# если b - вершина из B, то pair[b] - второй конец ребра паросочетания,
# покрывающего b, либо -1, если такого ребра нет
#
# функция возвращает True, если дополняющий путь нашёлся;
# при этом на обратном ходе рекурсии происходит перестройка паросочетания
dfs(a): # a - вершина из A
  used[a] = True
  for b in es[a]:
    if pair[b] == -1 or (not used[pair[b]] and dfs(pair[b])):
      pair[b] = a
      return True
  return False

# вершины из A пронумерованы числами от 0 до n-1
fill(covered, False)
# n раз пытаемся найти дополняющий путь, запуская поиск из
# всех непокрытых вершин в A
for i = 0..(n - 1):
  for a = 0..(n - 1):
    if not covered[a]:
      fill(used, False)
      if dfs(a):
        covered[a] = True
        break
```

Получившийся алгоритм работает за $O(V^2E)$. Время работы можно улучшить, воспользовавшись следующей леммой:

Лемма 8.2.1 Пусть M — паросочетание; вначале M пустое, после чего M несколько раз увеличивается "инвертированием" рёбер вдоль дополняющего пути. Пусть в какой-то момент оказалось, что вершина $v \in A$ не является концом никакого дополняющего пути. Тогда v уже никогда не будет концом дополняющего пути.

Доказательство. Пусть это не так; рассмотрим ближайший более поздний момент, в который появился дополняющий путь P с концами $v \in A$, $u \in B$; этот путь соответствует простому пути из v в u в графе G'. Пусть $b \to a$ — ближайшее к v ребро на пути P, которого не было в графе G' до последней перестройки паросочетания; $b \in B$, $a \in A$, поскольку в G' меняются только рёбра, ведущие из B в A. Тогда ребро $a \to b$ принадлежало пути Q, вдоль которого происходила последняя перестройка; обозначим концы этого пути за $x \in A$, $y \in B$.

Пусть $P(v, b)$ — часть пути P от v до b; $Q(b, y)$ — часть пути Q от b до y. До последней перестройки существовал дополняющий путь, являющийся конкатенацией $P(v, b)$ и $Q(b, y)$; это противоречит нашему предположению. ■

Таким образом, запускать поиск из каждой вершины доли A достаточно один раз: если дополняющий путь найдётся, то вершина уже всегда будет покрыта паросочетанием; если же он не найдётся, то эта вершина уже никогда не будет концом дополняющего пути. Получаем алгоритм с оценкой времени работы $O(VE)$.

```
for a = 0..(n - 1):
  fill(used, False)
  dfs(a)
```

Оценку можно сделать ещё более точной, если очищать массив пометок в поиске только когда граф перестраивается: получившийся алгоритм работает за $O(|M|E)$, где $|M|$ — размер максимального паросочетания.

```
fill(used, False)
for a = 0..(n - 1):
  if dfs(a):
    fill(used, False)
```

На практике часто пользуются ещё одной оптимизацией: найдём какое-нибудь максимальное **по включению** паросочетание жадным алгоритмом, после чего перестроим его до максимального по размеру, пользуясь вышеописанным алгоритмом. Разумеется, эта оптимизация не улучшает асимптотическую оценку времени работы; тем не менее, на практике она часто даёт очень серьёзное ускорение.

```
fill(covered, False)
for a = 0..(n - 1):
  for b in es[a]:
    if pair[b] == -1:
      pair[b] = a
      covered[a] = True
fill(used, False)
for a = 0..(n - 1):
  if not covered[a] and dfs(a):
    covered[a] = True
    fill(used, False)
```

R Чуть позже мы научимся искать максимальное паросочетание в двудольном графе за время $O(E\sqrt{V})$. В произвольном графе задачу умеют решать за то же время, но соответствующие алгоритмы (например, Micali, Vazirani, 1980, [21]) устроены намного сложнее.

8.3 Поиск минимального вершинного покрытия в двудольном графе

Напомним, что decision-версия задачи о вершинном покрытии в произвольном графе является NP-полной. Тем не менее, задача поиска минимального (по размеру) вершинного покрытия **в двудольном графе** оказывается не сложнее задачи поиска максимального паросочетания.

Теорема 8.3.1 — Теорема Кёнига-Эгервари (König, 1931, [8]; Egerváry, 1931, [11]). В двудольном графе размер максимального паросочетания совпадает с размером минимального вершинного покрытия.

Доказательство. Заметим сначала, что любое вершинное покрытие имеет не меньший размер, чем любое паросочетание, поскольку никакие два ребра паросочетания нельзя покрыть одной вершиной. Значит, достаточно показать, что существует вершинное покрытие того же размера, что и максимальное паросочетание.

Пусть M — максимальное паросочетание, A, B — доли графа. Обозначим за $A^+ \subset A$, $B^+ \subset B$ те вершины, в которые можно попасть, пройдя по чередующемуся пути из не покрытой паросочетанием M вершины доли A (в частности, все не покрытые M

вершины доли A лежат в A^+, так как пустой путь является чередующимся). Обозначим $A^- = A \setminus A^+$, $B^- = B \setminus B^+$ и рассмотрим множество $C = A^- \cup B^+$.

C — вершинное покрытие, поскольку между A^+ и B^- рёбер быть не может по определению чередующегося пути. При этом любая вершина $v \in C$ является концом какого-то ребра из M: для $v \in A^-$ это так, поскольку все непокрытые вершины доли A лежат в A^+; для $v \in B^+$ это так, поскольку M — максимальное паросочетание, а, значит, в графе нет дополняющих путей, и чередующийся путь до v можно продолжить каким-то ребром паросочетания. Наконец, рёбра паросочетания, соответствующие вершинам C, попарно различны: никакие $v \in A^-$, $u \in B^+$ не могут лежать на одном ребре M, так как в этом случае v лежало бы в A^+. Тогда $|C| \leqslant |M|$, то есть $|C| = |M|$, C — минимальное вершинное покрытие. ■

Для того, чтобы найти в двудольном графе минимальное вершинное покрытие, найдём в нём максимальное паросочетание, после чего поиском в глубину найдём множества A^+, A^-, B^+, B^-; из доказательства теоремы мы знаем, что $A^- \cup B^+$ — минимальное вершинное покрытие.

9. Стабильное паросочетание

9.1 Формулировка задачи

Пусть есть n соискателей и n компаний, куда соискатели хотят устроиться на работу. Каждая компания хочет нанять ровно одного соискателя, при этом у каждой компании есть свой *список предпочтений* — список всех соискателей в порядке убывания приоритета (чем выше в списке соискатель, тем больше компания хотела бы его нанять). Каждый соискатель также имеет аналогичный список предпочтений — упорядоченный список всех компаний. Выражаясь в терминах теории графов, дан полный двудольный граф с обеими долями размера n, а также упорядоченный список рёбер в каждой вершине. Будем обозначать отношение, соответствующее вершине a, за $\prec_a$: утверждение "v находится выше u в списке предпочтений a" будем записывать как $v \prec_a u$.

Требуется устроить каждого соискателя в одну из компаний, то есть выбрать совершенное (покрывающее все вершины) паросочетание в этом графе. При этом хочется, чтобы распределение соискателей по компаниям было в некотором смысле устойчивым, то есть не было бы таких компании a и соискателя b, что a хотела бы нанять b вместо текущего работника, а b хотел бы сменить своего работодателя на a. Формально, назовём совершенное паросочетание M *стабильным*, если в нём не найдётся такой пары рёбер (a_1, b_1), (a_2, b_2), что $b_2 \prec_{a_1} b_1$, а $a_1 \prec_{b_2} a_2$ (будем говорить в таком случае, что пара a_1, b_2 *нарушает стабильность*). Задача состоит в том, чтобы найти в данном графе стабильное паросочетание.

R Можно рассматривать более общие версии задачи: компаний и соискателей может быть не поровну, списки предпочтений могут быть не полными (компания может рассматривать в качестве кандидатов на приём на работу не всех соискателей, а соискатель может хотеть работать не в каждой компании), компания может иметь не одну, а несколько вакансий. Разумеется, то, что требуется найти в перечисленных версиях задачи, не всегда является совершенным паросочетанием, а в последнем случае не является даже паросочетанием. Тем не менее, описанное ниже решение задачи легко модифицируется на все перечисленные обобщения.

9.2 Алгоритм Гейла-Шепли

Из определения, вообще говоря, не очевидно даже, всегда ли стабильное паросочетание существует. Оказывается, что это так, и в произвольном графе его можно найти несложным алгоритмом.

Теорема 9.2.1 — Gale, Shapley (1962, [14]). Стабильное паросочетание существует.

Доказательство. Покажем, что следующий алгоритм находит стабильное паросочетание:

1. Каждая компания делает оффер первому в её списке предпочтений соискателю.
2. Этот шаг повторяется, пока существует соискатель b, имеющий офферы сразу от двух компаний a_1, a_2. Пусть, не нарушая общности, $a_1 \prec_b a_2$, тогда b отклоняет оффер a_2, а a_2 делает оффер следующему соискателю в своём списке предпочтений.

Заметим, что на втором шаге алгоритма у a_2 всегда найдётся следующий в списке предпочтений соискатель (то есть список не может закончиться). Действительно, пусть b — последний в списке предпочтений a_2, тогда a_2 уже делала оффер всем соискателям. При этом с момента, как соискатель впервые получает какой-то оффер, и до конца работы алгоритма этот соискатель всегда будет иметь на руках хотя бы один оффер. Значит, в момент, когда b отклоняет оффер a_2, каждый соискатель имеет хотя бы один оффер. Но b имеет хотя бы два оффера, а это невозможно по принципу Дирихле.

Итак, поскольку списки предпочтений конечны, алгоритм завершится за конечное время. В конце работы алгоритма каждая компания делает оффер какому-то соискателю, при этом никакой соискатель не имеет два оффера. Значит, алгоритм нашёл какое-то совершенное паросочетание; обозначим его за M. Осталось показать, что M — стабильное.

Предположим, что пара a, b нарушает стабильность M; пусть $(a, p), (q, b) \in M$. Поскольку $b \prec_a p$, компания a делала оффер соискателю b в какой-то момент в ходе выполнения алгоритма. Но это противоречит тому, что $a \prec_b q$: минимум среди компаний, предлагающих соискателю оффер, номеров в списке предпочтений этого соискателя в ходе работы алгоритма может только уменьшаться. ■

Аккуратная реализация описанного алгоритма имеет время работы $O(n^2)$.

9.3 Оптимальность решения

В алгоритме Гейла-Шепли есть некоторая неопределённость: если в какой-то момент сразу несколько соискателей имеют по несколько офферов, то они могут отклонять офферы в любом порядке; анализ алгоритма от этого не изменится. Возникает вопрос: зависит ли получающееся в конце стабильное паросочетание от порядка отклонения офферов?

Заметим, что, вообще говоря, стабильное паросочетание в графе совсем не обязательно единственно. Пусть, например, есть две компании (Гугл и Яндекс) и два соискателя (Вася и Петя). При этом Гугл предпочитает Васю, а Яндекс — Петю; Вася предпочёл бы работать в Яндексе, а Петя — в Гугле. В этом примере оба совершенных паросочетания стабильны.

Тем не менее, оказывается, что любая реализация алгоритма будет возвращать одно и то же паросочетание; более того, это паросочетание в каком-то смысле является оптимальным для компаний. Будем называть соискателя b *потенциальным* для компании a, а a — *потенциальной* для b, если $(a, b) \in M$ для какого-либо стабильного паросочетания M. Будем обозначать за $best(a)$ лучшего из потенциальных соискателей для a, то есть находящегося выше всех остальных потенциальных соискателей в списке предпочтений a.

Предложение 9.3.1 Пусть M — стабильное паросочетание, найденное какой-либо реализацией алгоритма Гейла-Шепли; A — множество компаний. Тогда

$$M = \{(a, best(a)) \mid a \in A\}.$$

Доказательство. Пусть это не так, тогда (поскольку компании делают офферы в порядке списка предпочтений) найдётся момент, в который некоторый соискатель b отклоняет оффер такой компании a, что $b = best(a)$; рассмотрим первый такой момент. Существует такое стабильное паросочетание N, что $(a, b) \in N$. Поскольку b отклоняет оффер a, он имеет на руках оффер от некоторой компании c такой, что $c \prec_b a$. Пусть $(c, d) \in N$. c — потенциальная для d, значит, d ещё не отклонял оффер c (так как мы рассмотрели первый подобный момент). Но тогда $b \prec_c d$, поскольку в данный момент времени b имеет на руках оффер от c. Тогда пара b, c нарушает стабильность N, и мы получаем противоречие. ■

Покажем также, что для соискателей алгоритм, наоборот, находит в каком-то смысле худшее возможное стабильное паросочетание.

Предложение 9.3.2 Пусть M — стабильное паросочетание, найденное какой-либо реализацией алгоритма Гейла-Шепли. Тогда для любых $(a, b) \in M$ компания a — последняя в списке предпочтений среди потенциальных для b.

Доказательство. Пусть это не так для некоторых $(a, b) \in M$, то есть существует стабильное паросочетание N такое, что $(c, b) \in N$, $a \prec_b c$. Из предыдущего предложения мы знаем, что $b = best(a)$. Пусть $(a, d) \in N$, тогда d — потенциальный для a, значит, $best(a) = b \prec_a d$. Пара a, b нарушает стабильность N, противоречие. ■

R Вариации алгоритма Гейла-Шепли широко применяются на практике: помимо прочего, подобные алгоритмы используются в США для распределения выпускников медицинских вузов по больницам; учеников по школам; донорских органов по нуждающимся пациентам. В 2012 году Ллойд Шепли и Элвин Рот (последний как раз занимался адаптацией алгоритма Гейла-Шепли к вышеперечисленным ситуациям) получили Нобелевскую премию по экономике с формулировкой “for the theory of stable allocations and the practice of market design”. Кроме того, алгоритмы, подобные алгоритму Гейла-Шепли, применяются для оптимизации распределения интернет-пользователей по серверам в Content Delivery Networks (подробности можно почитать здесь).

IV Потоки

10. Максимальный поток

10.1 Определения и базовые свойства

Пусть дан ориентированный граф $G = (V, E)$ с двумя зафиксированными вершинами $s \in V$ — *истоком (source)* и $t \in V$, $t \neq s$ — *стоком (sink)*, а каждому ребру графа $e \in E$ соответствует вещественное число $c_e \geqslant 0$ — *пропускная способность (capacity)* ребра. Будем называть *сетью (network)* и обозначать той же буквой, что и граф, весь набор $G = (V, E, s, t, \{c_e\}_{e \in E})$.

Множество входящих в вершину $v \in V$ рёбер будем обозначать как $E_{in}(v)$, множество исходящих из вершины v рёбер — как $E_{out}(v)$.

Определение 10.1.1 *Поток (flow)* в сети G — это такой набор вещественных чисел, соотвествующих рёбрам графа $\{f_e\}_{e \in E}$, что выполняются следующие условия:

1. $0 \leqslant f_e \leqslant c_e$ для любого $e \in E$;
2. $\sum_{e \in E_{in}(v)} f_e = \sum_{e \in E_{out}(v)} f_e$ для любой $v \in V$, $v \neq s, t$.

R Удобно представлять поток как некую жидкость, "текущую" по сети из истока в сток. Пропускная способность ребра — это максимальное количество жидкости, которое может протечь по этому ребру за единицу времени. Тогда первое условие в определении потока означает, что по каждому ребру течёт неотрицательное количество жидкости, не превосходящее пропускную способность ребра; второе условие означает, что в промежуточных вершинах жидкость не задерживается: сколько её втекает, столько и вытекает.

Величиной потока (value of the flow) будем называть

$$|f| = \sum_{e \in E_{out}(s)} f_e - \sum_{e \in E_{in}(s)} f_e.$$

Другими словами, $|f|$ — количество жидкости, возникающее в истоке и вытекающее из него за единицу времени. При этом жидкость не задерживается в промежуточных вершинах, и пропадает только в стоке, поэтому из жидкостной интерпретации интуитивно следует, что $|f|$ должно равняться и количеству жидкости, втекающему в сток и пропадающему в нём за единицу времени. Докажем это утверждение формально.

Лемма 10.1.1

$$|f| = \sum_{e \in E_{in}(t)} f_e - \sum_{e \in E_{out}(t)} f_e.$$

Доказательство. Запишем сумму величин потока по всем рёбрам сети двумя способами:

$$\sum_{v \in V} \sum_{e \in E_{in}(v)} f_e = \sum_{v \in V} \sum_{e \in E_{out}(v)} f_e.$$

По второму свойству потока из сумм слева и справа можно убрать все вершины, кроме истока и стока:

$$\sum_{e \in E_{in}(s)} f_e + \sum_{e \in E_{in}(t)} f_e = \sum_{e \in E_{out}(s)} f_e + \sum_{e \in E_{out}(t)} f_e.$$

Получившееся равенство эквивалентно утверждению леммы. ■

Первая задача, которой мы займёмся — поиск максимального (по величине) потока в сети. Как можно искать максимальный поток? Пусть у нас уже есть какой-то поток f в сети G (начать можно, например, с потока, равного нулю в каждом ребре). Будем называть ребро сети $e \in E$ *насыщенным (saturated)*, если $f_e = c_e$. Как можно пытаться увеличить поток f? Естественная идея: если в сети есть путь P из истока в сток, все рёбра которого ненасыщены, то поток f можно увеличить, увеличив f_e для каждого ребра e на этом пути на одну и ту же величину $\Delta = \min_{e \in P}(c_e - f_e)$. Поскольку $f_e + \Delta \leqslant c_e$ для всех рёбер на пути, и каждая промежуточная вершина имеет поровну входящих и исходящих рёбер на пути, свойства потока не нарушатся, а величина потока увеличится на Δ. Пусть пути из истока в сток по ненасыщенным рёбрам в сети нет; правда ли, что тогда текущий поток — максимальный? Нет, неправда, как видно из следующего примера.

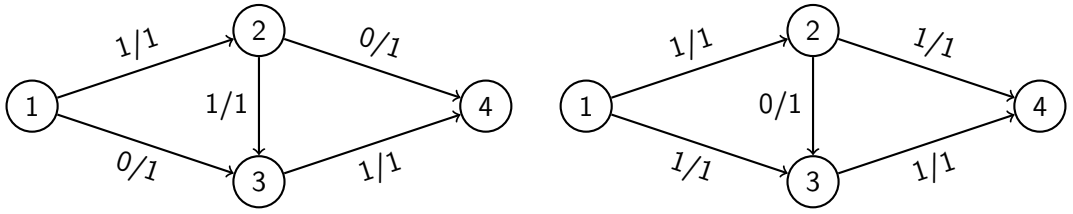

Рис. 10.1: Два потока в одной и той же сети ($s = 1$, $t = 4$). Отметка на ребре e имеет формат c_e/f_e.

На рисунке слева изображён поток размера 1, справа — поток в той же сети размера 2. В левом рисунке нет пути из s в t по ненасыщенным рёбрам, но поток не является максимальным (так как поток на правом рисунке больше по размеру). В чём проблема нашего текущего подхода? Оказывается, иногда для того, чтобы получить поток большего размера, нужно "отменить" часть потока, текущего по некоторым рёбрам (по ребру $2 \to 3$ в примере выше).

Более удобные определения

Для того, чтобы учитывать возможность "отмены" потока, оказывается удобным добавить для каждого ребра $e \in E$, $e : a \to b$ исходного графа фиктивное *обратное (backward)* ребро $e' : b \to a$ с нулевой пропускной способностью: $c_{e'} = 0$. Обозначим множество обратных рёбер за E^{bwd}; множество *прямых (forward)* рёбер, то есть рёбер исходного графа, переобозначим за E^{fwd}; в сети теперь будут и прямые, и обратные рёбра: $E = E^{bwd} \cup E^{fwd}$. Для удобства будем считать, что $(e')' = e$ для любого прямого ребра $e \in E^{fwd}$. На добавленных нами фиктивных рёбрах теперь надо как-то доопределить поток.

Определение 10.1.2 *Поток (flow)* в сети G — это такой набор вещественных чисел, соотвествующих рёбрам графа $\{f_e\}_{e \in E}$, что выполняются следующие условия:

1. $f_e \leqslant c_e$ для любого $e \in E$;
2. $f_e = -f_{e'}$ для любого $e \in E$;
3. $\sum_{e \in E_{in}(v)} f_e = \sum_{e \in E_{out}(v)} f_e = 0$ для любой $v \in V$, $v \neq s, t$.

Заметим, что поток вдоль любого фиктивного ребра неположителен: для любого $e' \in E^{bwd}$ верно $f_{e'} \leqslant c_{e'} = 0$. Отсюда сразу следует, что поток вдоль любого прямого

ребра неотрицателен: для любого $e \in E^{fwd}$ верно $f_e = -f_{e'} \geqslant 0$; значит, условие на прямые рёбра $0 \leqslant f_e \leqslant c_e$ из старого определения потока по-прежнему выполняется. По-прежнему выполняется и второе условие из старого определения: для любой $v \in V$, $v \neq s, t$

$$0 = \sum_{e \in E_{in}(v)} f_e = \sum_{e \in E^{fwd}_{in}(v)} f_e + \sum_{e \in E^{bwd}_{in}(v)} f_e = \sum_{e \in E^{fwd}_{in}(v)} f_e - \sum_{e \in E^{fwd}_{out}(v)} f_e.$$

Итак, новое определение совпадает со старым, если удалить фиктивные рёбра и оставить только прямые. Заметим также, что формулу для величины потока теперь можно записать более компактно:

$$|f| = \sum_{e \in E_{out}(s)} f_e = \sum_{e \in E_{in}(t)} f_e.$$

Остаточная сеть и дополняющие пути

Чем же полезны обратные рёбра и новое определение потока? Пусть, опять, f — какой-то уже имеющийся поток в сети G, и пусть при этом в сети имеется путь P из s в t по ненасыщенным рёбрам (некоторые из которых, возможно, обратные). Мы снова можем увеличить поток вдоль всех рёбер на пути на одну и ту же величину $\Delta = \min_{e \in P}(c_e - f_e)$ (и, чтобы не нарушить определение потока, уменьшить поток на ту же величину вдоль рёбер, обратных к рёбрам на пути); свойства потока не нарушатся, а величина потока увеличится на Δ. При этом увеличение потока вдоль фиктивного ребра как раз соответствует "отмене" части потока вдоль соответствующего ему прямого ребра.

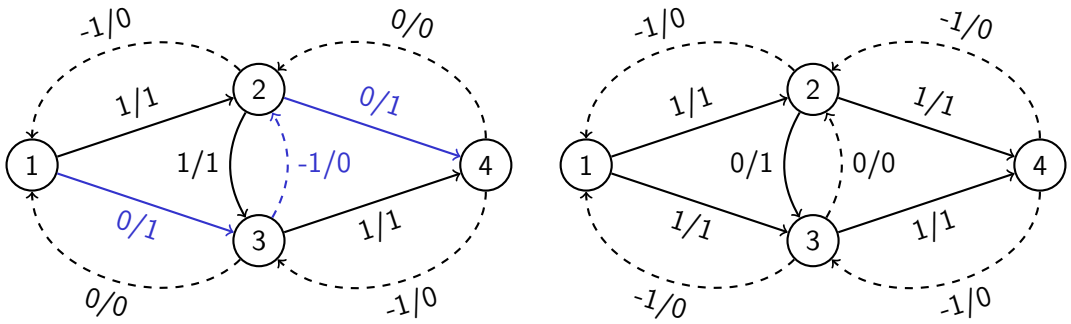

Рис. 10.2: Те же два потока, но уже в сети с обратными рёбрами (помечены пунктиром). Путь из s в t, состоящий из ненасыщенных рёбер, отмечен синим цветом.

Оказывается, если в сети с обратными рёбрами нет пути из истока в сток по ненасыщенным рёбрам, то текущий поток является максимальным. Наша ближайшая цель — доказать этот факт. Введём ещё несколько вспомогательных определений:

Пусть f — поток в сети G. Определим *остаточную сеть (residual net)* G_f потока f: $G_f = (V, E, s, t, \{c^f\}_{e \in E})$, где $c^f_e = c_e - f_e$ для любого ребра $e \in E$. Заметим, что, поскольку $f_e \leqslant c_e$, $c^f_e \geqslant 0$ для любого $e \in E$, то есть G_f является сетью (но фиктивные обратные рёбра в G_f уже могут иметь положительную пропускную способность). c^f_e называют *остаточной пропускной способностью (residual capacity)* ребра e; неформально говоря, c^f_e показывает, сколько ещё потока можно "пустить" по ребру e (или, если e — фиктивное, то сколько потока вдоль соответствующего прямого ребра можно "отменить"). Заметим, что ненасыщенные относительно f рёбра сети G — это ровно рёбра, имеющие положительную пропускную способность в G_f.

Под суммой и разностью в лемме ниже имеются в виду поточечные сумма и разность: $(f \pm h)_e = f_e \pm h_e$ для любого $e \in E$.

Лемма 10.1.2 — Лемма о суммах и разностях потоков.

1. Пусть f — поток в сети G, h — поток в G_f. Тогда $f+h$ — поток в G, $|f+h| = |f|+|h|$.
2. Пусть f, h — потоки в сети G, тогда $h - f$ — поток в G_f, $|h - f| = |h| - |f|$.
3. Пусть f, h — потоки в сети G, $f_e \leqslant h_e$ для любого прямого ребра $e \in E^{fwd}$. Тогда $h - f$ — поток в G, $|h - f| = |h| - |f|$.

Доказательство. Заметим, что в каждом случае второе и третье условия на потоки выполняются для суммы/разности потоков по линейности: поскольку соответствующие равенства верны для каждого из потоков по отдельности, они верны и для их суммы/разности. По тем же причинам верны выражения для величины суммы/разности потоков. Значит, достаточно показать, что выполняется первое условие на потоки.

1. $(f + h)_e = f_e + h_e \leqslant f_e + (c_e - f_e) \leqslant c_e$ для любого $e \in E$.
2. $(h - f)_e = h_e - f_e \leqslant c_e - f_e = c_e^f$ для любого $e \in E$.
3. Для любого $e \in E^{fwd}$ верно $(h - f)_e = h_e - f_e \leqslant h_e \leqslant c_e$; для обратного к нему e' верно $(h - f)_{e'} = h_{e'} - f_{e'} = -(h_e - f_e) \leqslant 0 = c_{e'}$.

■

Пусть $P \subset E$ — какой-то набор рёбер сети (например, путь или цикл). Определим

$$c(P) = \min_{e \in P} c_e, \quad f(P) = \min_{e \in P} f_e, \quad c^f(P) = \min_{e \in P} c_e^f.$$

Также будем использовать обозначения вида $x \cdot flow_e^P$, $e \in E$: если x — вещественное число, то

$$x \cdot flow_e^P = \begin{cases} x, & \text{если } e \in P, \\ -x, & \text{если } e' \in P, \\ 0, & \text{иначе.} \end{cases}$$

Будем называть путь P из s в t *дополняющим (augmenting)* в G_f, если $c^f(P) > 0$. В примерах выше мы говорили как раз о дополняющих путях. Нетрудно заметить, что если P — дополняющий путь, то $c^f(P) \cdot flow^P$ — поток в G_f. Тогда, по первому пункту леммы о сумме и разности потоков, $f + c^f(P) \cdot flow^P$ — поток в G,

$$|f + c^f(P) \cdot flow^P| = |f| + |c^f(P) \cdot flow^P| = |f| + c^f(P) > |f|;$$

другими словами, как мы и обсуждали выше, вдоль дополняющего пути поток всегда можно увеличить.

Разрезы и их свойства

(s,t)-*разрез* ((s,t)-*cut*) (S, T) — это разбиение множества вершин сети на два таких непересекающихся множества, что исток лежит в одном из них, а сток в другом: $S, T \subset V$, $S \cup T = V$, $S \cap T = \emptyset$, $s \in S$, $t \in T$.

Для краткости будем говорить просто "разрез" вместо "(s,t)-разрез", если из контекста понятно, о какой сети идёт речь.

Будем говорить, что ребро $e : a \to b$ *проходит через разрез* (S, T), если $a \in S$, $b \in T$. Для краткости при суммировании будем использовать сокращение $e : A \to B$, имея в виду суммирование по рёбрам, ведущим из множества A в множество B.

Пропускная способность разреза — это сумма пропускных способностей всех проходящих через него рёбер; аналогично определим величину потока, проходящего через разрез:

$$c(S,T) = \sum_{e:S\to T} c_e, \quad f(S,T) = \sum_{e:S\to T} f_e.$$

Мы будем иногда использовать те же обозначения для двух произвольных подмножеств вершин S, T, не обязательно образующих разрез.

Минимальный разрез — это разрез с минимальной возможной пропускной способностью.

Лемма 10.1.3 Пусть f — поток в сети G, (S,T) — разрез в той же сети. Тогда
1. $|f| = f(S,T)$;
2. $f(S,T) \leqslant c(S,T)$;
3. Если $f(S,T) = c(S,T)$, то f — максимальный поток, а (S,T) — минимальный разрез.

Доказательство.
1. $f(S,T) = f(S,V) - f(S,S) = f(S,V)$, так как $f(S,S) = 0$ по второму свойству потока. $f(S,V) = f(S\setminus\{s\},V) + |f| = |f|$, так как $f(S\setminus\{s\},V) = 0$ по третьему свойству потока.
2. Суммируем неравенство $f_e \leqslant c_e$ по всем рёбрам, проходящим через разрез.
3. Пусть (S',T') — какой-то другой разрез. $c(S,T) = f(S,T) = f(S',T') \leqslant c(S',T')$; значит, (S,T) — минимальный.
   Пусть f' — какой-то другой поток. $|f'| = f'(S,T) \leqslant c(S,T) = |f|$; значит, f — максимальный.

■

10.2 Алгоритм Форда-Фалкерсона

Теорема 10.2.1 — Теорема Форда-Фалкерсона (Ford, Fulkerson, 1956, [12]).
Пусть f — поток в сети G. Следующие три условия эквивалентны:
1. f — максимальный поток;
2. в G_f нет дополняющего пути;
3. $|f| = c(S,T)$ для некоторого разреза (S,T).

Доказательство.

$1 \Rightarrow 2$. Если бы в остаточной сети G_f нашёлся дополняющий путь, поток f не был бы максимальным.

$2 \Rightarrow 3$. Пусть S — множество вершин, достижимых из s по ненасыщенным рёбрам, $T = V\setminus S$. Поскольку в G_f нет дополняющих путей, $t \in T$; значит, (S,T) — разрез. $|f| = f(S,T)$; для любого проходящего через разрез ребра e выполняется $f_e = c_e$, так как иначе конец ребра лежал бы в S. Значит, $|f| = f(S,T) = c(S,T)$.

$3 \Rightarrow 1$. Уже доказано в лемме выше.

■

R Заметим, что теорема Форда-Фалкерсона ничего не говорит о существовании максимального потока; она лишь позволяет проверить, является ли данный поток максимальным.

Доказательство теоремы Форда-Фалкерсона даёт алгоритм нахождения минимального разреза по уже имеющемуся максимальному потоку за $O(V+E)$.

Алгоритм Форда-Фалкерсона

Сам же максимальный поток мы будем искать итеративно: будем поддерживать текущий поток f и остаточную сеть G_f; начнём с нулевого потока ($f_e = 0$ для $e \in E$). Будем искать дополняющий путь в G_f поиском в глубину; если путь находится, будем увеличивать поток вдоль пути; если пути не находится, то по теореме Форда-Фалкерсона текущий поток — максимальный.

Будем называть сеть *целочисленной*, если $c_e \in \mathbb{Z}$ для любого $e \in E$. Будем называть поток *целочисленным*, если $f_e \in \mathbb{Z}$ для любого $e \in E$.

Предложение 10.2.2 Пусть G — целочисленная сеть. Тогда вышеописанный алгоритм работает за $O(E \cdot |f|) = O(E \cdot \sum_{e \in E_{out}(s)} c_e)$, где $|f|$ — величина максимального потока в сети.

Доказательство. Покажем по индукции, что в начале каждой итерации алгоритма поток целочисленный. Каждая итерация имеет время работы $O(E)$, при этом если дополняющий путь P нашёлся, то величина потока увеличится на целое число, больше или равное единицы (поскольку для любого $e \in P$ верно $c_e^f > 0$, $c_e^f \in \mathbb{Z}$, то есть $c_e^f \geqslant 1$). ■

> **Следствие 10.2.3** Пусть G — целочисленная сеть. Тогда существует целочисленный максимальный поток.

R Если сеть не является целочисленной, то вышеописанный алгоритм может работать бесконечно долго. Более того, в такой сети величина текущего потока в алгоритме может не сходиться к величине максимального потока в сети.

Заметим также, что пока мы даже не знаем, всегда ли существует максимальный поток в нецелочисленной сети.

Реализация

Будем хранить сеть с помощью списков смежности. Остаточную сеть явно хранить не будем, поскольку остаточную пропускную способность легко вычислить, зная пропускную способность ребра и величину потока через это ребро.

```
class Edge:
  int to, f, c
  Edge *rev

vector<Edge> es

# добавить в сеть ребро из a в b с пропускной способностью c,
# а также обратное ребро
addEdge(a, b, c):
  e = new Edge(b, 0, c)
  erev = new Edge(a, 0, 0)
  e->rev = erev
  erev->rev = e
  es[a].push_back(e)
  es[b].push_back(erev)
```

Перестраивать поток удобно на выходе из рекурсии в поиске в глубину.

```
# minf - минимум остаточной пропускной способности на пути от s до v
# dfs возвращает пропускную способность найденного дополняющего пути,
# либо 0, если дополняющий путь не был найден
dfs(v, minf):
  if v == t:
    return minf
  used[v] = True
  for e in es[v]:
    if e.f < e.c and not used[e.to]:
      flow = dfs(e.to, min(minf, e.c - e.f))
      if flow > 0:
        e.f += flow
        e.rev->f -= flow
        return flow
  return 0

flow = 0
fill(used, False)
while True:
  curFlow = dfs(s, inf) # inf заведомо больше величины максимального потока
  if curFlow == 0:
    break
  flow += curFlow
  fill(used, False)
```

10.3 Декомпозиция потока

Предложение 10.3.1 Пусть f — поток в сети G, $|f| \geqslant 0$. Тогда f можно представить в виде суммы $O(E)$ потоков вида $x \cdot flow^P$, где $x > 0$, P — путь из s в t либо цикл.

Доказательство. Пусть $cnt(f) = |\{e \in E \mid f_e > 0\}|$; Докажем по индукции, что f можно представить в виде суммы не более $cnt(f)$ потоков вышеописанного вида. Если $cnt(f) = 0$, доказывать нечего. Если $cnt(f) > 0$, найдётся хотя бы одно ребро $e_1 \in E$ такое, что $f_{e_1} > 0$; если $|f| > 0$, то выберем e_1 так, что $e_1 \in E_{out}(s)$.

Теперь будем повторять следующие рассуждения: пусть уже найдены рёбра $e_1 : v_0 \to v_1$, $e_2 : v_1 \to v_2$, $\ldots$, $e_k : v_{k-1} \to v_k$; $f_{e_i} > 0$ для $1 \leqslant i \leqslant k$; $v_0, \ldots, v_{i-1}$ различны и не совпадают с t.

- Пусть $|f| > 0$, $v_k = t$, тогда $P = \{e_1, \ldots, e_k\}$ — путь из s в t с $f(P) > 0$. Тогда $f(P) \cdot flow^P$ — поток в G, $h = f - f(P) \cdot flow^P$ — поток в G по третьему пункту леммы о суммах и разностях потоков, причём $cnt(h) < cnt(f)$. Значит, по индукции, утверждение верно для h, то есть и для f.
- Пусть $v_k = v_i$ для некоторого $i < k$. Тогда $C = \{e_{i+1}, \ldots, e_k\}$ — цикл, $f(C) > 0$. Тогда $f(C) \cdot flow^C$ — поток в G, $h = f - f(C) \cdot flow^C$ — тоже поток в G, причём $cnt(h) < cnt(f)$. Значит, по индукции, утверждение верно для h, то есть и для f.
- Пусть ни одно из предыдущих двух предположений не выполнено. Тогда либо $v_k \neq s, t$, либо $|f| = 0$. В любом случае, $\sum_{e \in E_{in}^{fwd}(v_k)} f_e = \sum_{e \in E_{out}^{fwd}(v_k)} f_e$. Тогда, поскольку $e_k \in E_{in}^{fwd}(v_k)$, $f_{e_k} > 0$, найдётся $e_{k+1} \in E_{out}^{fwd}(v_k)$, $f_{e_{k+1}} > 0$. Повторим рассуждения для $e_1, \ldots, e_{k+1}$.

■

Описанное выше представление называют *декомпозицией* потока. Заметим, что все пути и циклы в декомпозиции состоят из прямых рёбер, поскольку по фиктивному ребру не может течь поток положительной величины. Доказательство предложения фактически

описывает алгоритм нахождения декомпозиции: будем искать нужный путь или цикл поиском в глубину. Наивная реализация имеет время работы $O(E^2)$, поскольку при поиске очередного пути или цикла в худшем случае придётся перебрать все исходящие рёбра у всех вершин. Время работы можно улучшить: заметим, что в списке рёбер вершины нас всегда интересует первое ребро, поток по которому положителен. Будем поддерживать указатель на такое ребро, и сдвигать его, когда поток вдоль ребра становится равен нулю. Получаем алгоритм с временем работы $O(VE + E) = O(VE)$.

10.4 Масштабирование потока

Пусть G — целочисленная сеть; обозначим $U = \sum_{e \in E_{out}(s)} c_e$. Иногда время работы алгоритма Форда-Фалкерсона можно улучшить с помощью техники *масштабирования потока (capacity scaling)*. Будем запускать алгоритм несколько раз с дополнительным параметром $k = \lfloor \log_2 U \rfloor, \ldots, 0$. При этом поиском в глубину будем искать такой дополняющий путь P, что $c^f(P) \geqslant 2^k$.

```
dfs(v, minf, k):
  if v == t:
    return minf
  used[v] = True
  for e in es[v]:
    if e.c - e.f >= (1 << k) and not used[e.to]:
      flow = dfs(e.to, min(minf, e.c - e.f)
      if flow > 0:
        e.f += flow
        e.rev->f -= flow
        return flow
  return 0

flow = 0
for k = log(U)..0:
  fill(used, False)
  while True:
    curFlow = dfs(s, inf, k)
    if curFlow == 0:
      break
    flow += curFlow
    fill(used, False)
```

Полученный алгоритм корректно находит максимальный поток, поскольку последняя итерация ($k = 0$) завершится, когда в сети совсем не будет дополняющих путей.

Предложение 10.4.1 Алгоритм Форда-Фалкерсона с масштабированием потока имеет время работы $O(E^2 \log U)$.

Доказательство. Достаточно доказать, что на каждой фазе алгоритм найдёт не более $2|E|$ дополняющих путей.

Пусть h — максимальный поток в сети. $|h| \leqslant U$, значит, на фазе $k = \lfloor \log_2 U \rfloor$ алгоритм найдёт не более одного дополняющего пути.

За f^i обозначим поток, полученный алгоритмом в конце фазы $k = i$. Поскольку дополняющих путей относительно f^i с пропускной способностью 2^i в сети нет, в G_{f^i} найдётся такой разрез (S, T), что для любого ребра e, проходящего через разрез, верно $c_e - f^i_e < 2^i$ (за S можно взять множество вершин, достижимых из s по рёбрам с $c^{f^i} \geqslant 2^i$). По второму пункту леммы о сумме и разности потоков, $h - f^i$ — поток в G_{f^i}, при этом

$$|h| - |f^i| \leqslant c(S, T) - f^i(S, T) < 2^i \cdot |E|,$$

то есть $|h| \leqslant |f^i| + 2^i \cdot |E|$. Тогда на фазе $k = i - 1$ алгоритм найдёт дополняющий путь не более $2|E|$ раз (поскольку каждый из этих путей увеличит поток хотя бы на 2^{i-1}). ■

10.5 Алгоритм Эдмондса-Карпа

Алгоритм Эдмондса-Карпа (Edmonds, Karp, 1972, [10]) отличается от алгоритма Форда-Фалкерсона тем, что вместо обхода в глубину он использует обход в ширину; другими словами, на каждом шаге он выбирает кратчайший **по числу рёбер** дополняющий путь. Оказывается, что такой алгоритм находит максимальный поток уже в произвольной (не обязательно целочисленной) сети за время, которое можно оценить функцией от числа вершин и рёбер сети, не используя величину максимального потока и пропускные способности рёбер.

Обозначим за d_v^i длину кратчайшего пути по ненасыщенным рёбрам от s до v на i-м шаге алгоритма (то есть расстояние, найденное i-м запуском поиска в ширину); считаем, что $d_v^i = \infty$, если вершина v не достижима по ненасыщенным рёбрам из s. Другими словами, d_v^i — номер *слоя*, в котором оказалась вершина v на i-шаге. Кратчайший дополняющий путь, найденный поиском в ширину на i-м шаге алгоритма, обозначим за P_i.

Заметим, что при увеличении потока вдоль P_i могут перестать быть насыщенными лишь рёбра, обратные к рёбрам P_i. Значит, ненасыщенных рёбер, ведущих больше чем один слой вперёд (то есть таких $e : a \to b$, что $d_b^i - d_a^i \geqslant 2$), при увеличении потока появиться не может. Отсюда сразу же следует, что расстояние от истока до любой вершины в ходе алгоритма может только увеличиваться: $d_v^j \geqslant d_v^i$ для любой v и любых $j > i$.

Теорема 10.5.1 Время работы алгоритма Эдмондса-Карпа — $O(E^2V)$.

Доказательство. При каждом увеличении потока вдоль дополняющего пути хотя бы одно ребро этого пути *насыщается* (то есть ненасыщенное ребро становится насыщенным). Покажем, что каждое ребро могло насытиться за время работы алгоритма лишь $O(V)$ раз; тогда число запусков поиска в ширину не превосходит $O(VE)$, откуда следует требуемая оценка.

Пусть ребро $e : a \to b$ насытилось на i-м шаге алгоритма ($e \in P_i$), после чего в следующий раз оно насытилось на j-м шаге ($e \in P_j$). Ребро e перестало быть насыщенным в какой-то момент между этими двумя, то есть на k-м шаге для некоторого $i < k < j$. Тогда обратное ребро $e' : b \to a$ лежало на пути, вдоль которого увеличивался поток на этом шаге: $e' \in P_k$. Каждый из путей P_i, P_k, P_j был кратчайшим на соответствующем шаге, значит,

$$d_b^i = d_a^i + 1, \quad d_a^k = d_b^k + 1, \quad d_b^j = d_a^j + 1.$$

Поскольку расстояние от истока до любой вершины в ходе алгоритма лишь возрастает,

$$d_b^j = d_a^j + 1 \geqslant d_a^k + 1 = d_b^k + 2 \geqslant d_b^i + 2.$$

Таким образом, за время между любыми двумя соседними насыщениями ребра расстояние от истока до конца этого ребра возрастает хотя бы на два. Расстояние до любой вершины не превосходит $|V| - 1$ (или равно бесконечности), значит, никакое ребро не могло насытиться больше, чем $|V|/2 = O(V)$ раз. ■

Следствие 10.5.2 Пусть G — сеть с произвольными неотрицательными вещественными пропускными способностями. Тогда в G существует максимальный поток.

10.6 Алгоритм Диница

В ходе работы алгоритма Эдмондса-Карпа расстояние от истока до любой другой вершины (в частности, до стока) может только увеличиваться. Значит, длины дополняющих путей, вдоль которых увеличивается поток, в ходе работы алгоритма тоже могут лишь возрастать. Изучим множество найденных алгоритмом дополняющих путей какой-нибудь фиксированной длины; все эти пути были найдены на каком-то подотрезке множества шагов алгоритма; обозначим их за $P_i, \dots, P_j$. Рассмотрим разбиение множества вершин на слои d^i, полученное на i-м шаге алгоритма. Любое ребро любого из путей $P_i, \dots, P_j$ направлено из некоторого слоя в следующий: если $e \in P_k$, $i \leqslant k \leqslant j$, $e : a \to b$, то $d^i_b = d^i_a + 1$. Действительно, это так для пути P_i; это так и для любого последующего пути P_k, $i < k \leqslant j$, поскольку при увеличении потока вдоль P_{k-1} любое ребро, которое перестаёт быть насыщенным, ведёт из некоторого слоя в предыдущий.

Тогда при поиске этих путей можно не рассматривать никакие рёбра, кроме ведущих из слоя в следующий за ним; будем называть такие рёбра *ведущими вперёд*. Заметим, что при поиске путей длины d ведущие вперёд рёбра могут лишь насыщаться, но не могут переставать быть насыщенными. В частности, отсюда следует, что если в какой-то момент из какой-то вершины нет пути по ведущим вперёд ненасыщенным рёбрам до стока, то такого пути не появится, пока расстояние от истока до стока не увеличится.

Значит, можно применить оптимизацию, которую мы уже использовали при поиске декомпозиции потока: будем поддерживать для каждой вершины указатель на первое в её списке ненасыщенное ведущее вперёд ребро; будем сдвигать этот указатель, когда ребро насыщается, или когда оказалось, что из конца ребра нельзя дойти по ненасыщенным ведущим вперёд рёбрам до стока. Получился алгоритм Диница (Dinitz, 1970, [9]).

Оценка времени работы

Алгоритм состоит из нескольких фаз; на каждой фазе алгоритм ищет дополняющие пути фиксированной длины. В начале фазы мы один раз запускаем поиск в ширину, чтобы найти разбиение на слои (и проверить, достижим ли вообще сток из истока). После этого ищем дополняющие пути поиском в глубину, который ходит только по ненасыщенным ведущим вперёд рёбрам и пользуется вышеописанными указателями. При этом каждый найденный путь насыщает хотя бы одно из ведущих вперёд рёбер, значит, число запусков поиска в глубину в течение одной фазы есть $O(E)$. При каждом запуске поиска в глубину каждая из вершин будет посещена не более одного раза; при этом каждое рассмотрение ребра будет вести либо к рекурсивному запуску поиска в глубину, либо к сдвигу указателя; суммарное количество сдвигов указателей за одну фазу есть $O(E)$. Наконец, длина дополняющего пути не превосходит $|V| - 1$, значит, число фаз есть $O(V)$. Таким образом, суммарное время работы алгоритма есть $O(V \cdot ((V + E) + E \cdot V + E)) = O(V^2 E)$.

Реализация

```
bfs():
  fill(d, inf)
  d[s] = 0
  q <-- s
  while not q.empty():
    q --> v
    for e in es[v]:
      if e.f < e.c and d[e.to] == inf:
        d[e.to] = d[v] + 1
        q <-- e.to
  return (d[t] != inf)
```

```
dfs(v, minf):
  if v == t:
    return minf
  for (; p[v] < es[v].size(); p[v] += 1):
    e = es[v][p[v]]
    if e.f < e.c and d[e.to] == d[v] + 1:
      flow = dfs(e.to, min(minf, e.c - e.f))
      if flow > 0:
        e.f += flow
        e.rev->f -= flow
        return flow
  return 0

flow = 0
while bfs():
  fill(p, 0)
  while True:
    curFlow = dfs(s, inf)
    if curFlow == 0:
      break
    flow += curFlow
```

10.7 Алгоритм Диница с масштабированием потока

Пусть сеть, в которой мы хотим искать максимальный поток, целочисленна. Технику масштабирования потока можно применить и к алгоритму Диница: снова будем запускать алгоритм несколько раз с дополнительным параметром $k = \lfloor \log_2 U \rfloor, \dots, 0$, где $U = \sum_{e \in E_{out}(s)} c_e$. При этом в поиске в ширину и в поиске в глубину будем считать ненасыщенными те рёбра, что имеют остаточную пропускную способность хотя бы 2^k.

Предложение 10.7.1 Алгоритм Диница с масштабированием потока имеет время работы $O(VE \log U)$.

Доказательство. Вспомним, что за одну фазу масштабирования $k = i$ поток вдоль дополняющего пути с остаточной пропускной способностью хотя бы 2^i можно увеличить не более $2|E|$ раз. Значит, суммарное число запусков поиска в глубину можно оценить как $O(E)$ не в одной фазе алгоритма Диница, а в каждом запуске алгоритма Диница целиком. Получаем оценку $O(\log U \cdot (V \cdot ((V + E) + E) + E \cdot V) = O(VE \log U)$. ■

R Время работы одной фазы алгоритма Диница можно ускорить при помощи link/cut trees; получится алгоритм с временем работы $O(VE \log V)$.

Не все алгоритмы поиска максимального потока основаны на идее увеличения потока вдоль дополняющего пути. Различные реализации алгоритма проталкивания предпотока работают за $O(V^3)$, $O(V^2\sqrt{E})$, $O(VE \log(V^2/E))$.

Недавно максимальный поток научились искать за $O(VE)$ (Orlin, 2013, [24]).

10.8 Паросочетания и потоки

Научимся сводить задачу поиска максимального паросочетания в двудольном графе к задаче поиска максимального потока. Пусть $G = (V, E)$ — двудольный граф с долями A, B, в котором мы хотим найти максимальное паросочетание. Построим по этому графу следующую сеть G': к множеству вершин графа V добавим две новых вершины: исток s и сток t; все рёбра графа направим из доли A в долю B и назначим им единичную пропускную способность. Кроме того, проведём рёбра с единичной пропускной способностью из s во все вершины доли A и из всех вершин доли B в t.

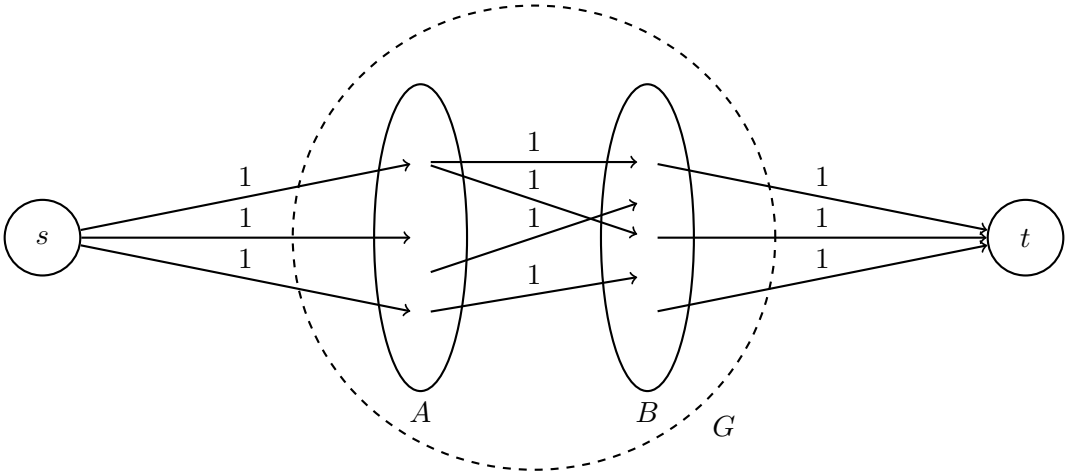

Рис. 10.3: Сеть G' для поиска максимального паросочетания в двудольном графе G

Пусть f — поток в сети G'. По теореме о декомпозиции потока f можно разбить на $|f|$ потоков вдоль путей из s в t, попарно не пересекающихся по промежуточным вершинам; каждому из этих путей соответствует своё ребро графа G, причём эти рёбра не имеют общих вершин, то есть образуют паросочетание. Наоборот, любому паросочетанию M в G несложно сопоставить поток величины $|M|$ в описанной сети: для каждого из рёбер паросочетания пустим единицу потока вдоль пути из s в t, проходящего через это ребро.

Таким образом, мы установили биекцию между потоками в сети G' и паросочетаниями в исходном графе G, причём размер потока равен размеру соответствующего ему паросочетания. Тогда для того, чтобы найти максимальное паросочетание в G, достаточно найти максимальный поток в G'; алгоритм Форда-Фалкерсона сделает это за $O(E|f|) = O(E|M|) = O(VE)$, где f — максимальный поток в G', M — максимальное паросочетание в G.

R Заметим, что изученный нами ранее алгоритм Куна фактически осуществляет те же действия, что и алгоритм Форда-Фалкерсона, просто не строит сеть явно (даже понятие дополняющего пути в обоих случаях имеет один и тот же смысл).

Минимальное вершинное покрытие и потоки

Вышеописанным методом несложно найти и минимальное вершинное покрытие в двудольном графе. Для удобства поменяем сеть, изменив пропускную способность рёбер G на бесконечно большую (это никак не помешает нам при поиске максимального потока, поскольку по этим рёбрам в любом случае не удастся пустить больше единицы потока). Вспомним, что по теореме Кёнига-Эгервари множество $B^+ \cup A^-$ является минимальным вершинным покрытием. Пусть (S, T) — минимальный разрез в сети G', соответствующий максимальному потоку f. Заметим, что $S = \{s\} \cup A^+ \cup B^+$, поскольку S — множество вершин, достижимых из s по ненасыщенным рёбрам (здесь важно, что рёбрам графа G мы назначили бесконечную пропускную способность, поэтому они не могут проходить через разрез (S, T)). Тогда минимальное вершинное покрытие можно найти как $(S \cap B) \cup (T \cap A)$.

Заметим, что здесь тоже имеет место биекция между вершинными покрытиями в G и разрезами **с конечной пропускной способностью** в G'. Действительно, тот факт, что вершинное покрытие покрывает все рёбра, соответствует тому, что через разрез не проходит рёбер с бесконечной пропускной способностью; число вершин в покрытии равно пропускной способности соответствующего разреза, поскольку каждой вершине из

$(S \cap B) \cup (T \cap A)$ соответствует ровно одно ребро пропускной способности 1, проходящее через разрез.

Взвешенные версии задач

С помощью потоков можно решать следующие обобщения задач о паросочетании и вершинном покрытии в двудольных графах:

- **Задача о мультисочетании.** Пусть мы хотим выбрать максимальное по размеру *мультисочетание* — подмножество рёбер $F \subset E$ так, чтобы любая вершина $v \in A$ была концом не более a_v рёбер из F, а любая $u \in B$ — концом не более b_u рёбер (неотрицательные числа a_v, b_u — параметры задачи; $a_v = b_u = 1$ для любых u, v соответствует обычной задаче о максимальном паросочетании).
  Рассмотрим всё ту же сеть, но заменим пропускные способности рёбер из s в $v \in A$ на a_v, а рёбер из $u \in B$ в t на b_u. Заметим, что теперь имеет место биекция между потоками в полученной сети и мультисочетаниями в исходном графе, причём эта биекция снова сохраняет размер(величину) объекта.
- **Минимальное по весу вершинное покрытие.** Пусть теперь неотрицательные числа a_v, $v \in A$; b_u, $u \in B$ — веса вершин, и мы хотим найти вершинное покрытие, имеющее минимально возможный суммарный вес.
  Модифицируем сеть так же, как и в предыдущей задаче, а также назначим рёбрам из A в B бесконечную пропускную способность (чтобы они не могли проходить через минимальный разрез). Имеет место биекция между вершинными покрытиями в G и разрезами с конечной пропускной способностью в полученной сети, причём эта биекция сохраняет сумму весов (пропускных способностей).

10.9 Теоремы Карзанова

Заметим, что в случае сети для поиска максимального паросочетания имеющаяся у нас оценка на алгоритм Диница $O(V^2E)$ ($O(VE \log V)$ при использовании техники масштабирования) оказывается хуже оценки времени работы алгоритма Форда-Фалкерсона $O(VE)$. Разумеется, алгоритм Диница на самом деле решает эту задачу не медленнее алгоритма Форда-Фалкерсона; дело здесь в том, что общая оценка $O(V^2E)$ далеко не всегда оказывается точной. Пользуясь свойствами данной в конкретной задаче сети, часто удаётся написать более точную оценку.

Вариации следующих теорем доказывались разными авторами в разное время; в России их принято называть теоремами Карзанова, в работе которого (1973, [28]) были доказаны менее общие утверждения, однако уже присутствовали все основные идеи доказательств.

Пусть $G = (V, E, s, t, \{c_e\})$ — целочисленная сеть. Обозначим

$$c_{in}(v) = \sum_{e \in E_{in}(v)} c_e, \quad c_{out}(v) = \sum_{e \in E_{out}(v)} c_e, \quad c(v) = \min(c_{in}(v), c_{out}(v)), \quad C = \sum_{v \neq s,t} c(v).$$

Лемма 10.9.1 Пусть f — поток в G. Для любой $v \neq s, t$ выполняется $c(v) = c^f(v)$.

Доказательство.

$$c^f_{in}(v) = \sum_{e \in E_{in}(v)} c^f_e = \sum_{e \in E_{in}(v)} c_e - \sum_{e \in E_{in}(v)} f_e = \sum_{e \in E_{in}(v)} c_e = c_{in}(v)$$

по третьему свойству потока. Равенство $c^f_{out}(v) = c_{out}(v)$ получается аналогично. ■

Теорема 10.9.2 Число фаз алгоритма Диница на целочисленной сети не превосходит $2\sqrt{C}$.

Доказательство. Пусть f — поток, полученный алгоритмом после первых $\sqrt{C}$ фаз, а h — какой-нибудь максимальный поток. $h - f$ — поток в G_f, его декомпозиция — набор из $k = |h| - |f|$ единичных потоков вдоль (возможно, повторяющихся) путей из s в t, а также нескольких потоков вдоль циклов. Будем считать, что потоков вдоль циклов в декомпозиции нет, так как их можно вычесть из h и получить поток той же величины. Заметим, что каждый путь в декомпозиции содержит хотя бы $\sqrt{C}$ промежуточных вершин: действительно, в конце каждой фазы алгоритма расстояние от s до t возрастает хотя бы на единицу.

Пусть через вершину $v \neq s, t$ проходит α_v из этих путей. Поскольку $h - f$ — поток в G_f, выполняется неравенство $\alpha_v \leqslant c^f(v) = c(v)$. Тогда

$$k\sqrt{C} \leqslant \sum_{v \neq s,t} \alpha_v \leqslant \sum_{v \neq s,t} c(v) = C,$$

откуда $k \leqslant \sqrt{C}$. Итак, $|h| \leqslant |f| + \sqrt{C}$, а на каждой фазе алгоритма находится хотя бы один дополняющий путь. Значит, алгоритм завершит работу через ещё не более чем $\sqrt{C}$ фаз. ■

Будем называть сеть *единичной*, если пропускные способности всех рёбер равны единице.

Лемма 10.9.3 Одна фаза алгоритма Диница на единичной сети имеет время работы $O(E)$.

Доказательство. Поиск в глубину пройдёт по каждому ненасыщенному ребру ровно один раз: если дополняющий путь рекурсивно найти не удастся, ребро будет пропущено; если же путь будет найден, ребро насытится. ■

Следствие 10.9.4 Алгоритм Диница на сети для поиска максимального паросочетания имеет время работы $O(E\sqrt{V})$.

Доказательство. Сеть для поиска максимального паросочетания — единичная; $c(v) = 1$ для любой $v \neq s, t$, поэтому $C \leqslant |V|$. Осталось применить только что доказанные теорему и лемму. ■

R Максимальное паросочетание в двудольном графе можно искать за $O(E\sqrt{V})$ алгоритмом Хопкрофта-Карпа. Подобно алгоритму Куна, он фактически моделирует работу алгоритма Диница на исходном графе, не строя сеть явно.

Следствие 10.9.5 Время работы алгоритма Диница на единичной сети — $O(E\sqrt{E})$.

Доказательство. Аналогично, только $C \leqslant |E|$. ■

Введём ещё одно обозначение: пусть $U = \max_{e \in E} c_e$ — максимальная пропускная способность ребра в сети.

Теорема 10.9.6 Пусть G — целочисленная сеть, в которой нет двух параллельных прямых рёбер (то есть таких $e_1, e_2 \in E^{fwd}$, что $e_1 : a \to b$, $e_2 : a \to b$). Число фаз алгоритма Диница на сети G не превосходит $2U^{1/3}V^{2/3}$.

Доказательство. Пусть f — поток, полученный алгоритмом после первых $k-1$ фаз, а h — какой-нибудь максимальный поток. Пусть $A_0, A_1, \ldots, A_m$; $t \in A_m$ — полученные в начале k-й фазы слои; $m \geqslant k$, поскольку $d_t^k \geqslant k$. Обозначим $a_i = |A_i|$.

Для любого $0 \leqslant i < m$ рассмотрим разрез $(S_i, T_i) = (A_0 \cup \cdots \cup A_i, A_{i+1} \cup \cdots \cup A_m)$; все ненасыщенные рёбра, проходящие через разрез, ведут из A_i в A_{i+1}. Поскольку в сети G нет прямых параллельных рёбер, в сети G_f имеется не более двух рёбер в каждом направлении между любой фиксированной парой вершин. Значит, выполняется неравенство $c^f(S_i, T_i) \leqslant 2Ua_ia_{i+1}$. Поскольку $h - f$ — поток в G_f, каждый такой разрез даёт нам оценку на величину максимального потока:

$$|h| \leqslant |f| + 2U \min_{0 \leqslant i < m} a_i a_{i+1}.$$

Будем называть слой A_i большим, если $a_i > 2|V|/k$, и маленьким иначе. Заметим, что больших слоёв меньше, чем $k/2 \leqslant m/2$. Значит, обязательно найдутся два соседних маленьких слоя A_i, A_{i+1}; получаем оценку

$$|h| \leqslant 2U \cdot \frac{4|V|^2}{k^2} + |f| = \frac{8U|V|^2}{k^2} + |f|.$$

Значит, алгоритму осталось выполнить не более $8U|V|^2/k^2$ фаз. Таким образом, общее число фаз не превышает $k - 1 + 8U|V|^2/k^2$. При $k = (U|V|^2)^{1/3}$ получаем оценку $9(U|V|^2)^{1/3} = O(U^{1/3}V^{2/3})$ на число фаз алгоритма. ■

Следствие 10.9.7 На единичной сети без параллельных прямых рёбер алгоритм Диница работает за $O(E \cdot \min(\sqrt{E}, V^{2/3}))$.

11. Макс. поток минимальной стоимости

11.1 Определения и свойства

Пусть теперь каждое ребро e сети имеет не только пропускную способность c_e, но и *стоимость (cost)* w_e — вещественное число, о котором можно неформально думать как о "стоимости пустить единицу потока по этому ребру". На обратных рёбрах зададим стоимость по правилу $w_{e'} = -w_e$; неформально это соответствует тому, что при отмене потока по прямому ребру деньги нам возвращаются. У потока теперь можно определить стоимость:

$$w(f) = \sum_{e \in E^{fwd}} w_e \cdot f_e.$$

Будем называть поток f *потоком минимальной стоимости*, если f имеет минимальную стоимость среди всех потоков величины $|f|$. Задача поиска максимального потока минимальной стоимости состоит в том, чтобы найти максимальный поток, являющийся потоком минимальной стоимости.

Для набора рёбер сети $P \subset E$ (например, пути или цикла), определим суммарную стоимость

$$w(P) = \sum_{e \in P} w_e.$$

Лемма 11.1.1 — Лемма о потоке минимальной стоимости. Поток f в сети G является потоком минимальной стоимости тогда и только тогда, когда в остаточной сети G_f нет циклов отрицательной стоимости, состоящих из ненасыщенных рёбер, то есть нет такого цикла C, что $c^f(C) > 0$, $w(C) < 0$.

Доказательство. Пусть нашёлся цикл C, такой что $c^f(C) > 0$, $w(C) < 0$. Тогда рассмотрим $h = c^f(C) \cdot flow_C$ — поток в G_f. По лемме о суммах и разностях потоков $f + h$ — поток в G, причём $|f + h| = |f|$, $w(f + h) = w(f) + w(h) = w(f) + c^f(C) \cdot w(C) < w(f)$; значит, f не является потоком минимальной стоимости.

Пусть теперь, наоборот, такого цикла C не нашлось. Пусть f' — любой другой поток в G той же величины, что и f: $|f'| = |f|$. По лемме о суммах и разностях потоков $f' - f$ — поток в G_f; поскольку $|f' - f| = 0$, его декомпозиция — набор циклов; каждый из этих циклов по предположению имеет неотрицательную стоимость. Тогда $w(f') \geqslant w(f)$ для любого потока f', $|f'| = |f|$; значит, f — поток минимальной стоимости. ■

Теорема 11.1.2 — Теорема о дополняющем пути минимальной стоимости. Пусть f — поток минимальной стоимости в сети G, P — дополняющий путь в G_f минимальной стоимости. Для любого $0 \leqslant \delta \leqslant c^f(P)$ поток $f + \delta \cdot flow_P$ является потоком минимальной стоимости.

Доказательство. Пусть g, $|g| = |f| + \delta$ — произвольный поток такой же стоимости, что и $f + \delta \cdot flow_P$. $g - f$ — поток в G_f, его декомпозиция имеет вид

$$g - f = \sum_i \delta_i \cdot flow_{P_i} + \sum_j \sigma_j \cdot flow_{C_j},$$

где P_i — некоторые пути из s в t, C_j — циклы, $\sum_i \delta_i = \delta$.

По лемме о потоке минимальной стоимости, $w(C_j) \geqslant 0$ для любого из циклов C_j. Тогда, поскольку P имеет минимальную стоимость среди всех дополняющих путей,

$$w(g-f) = \sum_i \delta_i \cdot w(P_i) + \sum_j \sigma_j \cdot w(C_j) \geqslant w(P) \cdot \sum_i \delta_i = \delta \cdot w(P).$$

Значит,

$$w(g) \geqslant w(f) + \delta \cdot w(P) = w(f + \delta \cdot flow_P),$$

то есть $f + \delta \cdot flow_P$ — поток минимальной стоимости. ■

11.2 Сеть без отрицательных циклов

Пусть в исходной сети G нет отрицательных циклов (циклов отрицательной стоимости). Тогда, по лемме о потоке минимальной стоимости, нулевой поток является потоком минимальной стоимости (его остаточная сеть совпадает с исходной сетью).

Пусть теперь мы имеем какой-то поток минимальной стоимости f (вначале f — нулевой поток). Поскольку в G_f нет циклов отрицательной стоимости, запустив алгоритм Форда-Беллмана, можно либо проверить, что дополняющих путей в G_f нет, либо найти минимальный по стоимости дополняющий путь P. В первом случае f — максимальный поток по теореме Форда-Фалкерсона, то есть алгоритм можно завершить. Во втором случае, по доказанной выше теореме, $f + c^f(P) \cdot flow_P$ — поток минимальной стоимости; применим к нему те же рассуждения.

Если G — целочисленная сеть, то описанный алгоритм найдёт **целочисленный** максимальный поток минимальной стоимости за время $O(|f|VE)$, где $|f|$ — величина максимального потока в сети.

11.3 Циркуляция минимальной стоимости

Пусть теперь G — сеть без выделенных истока и стока, но с пропускными способностями и стоимостями на рёбрах, а также с добавленными обратными рёбрами.

Определение 11.3.1 *Циркуляция (circulation)* в сети G — это такой набор вещественных чисел, соотвествующих рёбрам графа $\{f_e\}_{e \in E}$, что выполняются следующие условия:

1. $f_e \leqslant c_e$ для любого $e \in E$;
2. $f_e = -f_{e'}$ для любого $e \in E$;
3. $\sum_{e \in E_{in}(v)} f_e = \sum_{e \in E_{out}(v)} f_e = 0$ для любой $v \in V$.

Определение циркуляции совпадает с определением потока нулевой стоимости в сети с истоком и стоком; такие потоки мы тоже иногда будем называть циркуляциями. С другой стороны, в сеть без истока и стока их можно искуственно добавить изолированными вершинами; при этом циркуляция станет потоком нулевой величины. Отсюда следует, что доказанные нами утверждения про потоки можно применять и к циркуляциям: в частности, к циркуляциям применима лемма о суммах и разностях; для циркуляции можно построить декомпозицию на потоки вдоль циклов; циркуляция имеет минимальную стоимость тогда и только тогда, когда в остаточной сети относительно неё нет отрицательных циклов.

Предложение 11.3.1 Задачи поиска циркуляции минимальной стоимости и поиска максимального потока минимальной стоимости сводятся друг к другу за полиномиальное время.

Доказательство. Пусть мы умеем искать максимальный поток минимальной стоимости. Тогда мы умеем искать и циркуляцию минимальной стоимости, поскольку она является максимальным потоком минимальной стоимости в сети с добавленными изолированными истоком и стоком.

Пусть, наоборот, мы умеем искать циркуляцию минимальной стоимости. Тогда найдём какой-нибудь максимальный поток f, не обращая внимания на стоимости рёбер, после чего найдём циркуляцию минимальной стоимости h в остаточной сети G_f. В G_{f+h} нет отрицательных циклов, $|f+h| = |f|$, значит, $f+h$ — максимальный поток минимальной стоимости. ■

Научимся искать циркуляцию минимальной стоимости: пусть вначале f — нулевая циркуляция (равная нулю на каждом ребре). Будем повторять следующие действия: с помощью алгоритма Форда-Беллмана найдём в G_f отрицательный цикл, либо проверим, что их нет. Если отрицательных циклов нет, то текущая циркуляция f — циркуляция минимальной стоимости. Если же нашёлся цикл C, такой что $c^f(C) > 0$, $w(C) < 0$, то заменим f на $f + c^f(C) \cdot flow_C$; это циркуляция меньшей стоимости, чем f. Применим к ней те же рассуждения.

Если все пропускные способности и стоимости рёбер — целые числа, то описанный алгоритм завершится за конечное время, а найденная циркуляция будет целочисленной. Действительно, текущая циркуляция будет целочисленной в любой момент времени, а стоимость текущей циркуляции $w(f)$ на каждом шаге уменьшается хотя бы на единицу. Тогда число итераций не превышает модуля минимально возможной стоимости циркуляции, то есть, например, $O(EWU)$, где $U = \max_e c_e$ — максимальная пропускная способность ребра в сети, а $W = \max_e |w_e|$ — максимальный модуль стоимости ребра в сети. Таким образом, мы научились искать циркуляцию минимальной стоимости за $O(VE^2WU)$.

Поиск макс. потока мин. стоимости в сети с отрицательными циклами

Сначала найдём циркуляцию (поток нулевой величины) минимальной стоимости, алгоритмом, описанным выше. Поскольку в её остаточной сети нет отрицательных циклов, дальше можно, как и раньше, искать дополняющий путь мин. стоимости алгоритмом Форда-Беллмана, и увеличивать вдоль него поток, пока не окажется, что дополняющих путей в остаточной сети нет. Получаем алгоритм с временем работы $O(VE(|f|+EWU)) = O(VE(EU+EWU)) = O(VE^2WU)$.

R Заметим, что можно действовать и в обратном порядке: сначала не обращать внимания на стоимости и просто найти какой-нибудь максимальный поток, а после этого насыщать циклы отрицательной стоимости в остаточной сети, пока не окажется, что их там нет.

11.4 Циркуляция с избытками и недостатками

Рассмотрим ещё одну вспомогательную задачу. Пусть G — сеть без истока и стока, в которой у каждого ребра есть пропускная способность и стоимость (а также определённое, как обычно, обратное ребро), а у каждой вершины v есть *потребность* — вещественное число d_v. Если $d_v > 0$, то в вершине имеется *недостаток (deficit)* d_v единиц потока; если $d_v < 0$, то, наоборот, в вершине имеется *избыток (excess)* $-d_v$ единиц потока. Требуется направить поток из вершин с избытком в вершины с недостатком, заплатив при этом как можно меньше. Более формально:

Определение 11.4.1 *Циркуляция (circulation)* в сети G с избытками и недостатками — это такой набор вещественных чисел, соответствующих рёбрам графа $\{f_e\}_{e \in E}$, что выполняются следующие условия:

1. $f_e \leqslant c_e$ для любого $e \in E$;
2. $f_e = -f_{e'}$ для любого $e \in E$;
3. $\sum_{e \in E_{in}(v)} f_e \left(= \sum_{e \in E_{in}^{fwd}(v)} f_e - \sum_{e \in E_{out}^{fwd}(v)} f_e \right) = d_v$ для любой $v \in V$.

Заметим, что если циркуляция существует, то

$$\sum_{v \in V} d_v = \sum_{v \in V} \sum_{e \in E_{in}(v)} f_e = \sum_{e \in E} f_e = \sum_{e \in E^{fwd}} f_e + \sum_{e \in E^{bwd}} f_e = 0.$$

Значит, равенство

$$\sum_{\substack{v \in V \\ d_v > 0}} d_v = \sum_{\substack{v \in V \\ d_v < 0}} -d_v$$

является необходимым для существования циркуляции. Введём обозначение

$$D = \sum_{\substack{v \in V \\ d_v > 0}} d_v.$$

Задача поиска циркуляции мин. стоимости в сети с избытками и недостатками является обобщением задачи поиска циркуляции мин. стоимости (такая задача соответствует $d_v = 0$ для всех v). Оказывается, эту задачу по-прежнему можно свести к задаче поиска макс. потока мин. стоимости.

Предложение 11.4.1 Задачи поиска макс. потока мин. стоимости и циркуляции мин. стоимости в сети с избытками и недостатками сводятся друг к другу за полиномиальное время.

Доказательство. Достаточно научиться сводить вторую задачу к первой. Для этого добавим в сеть две новых вершины — исток s и сток t; для каждой вершины v с $d_v > 0$ проведём ребро из v в t с нулевой стоимостью и пропускной способностью d_v; для каждой вершины v с $d_v < 0$ проведём ребро из s в v с нулевой стоимостью и пропускной способностью $-d_v$; разумеется, вместе с каждым ребром мы добавляем и обратное к нему ребро, определённое как обычно.

Пусть (S, T) — (s, t)-разрез, в котором $T = \{t\}$. Поскольку $c(S, T) = D$, величина максимального потока в построенной сети не превосходит D. При этом существует сохраняющая стоимость биекция между потоками величины D в построенной сети и циркуляциями в исходной сети (эта биекция заключается в удалении/добавлении вершин s, t и инцидентных им рёбер).

Найдём в построенной сети макс. поток мин. стоимости f. Если $|f| < D$, то циркуляции, удовлетворяющей условиям на избытки и недостатки, не существует. Если же $|f| = D$, то соответствующая потоку f циркуляция в исходной сети является циркуляцией мин. стоимости. ■

Следствие 11.4.2 Задачи поиска макс. потока мин. стоимости, циркуляции мин. стоимости, циркуляции мин. стоимости в сети с избытками и недостатками сводятся к задаче поиска макс. потока мин. стоимости в сети **с неотрицательными стоимостями рёбер** за полиномиальное время.

Доказательство. Достаточно научиться строить сведение для задачи поиска циркуляции мин. стоимости в сети с избытками и недостатками.

Итак, пусть G — сеть с избытками и недостатками. Рассмотрим функцию на рёбрах g, определённую следующим образом:

$$g_e = \begin{cases} c_e, & \text{если } w_e < 0, \\ -c_{e'}, & \text{если } w_e > 0 \text{ (то есть } w_{e'} < 0), \\ 0, & \text{иначе,} \end{cases}$$

а также вспомогательную сеть G' с пропускными способностями $c'_e = c_e - g_e$ и потребностями

$$d'_v = d_v - \sum_{e \in E_{in}(v)} g_e.$$

Неформально эту сеть можно интерпретировать как исходную сеть, в которой были принудительно насыщены все рёбра отрицательной стоимости. Заметим, что имеет место биекция между циркуляциями в G' и в G: f — циркуляция в G' тогда и только тогда, когда $f + g$ — циркуляция в G.

Задачу поиска циркуляции мин. стоимости в G' мы умеем сводить к задаче поиска макс. потока мин. стоимости. При этом любое ребро отрицательной стоимости в G' имеет нулевую пропускную способность, а при сведении в сеть будут добавлены лишь рёбра с нулевой стоимостью. Значит, осталось найти макс. поток мин. стоимости в сети, где все рёбра с положительной пропускной способностью имеют неотрицательную стоимость. ■

Таким образом, мы научились решать все три задачи за время $O(|f|VE)$, где $|f|$ — величина максимального потока во вспомогательной сети без ненасыщенных отрицательных рёбер. Поскольку $|f|$ всегда можно оценить как $O(EU)$, получаем оценку времени работы $O(VE^2U)$. Отметим, что эта оценка верна для сетей с целыми пропускными способностями **и произвольными вещественными стоимостями**.

11.5 Метод потенциалов

Алгоритм Дейкстры работает быстрее алгоритма Форда-Беллмана, но, к сожалению, требует, чтобы веса всех рёбер были неотрицательны. Заметим, что в задаче поиска макс. потока мин. стоимости это условие практически всегда не выполняется: даже если все стоимости рёбер в исходной сети были неотрицательны, после первого же увеличения потока вдоль дополняющего пути появятся обратные рёбра с положительной остаточной пропускной способностью и отрицательной стоимостью.

Пусть каждой вершине v назначен *потенциал* — некоторое вещественное число φ_v. Введём потенциальные стоимости рёбер: для ребра $e : a \to b$

$$w^{\varphi}_e = w_e + \varphi_a - \varphi_b.$$

Заметим, что потенциальная стоимость любого пути из s в t отличается от его обычной стоимости на одну и ту же величину $\varphi_s - \varphi_t$: пусть P — путь из s в t, состоящий из рёбер $e_1 : s \to v_1, e_2 : v_1 \to v_2, \ldots, e_m : v_{m-1} \to t$, тогда

$$w^{\varphi}(P) = w^{\varphi}_{e_1} + \cdots + w^{\varphi}_{e_m} = w_{e_1} + \varphi_s - \varphi_{v_1} + \cdots + w_{e_m} + \varphi_{v_{m-1}} - \varphi_t = w(P) + \varphi_s - \varphi_t.$$

В частности, это означает, что дополняющий путь минимальной стоимости является и дополняющим путём минимальной потенциальной стоимости (и наоборот). Аналогично, потенциальная стоимость любого цикла равна его исходной стоимости.

Предложение 11.5.1 Пусть f — поток в сети G, φ — такие потенциалы, что $w_e^\varphi \geqslant 0$ для любого такого ребра e, что $c_e^f > 0$. Пусть d_v — потенциальное расстояние от s до v в остаточной сети G_f, P — дополняющий путь мин. стоимости в G_f, $0 \leqslant \delta \leqslant c^f(P)$, $h = f + \delta \cdot flow_P$. Тогда, если $\chi_v = \varphi_v + d_v$, то $w_e^\chi \geqslant 0$ для любого такого ребра e, что $c_e^h > 0$.

Доказательство. Пусть $e : a \to b$ — такое ребро, что $c_e^h > 0$. Если $c_e^f > 0$, то выполняется неравенство

$$d_b \leqslant d_a + w_e^\varphi,$$

тогда

$$w_e^\chi = w_e^\varphi + d_a - d_b \geqslant 0.$$

Пусть теперь $c_e^f = 0$. Поскольку $c_e^h > 0$, обратное ребро $e' : b \to a$ лежало на дополняющем пути мин. стоимости P, вдоль которого был увеличен поток. Тогда выполняется равенство

$$d_a = d_b + w_{e'}^\varphi = d_b - w_e^\varphi,$$

откуда

$$w_e^\chi = w_e^\varphi + d_a - d_b = 0.$$

■

Сеть с неотрицательными стоимостями рёбер

Пусть G — целочисленная сеть с неотрицательными стоимостями рёбер. Доказанное предложение позволяет искать макс. поток мин. стоимости в G с помощью алгоритма Дейкстры. Действительно, возьмём вначале нулевой поток и нулевые потенциалы. Будем искать доп. путь мин. стоимости алгоритмом Дейкстры, использующим потенциальные стоимости рёбер. После увеличения потока вдоль доп. пути, для каждой вершины v увеличим её потенциал на расстояние от s до v, найденное алгоритмом Дейкстры. Тогда, по предложению, новые потенциальные стоимости всех ненасыщенных рёбер снова будут неотрицательны, значит, можно повторить те же действия. Получаем алгоритм с временем работы $O(|f|(V \log V + E))$.

Сеть с отр. рёбрами, но без отр. циклов

Пусть в целочисленной сети G могут быть отрицательные по стоимости рёбра, но нет отрицательных циклов. Запустим один раз алгоритм Форда-Беллмана и найдём расстояния от истока до остальных вершин; эти расстояния и возьмём в качестве начальных потенциалов. Любое ребро с положительной пропускной способностью будет иметь неотрицательную потенциальную стоимость, так как алгоритм Форда-Беллмана проводил релаксацию этого ребра. Получаем алгоритм с временем работы $O(VE + |f|(V \log V + E))$.

Произвольная целочисленная сеть

Пусть теперь G — произвольная целочисленная сеть. Сначала найдём циркуляцию мин. стоимости в G, пользуясь сведением к задаче поиска макс. потока мин. стоимости во вспомогательной сети G' с неотрицательными стоимостями рёбер; эту задачу мы решим с помощью алгоритма Дейкстры методом, описанным выше. После этого вернёмся к исходной сети и найдём в ней макс. поток мин. стоимости, пользуясь тем же методом (но начиная не с нулевого потока и нулевых потенциалов, а с уже найденной нами циркуляции мин. стоимости и потенциалов, полученных при её поиске).

Поскольку величину макс. потока и в сети G, и в сети G' можно оценить, как $O(EU)$, получаем алгоритм с временем работы $O(EU(V \log V + E))$.

11.6 Масштабирование потока

К получившемуся алгоритму снова можно применить технику масштабирования по пропускным способностям. Нам будет удобнее рассмотреть следующую вариацию масштабирования: на фазе масштабирования $k = i$ мы будем использовать пропускные способности в 2^i раз меньше настоящих: $c(i)_e = \lfloor c_e / 2^i \rfloor$. При переходе от фазы $k = i$ к фазе $k = i - 1$ будем удваивать текущий поток по каждому ребру; поскольку пропускные способности хотя бы удваиваются, это не нарушит свойств потока. Не нарушится и тот факт, что на каждой фазе масштабирования поток может увеличиться не более $2|E|$ раз.

R Отличие от того, как мы делали масштабирование раньше, фактически заключается в том, что теперь на фазе $k = i$ мы увеличиваем поток на величину, **кратную** 2^i, а не просто больше или равную 2^i.

Будем сохранять потенциалы вершин между фазами (то есть в очередной фазе начинать не с нулевых потенциалов, а с тех, что были в конце предыдущей фазы). Заметим, что в начале очередной фазы масштабирования в сети могут появиться ненасыщенные рёбра отрицательной потенциальной стоимости. Однако каждое такое ребро будет иметь остаточную пропускную способность, равную единице: действительно, в конце предыдущей фазы любое ребро с отрицательной потенциальной стоимостью было насыщено, значит, в начале текущей фазы величина потока и пропускная способность на таком ребре могут отличаться лишь в младшем бите.

Как обычно, мы принудительно насытим все такие рёбра, а полученную задачу в сети с избытками и недостатками сведём к задаче поиска макс. потока мин. стоимости в сети без отрицательных рёбер. Заметим, что из сказанного выше следует, что сумма недостатков по всем вершинам не превышает $|E|$. Значит, величина макс. потока во вспомогательной сети тоже не превышает $|E|$.

Таким образом, на каждой фазе масштабирования мы будем решать две задачи поиска макс. потока мин. стоимости; в первой из них величина макс. потока не превосходит $|E|$, во второй — $2|E|$. Значит, время работы одной фазы масштабирования — $O(E(V \log V + E))$.

Получаем алгоритм, решающий задачу поиска макс. потока мин. стоимости на любой целочисленной сети за $O(E \log U(V \log V + E))$.

R Существуют и алгоритмы, решающие задачу за полиномиальное время, не зависящее ни от стоимостей, ни от пропускных способностей (то есть, в частности, корректно работающие и на сетях с нецелыми пропускными способностями). Например, известен алгоритм с временем работы $O(E \log V(V \log V + E))$ (Orlin, 1988, [23]).

11.7 Задача о назначениях

Обсудим, как с помощью изученных алгоритмов можно решать ещё одну взвешенную вариацию задачи о максимальном паросочетании. *Задача о назначениях (assignment problem)* формулируется так: дан полный двудольный граф с равными по размеру долями и весами на рёбрах; требуется найти совершенное паросочетание минимального суммарного веса.

Рассмотрим ту же сеть, с помощью которой мы решали задачу о максимальном паросочетании, и назначим рёбрам исходного графа стоимости, равные данным в задаче весам; рёбрам, инцидентным истоку и стоку, назначим нулевую стоимость. Имеется биекция между макс. потоками в этой сети и совершенными паросочетаниями в исходном графе; эта биекция сохраняет стоимость. Значит, достаточно найти в этой сети макс. поток мин. стоимости.

В сети могут иметься рёбра отрицательной стоимости, но нет отрицательных циклов (поскольку нет циклов с положительной пропускной способностью вообще). Значит, задачу можно решить за $O(VE + |f|(V \log V + E)) = O(V^3)$; на самом деле достаточно использовать квадратичную реализацию алгоритма Дейкстры: $O(VE + |f|(V^2 + E)) = O(V^3)$.

Библиография

Книги

[1] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology* (цитируется на страницах 20—22).

[2] Donald E. Knuth. *The Art of Computer Programming (Third edition), Volume 2* (цитируется на странице 15).

Статьи

[3] Mohamed Ibrahim Abouelhoda, Stefan Kurtz и Enno Ohlebusch. «Replacing suffix trees with enhanced suffix arrays». *Journal of Discrete Algorithms* 2.1 (2004), страницы 53—86. DOI: 10.1016/s1570-8667(03)00065-0 (цитируется на странице 46).

[4] Alfred V. Aho и Margaret J. Corasick. «Efficient String Matching: An Aid to Bibliographic Search». *Commun. ACM* 18.6 (1975), страницы 333—340. DOI: 10.1145/360825.360855 (цитируется на странице 26).

[5] C. Berge. «TWO THEOREMS IN GRAPH THEORY». *Proceedings of the National Academy of Sciences* 43.9 (1957), страницы 842—844. DOI: 10.1073/pnas.43.9.842 (цитируется на странице 49).

[6] Robert S. Boyer и J. Strother Moore. «A Fast String Searching Algorithm». *Commun. ACM* 20.10 (1977), страницы 762—772. DOI: 10.1145/359842.359859 (цитируется на странице 19).

[7] Maxime Crochemore и Dominique Perrin. «Two-Way String-Matching». *J. ACM* 38.3 (1991), страницы 650—674. DOI: 10.1145/116825.116845 (цитируется на странице 23).

[8] König Dénes. «Gráfok és mátrixok». *Matematikai és Fizikai Lapok* 38 (1931), страницы 116—119 (цитируется на странице 51).

[9] Yefim Dinitz. «Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation». *Soviet Math. Dokl.* 11 (1970), страницы 1277—1280 (цитируется на странице 66).

[10] Jack Edmonds и Richard M. Karp. «Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems». *Journal of the ACM* 19.2 (1972), страницы 248—264. DOI: 10.1145/321694.321699 (цитируется на странице 65).

[11] Eugene Egerváry. «Matrixok kombinatorius tulajdonságairol». *Matematikai és Fizikai Lapok* 38 (1931), страницы 16—28 (цитируется на странице 51).

[12] L. R. Ford и D. R. Fulkerson. «Maximal Flow Through a Network». *Canadian Journal of Mathematics* 8 (1956), страницы 399—404. DOI: 10.4153/cjm-1956-045-5 (цитируется на странице 61).

[13] Martin Fürer. «Faster Integer Multiplication». STOC '07 (2007), страницы 57—66. DOI: 10.1145/1250790.1250800 (цитируется на странице 13).

[14] D. Gale и L. S. Shapley. «College Admissions and the Stability of Marriage». *The American Mathematical Monthly* 69.1 (1962), страница 9. DOI: 10.2307/2312726 (цитируется на странице 53).

[15] Gaston Gonnet, Ricardo Baeza-Yates и Tim Snider. «New Indices for Text: Pat Trees and Pat Arrays». 1992, страницы 66—82 (цитируется на странице 30).

[16] David Harvey и Joris Van Der Hoeven. «Integer multiplication in time O(n log n)» (2019). Препринт. URL: https://hal.archives-ouvertes.fr/hal-02070778 (цитируется на странице 13).

[17] R. M. Karp и M. O. Rabin. «Efficient randomized pattern-matching algorithms». *IBM Journal of Research and Development* 31.2 (1987), страницы 249—260 (цитируется на странице 21).

[18] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa и Kunsoo Park. «Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications». *Combinatorial Pattern Matching*. 2001, страницы 181—192 (цитируется на странице 33).

[19] Donald E. Knuth, Jr. James H. Morris и Vaughan R. Pratt. «Fast Pattern Matching in Strings». *SIAM Journal on Computing* 6.2 (1977), страницы 323—350. DOI: 10.1137/0206024 (цитируется на странице 18).

[20] Udi Manber и Gene Myers. «Suffix Arrays: A New Method for on-Line String Searches». *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '90. 1990, страницы 319—327 (цитируется на странице 30).

[21] Silvio Micali и Vijay V. Vazirani. «An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs». *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. 1980. DOI: 10.1109/sfcs.1980.12 (цитируется на странице 51).

[22] Ge Nong, Sen Zhang и Wai Hong Chan. «Linear Suffix Array Construction by Almost Pure Induced-Sorting». *2009 Data Compression Conference*. 2009, страницы 193—202. DOI: 10.1109/dcc.2009.42 (цитируется на странице 39).

[23] James Orlin. «A Faster Strongly Polynomial Minimum Cost Flow Algorithm». *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. 1988, страницы 377—387. DOI: 10.1145/62212.62249 (цитируется на странице 78).

[24] James B. Orlin. «Max Flows in $O(nm)$ Time, or Better». *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '13. 2013, страницы 765—774. DOI: 10.1145/2488608.2488705 (цитируется на странице 67).

[25] A. Schönhage и V. Strassen. «Schnelle Multiplikation großer Zahlen». *Computing* 7.3-4 (1971), страницы 281—292. DOI: 10.1007/bf02242355 (цитируется на странице 13).

[26] E. Ukkonen. «On-line construction of suffix trees». *Algorithmica* 14.3 (1995), страницы 249—260. DOI: 10.1007/bf01206331 (цитируется на странице 45).

[27] Peter Weiner. «Linear pattern matching algorithms». *14th Annual Symposium on Switching and Automata Theory*. 1973. DOI: 10.1109/swat.1973.13 (цитируется на странице 45).

[28] А. В. Карзанов. «О нахождении максимального потока в сетях специального вида и некоторых приложениях». *Математические вопросы управления производством* 5 (1973), страницы 81—94 (цитируется на странице 69).