СПб ВШЭ, 3-й курс, весна 2025
Конспект лекций по алгоритмам

Собрано 2 июня 2025 г. в 13:18

Содержание

Лекция #1: FFT и его друзья

1-я пара, весна 2025

1.1. FFT

1.1.1. Прелюдия

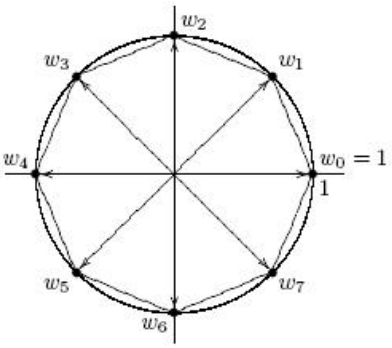
Пусть есть многочлены $A(x) = \sum a_i x^i$ и $B(x) = \sum b_i x^i$.
Посчитаем их значения в точках $x_1, x_2, \ldots, x_n$: $A(x_i) = fa_i, B(x_i) = fb_i$.
Значения $C(x) = A(x)B(x)$ в точках x_i можно получить за линейное время:

$$fc_i = C(x_i) = A(x_i)B(x_i) = fa_i fb_i$$

Схема быстрого умножения многочленов:

$$a_i, b_i \xrightarrow{\mathcal{O}(n \log n)} fa_i, fb_i \xrightarrow{\mathcal{O}(n)} fc_i = fa_i fb_i \xrightarrow{\mathcal{O}(n \log n)} c_i$$

Осталось подобрать правильные точки x_i.



FFT расшифровывается Fast Fourier Transform и за $\mathcal{O}(n \log n)$ вычисляет значения многочлена в комплексных точках $w_j = e^{\frac{2\pi i j}{n}}$ для $n = 2^k$ (то есть, только для степеней двойки).

Что нужно помнить про комплексные числа?
При умножении комплексных чисел углы складываются, длины перемножаются.
В частности, если обозначить $w = e^{\frac{2\pi i}{n}} = \cos\frac{2\pi i}{n} + i \sin\frac{2\pi i}{n}$, то $w_j = w^j$ (все корни из единицы – это степени главного корня, и они образуют циклическую группу). Также $w^{-j} = w^{n-j}$.

1.1.2. Собственно FFT

$A(x) = \sum a_i x^i = (a_0 + x^2 a_2 + x^4 a_4 + \ldots) + x(a_1 x + a_3 x^3 + a_5 x^5 + \ldots) = B(x^2) + xC(x^2)$ – обозначили все чётные коэффициенты многочлена A многочленом B, а нечётные соответственно C.

Посчитаем рекурсивно $B(w_j)$ и $C(w_j)$, зная их, за $\mathcal{O}(n)$ посчитаем $A(w_j) = B(w_j) + w_j C(w_j)$.

Заметим, что $\forall j\ w_j = w_{j \bmod n} \Rightarrow \forall j\ w_j^2 = w_{n/2+j}^2 \Rightarrow B$ и C нужно считать только в $\frac{n}{2}$ точках.

Итого алгоритм:

```
1 def FFT(a):
2     n = len(a)
3     if n == 1: return a[0] # посчитать значение многочлена A(x) ≡ a[0] в точке x = 1
4     for j=0..n-1: (j%2 ? c : b)[j / 2] = a[j]
5     b, c = FFT(b), FFT(c) # самое важное - две ветки рекурсии
6     for j=0..n-1: a[j] = b[j % (n/2)] + exp(2πi*j/n) * c[j % (n/2)]
7     return a
```

Время работы $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

1.1.3. Крутая нерекурсивная реализация FFT

Чтобы преобразование работало быстро, нужно заранее предподсчитать все $w_j = e^{\frac{2\pi ij}{n}}$.

Заметим, что b и c можно хранить прямо в массиве a. Тогда получается, что на прямом ходу рекурсии мы просто переставляем местами элементы a, на обратном ходу рекурсии делаем какие-то полезные действия. Число a_i перейдёт на позицию $a_{rev(i)}$, где $rev(i)$ – перевёрнутая битовая запись i. Кстати, $rev(i)$ мы уже умеем считать динамикой для всех i.

При реализации на C++ можно использовать стандартные комплексные числа `complex<double>`, но свои рукописные будут работать немного быстрее.

```
const int K = 20, N = 1 << K; // N - ограничение на длину результата умножения многочленов
complex<double> root[N];
int rev[N];
void init():
    for (int j = 0; j < N; j++):
        root[j] = exp(2πi*j/N); // cos(2πj/N), sin(2πj/N)
        rev[j] = rev[j >> 1] + ((j & 1) << (K - 1));
```

Теперь, корни из единицы степени k хранятся в `root[j*N/k]`, $j \in [0, k)$.
Доступ к памяти при этом не последовательный, проблемы с кешом.
Чтобы посчитать все корни, мы $2N$ раз вычисляли тригонометрические функции.

- **Улучшенная версия вычисления корней**

```
for (int k = 1; k < N; k *= 2):
    num tmp = exp(πi/k);
    root[k] = {1, 0}; // в root[k..2k) хранятся первые k корней степени 2k
    for (int i = 1; i < k; i++)
        root[k+i] = (i&1) ? root[(k+i) >> 1] * tmp : root[(k+i) >> 1];
```

Теперь код собственно преобразования Фурье может выглядеть так:

```
FFT(a): // a → f = FFT(a)
    vector<complex> f(N);
    for (int i = 0; i < N; i++) // прямой ход рекурсии превратился в один for =)
        f[rev[i]] = a[i];
    for (int k = 1; k < N; k *= 2) // пусть уже посчитаны FFT от кусков длины k
        for (int i = 0; i < N; i += 2 * k) // [i..i+k) [i+k..i+2k) → [i..i+2k)
            for (int j = 0; j < k; j++): // оптимально написанный главный цикл FFT
                num tmp = root[k + j] * f[i + j + k]; // root[] из «улучшенной версии»
                f[i + j + k] = f[i + j] - tmp; // w_{j+k} = -w_j при n = 2k
                f[i + j] = f[i + j] + tmp;
    return f;
```

1.1.4. Обратное преобразование

Обозначим $w = e^{2\pi i/n}$. Нам нужно из
$f_0 = a_0 + a_1 + a_2 + a_3 + \dots$
$f_1 = a_0 + a_1 w + a_2 w^2 + a_3 w^3 + \dots$
$f_2 = a_0 + a_1 w^2 + a_2 w^4 + a_3 w^3 + \dots$

научиться восстанавливать коэффициенты $a_0, a_1, a_2, \dots$

Заметим, что $\forall j \neq 0 \sum\limits_{k=0}^{n-1} w^{jk} = 0$ (сумма геометрической прогрессии).
И напротив при $j = 0$ получаем $\sum\limits_{k=0}^{n-1} w^{jk} = \sum 1 = n$.
Поэтому $f_0 + f_1 + f_2 + \dots = a_0 n + a_1 \sum_k w^k + a_2 \sum_k w^{2k} + \dots = \boxed{a_0 n}$
Аналогично $f_0 + f_1 w^{-1} + f_2 w^{-2} + \dots = \sum_k a_0 w^{-k} + a_1 n + a_2 \sum_k w^k + \dots = \boxed{a_1 n}$
И в общем случае $\sum_k f_k w^{-jk} = \boxed{a_j n}$
Рассмотрим $F(x) = f_0 + x f_1, x^2 f_2 + \dots \Rightarrow F(w^{-j}) = a_j n$, похоже на `FFT(f)`.
Осталось заметить, что множества чисел $w^{-j} = w^{n-j} \Rightarrow$

```
1 FFT_inverse(f): // f → a
2     a = FFT(f)
3     reverse(a + 1, a + N) // w^j ↔ w^{-j}
4     for (int i = 0; i < N; i++) a[i] /= N;
5     return a;
```

1.1.5. Два в одном

Часто коэффициенты многочленов – вещественные или даже целые числа.
Если у нас есть многочлены $A(x), B(x) \in \mathbb{R}[x]$, возьмём числа $c_j = a_j + ib_j$, коэффициенты $C(x) = A(x) + iB(x)$, посчитаем $fc = FFT(c)$. Тогда по f за $\mathcal{O}(n)$ можно восстановить fa и fb.

Для этого вспомним про сопряжения комплексных чисел:
$\overline{x + iy} = x - iy$, $\overline{u \cdot v} = \overline{u} \cdot \overline{v}$, $w^{n-j} = w^{-j} = \overline{w^j} \Rightarrow \overline{fc_{n-j}} = \overline{C(w^{n-j})} = \overline{C}(w^j) = A(w^j) - iB(w^j) \Rightarrow$
$fc_j + \overline{fc_{n-j}} = 2A(w^j) = 2 \cdot fa_j$. Аналогично $fc_j - \overline{fc_{n-j}} = 2B(w^j) = 2i \cdot fb_j$.

Итого для умножения двух многочленов можно использовать не 3 вызова FFT, а 2.

1.1.6. Умножение чисел, оценка погрешности

Число длины n в системе счисления 10 $\rightarrow$ система счисления 10^k $\rightarrow$ многочлен длины n/k.
Умножения многочленов такой длины будет работать за $\frac{n}{k} \log \frac{n}{k}$.
Отсюда возникает вопрос, какое максимальное k можно использовать?
Коэффициенты многочлена-произведения будут целыми числами до $(10^k)^2 \cdot \frac{n}{k}$.
Чтобы в типе `double` целое число хранилось с погрешностью меньше 0.5 (тогда мы его сможем правильно округлить к целому), оно должно быть не более 10^{15}.
Получаем при $n \leqslant 10^6$, что $(10^k)^2 \cdot 10^6 / k \leqslant 10^{15} \Rightarrow k \leqslant 4$.
Аналогично для типа `long double` имеем $(10^k)^2 \cdot 10^6 / k \leqslant 10^{18} \Rightarrow k \leqslant 6$.
Это оценка сверху, предполагающая, что само FFT погрешность не накапливает... на самом деле эта оценка очень близка к точной.

1.2. Разделяй и властвуй

1.2.1. Перевод между системами счисления

Задача: перевести число X длины $n = 2^k$ из a-ичной системы счисления в b-ичную.

Разобьём число X на $\frac{n}{2}$ старших цифр и $\frac{n}{2}$ младших цифр: $X = X_0 \cdot a^{n/2} + X_1 \Rightarrow$

$$F(X) = F(X_0)F(a^{n/2}) + F(X_1)$$

Умножение за $\mathcal{O}(n \log n)$ и сложение за $\mathcal{O}(n)$ выполняются в системе счисления b.
Предподсчёт $F(a^1), F(a^2), F(a^4), F(a^8), \ldots, F(a^n)$ займёт $\sum_k \mathcal{O}(2^k k) = \mathcal{O}(n \log n)$ времени.
Итого $T(n) = 2T(n/2) + \mathcal{O}(n \log n) = \mathcal{O}(n \log^2 n)$.

1.2.2. Деление многочленов с остатком

Задача: даны $A(x), B(x) \in \mathbb{R}[x]$, найти $Q(x), R(x)$: $\deg R < \deg B \ \wedge \ A(x) = B(x)Q(x) + R(x)$.

Зная Q мы легко найдём R, как $A(x) - B(x)Q(x)$ за $\mathcal{O}(n \log n)$. Сосредоточимся на поиске Q.
Пусть $\deg A = \deg B = n$, тогда $Q(x) = \frac{a_n}{b_n}$. То есть, $Q(x)$ можно найти за $\mathcal{O}(1)$.
Из этого мы делаем вывод, что Q зависит не обязательно от всех коэффициентов A и B.

Lm 1.2.1. $\deg A = m, \deg B = n \Rightarrow \deg Q = m - n$, и Q зависит только от $m{-}n{+}1$ коэффициентов A и $m{-}n{+}1$ коэффициентов B.

Доказательство. У A и $B \cdot Q$ должны совпадать $m{-}n{+}1$ старший коэффициент ($\deg R < n$). В этом сравнении участвуют только $m{-}n{+}1$ старших коэффициентов A. При домножение B на $x^{\deg Q}$, сравнятся как раз $m{-}n{+}1$ старших коэффициентов A и B. При домножении B на меньшие степени x, в сравнении будут участвовать лишь какие-то первые из этих $m{-}n{+}1$ коэффициентов. ■

Теперь будем решать задачу: даны n старших коэффициентов A и B, найти такой C из n коэффициентов, что у A и BC совпадает n старших коэффициентов. Давайте считать, что младшие коэффициенты лежат в первых ячейках массива.

```
1 Div(int n, int *A, int *B)
2     C = Div(n/2, A + n/2, B + n/2) // нашли старших n/2 коэффициентов ответа
3     A' = Subtract(n, A, n + n/2 - 1, Multiply(C, B))
4     D = Div(n/2, A', B + n/2) // сейчас A' состоит из n/2 не нулей и n/2 нулей
5     return concatenate(D, C) // склеили массивы коэффициентов
```

Здесь `Subtract` – хитрая функция. Она знает длины многочленов, которые ей передали, и сдвигает вычитаемый многочлен так, чтобы старшие коэффициенты совместились.

1.2.3. Вычисление значений в произвольных точках

Задача. Дан многочлен $A(x)$, $\deg A = n$ и точки $x_1, x_2, \ldots x_n$. Найти $A(x_1), A(x_2), \ldots A(x_n)$.

Вспомним теорему Безу: $A(w) = A(x) \bmod (x - w)$.
Обобщение: $B(x) = A(x) \bmod (x - x_1)(x - x_2)\ldots(x - x_n) \ \Rightarrow \ \forall j \ B(x_j) = A(x_j)$

```
1 def Evaluate(n, A, x[]): # n = 2^k
2     if n == 1: return list(A[0])
3     return Evaluate(n/2, A mod (x-x_1)·...·(x-x_{n/2}), [x_1, ... , x_{n/2}]) +
4         Evaluate(n/2, A mod (x-x_{n/2+1})·...·(x-x_n), [x_{n/2+1}, ... , x_n])
```

Итого $T(n) = 2T(n/2)+\mathcal{O}(\texttt{div}(n))$. Если деление реализовано за $\mathcal{O}(n \log n)$, получим $\mathcal{O}(n \log^2 n)$.

1.2.4. Интерполяция

Задача. Даны пары $(x_1, y_1), \ldots, (x_n, y_n)$. Найти многочлен A: $\deg A = n-1$, $\forall i\ A(x_i) = y_i$.

Сделаем интерполяцию по Ньютону методом разделяй и властвуй.
Сперва найдём интерполяционный многочлен B для $(x_1, y_1), (x_2, y_2), \ldots, (x_{n/2}, y_{n/2})$.

$$A = B + C \cdot D, \text{ где } D = \prod_{j=1..\frac{n}{2}} (x - x_j), \text{ а } C \text{ нужно найти}$$

Подгоним правильные значения в точках $x_{n/2+1}, \ldots, x_n$, вычислим b_j, d_j – значения B и D в точках $x_{n/2+1}, \ldots, x_n \Rightarrow C$ – интерполяционный многочлен точек $(x_j, -\frac{b_j}{d_j})$ при $j = \frac{n}{2}+1 \ldots n$.

Итого $T(n) = 2T(n/2) + 2\mathcal{O}(\texttt{evaluate}(n/2))$. При $\texttt{evaluate}(n) = \mathcal{O}(n \log^2 n)$ имеем $\mathcal{O}(n \log^3 n)$.

1.2.5. Извлечение корня

Дан многочлен $A(x)$: $\deg A \equiv 0 \bmod 2$. Задача – найти $R(x)$: $\deg(A - R^2)$ минимальна.
Пусть мы уже нашли старшие k коэффициентов R, обозначим их R_k. Найдём $2k$ коэфф-тов: $R_{2k} = R_k x^k + X, R_{2k}^2 = R_k^2 x^{2k} + 2R_k X \cdot x^k + X^2$. Правильно подобрав X, мы можем "обнулить" k коэффициентов $A - R_{2k}^2$, для этого возьмём $X = (A - R_k^2 x^{2k})/(2R_k)$. В этом частном нам интересны только k старших коэффициентов, поэтому переход от R_k к R_{2k} происходит за $\mathcal{O}(\texttt{mul}(k) + \texttt{div}(k))$. Итого суммарное время на извлечение корня – $\mathcal{O}(\texttt{div}(n))$.

1.3. Литература

[sankowski]. Слайды по FFT и всем идеям разделяй и властвуй.
[e-maxx]. Про FFT и оптимизации к нему.
[codeforces]. Задачи на тему FFT.
[vk]. Краткий конспект похожих идей от Александра Кулькова.

Лекция #1: Действия над многочленами

1-я пара, весна 2025

1.4. Над $\mathbb{F}_2$

Умножение/деление/gcd можно делать битовым сжатием за $\approx \frac{nm}{w}$, где w – word size.

• Хранение

$A(x) = a_0 + a_1 x + \cdots + a_n x^n \;\to$ `N = n + 1; bitset<N> a`
Обозначения: $n = \deg A$, $m = \deg B$, $N = \deg A + 1$, $M = \deg B + 1$.
Многочлен степени n = массив коэффициентов длины N.

• Умножение

```
1 for i=0..n
2     if a[i]
3         c ^= b << i
```

Время работы: $\mathcal{O}(n \cdot \lceil \frac{m}{w} \rceil)$. Можно n заменить на «*число не нулей в* `a`».

• Деление

```
1 for i=n..m
2     if a[i]
3         a ^= b << (i-m), c[i-m] = 1
```

Результат: в «`a`» лежит остаток, в «`c`» частное.
Время работы: $\mathcal{O}((n-m) \cdot \lceil \frac{m}{w} \rceil)$.

• gcd

Запускаем Евклида. Один шаг Евклида – деление. Деление работает за $\mathcal{O}((n-m) \cdot \lceil \frac{m}{w} \rceil)$ и уменьшает n на $n-m \Rightarrow$ суммарно все деления отработают за $\mathcal{O}((n-m) \cdot \lceil \frac{m}{w} \rceil + \frac{m^2}{w}) = \mathcal{O}(\frac{nm}{w})$.

1.5. Умножение многочленов

Над произвольным кольцом умеем за $\mathcal{O}(nm)$.
Точнее за «(*число не нулей в* `a`)$\cdot$(*число не нулей в* `b`)».

• Карацуба

Пусть $N = 2^k$. Если нет, дополним оба массива нулями (нулевые старшие коэффициенты).
Делим многочлены на две части: $A = A_0 + x^{N/2} A_1$, $B = B_0 + x^{N/2} B_1$

$$A \cdot B = A_0 B_0 + x^N A_1 B_1 + x^{N/2}(A_0 B_1 + A_1 B_0) = C_0 + x^N C_1 + x^{N/2}((A_0{+}B_0)(A_1{+}B_1) - C_0 - C_1)$$

Умножение многочленов длины N – сложение, вычитание и 3 умножения многочленов длины $\frac{N}{2}$.

$$T(n) = 3T(\tfrac{n}{2}) + n = \boxed{\Theta(n^{\log 3})}$$

Такой способ умножения работает над произвольным кольцом.

• Оптимальное над $\mathbb{F}_2$

У нас уже есть Карацуба и битовое сжатие. Соединим. Внутри Карацубы реализуем сложение, вычитание и разделение многочлена на две части за $\lceil \frac{n}{w} \rceil$. Казалось бы мы ускорили всё в w раз, но нет, время работы равно числу листьев рекурсии.

$$T(n) = 3T(\tfrac{n}{2}) + \lceil \tfrac{n}{w} \rceil = \Theta(n^{\log 3})$$

Асимптотическая оптимизация: в рекурсии при $N \leqslant w$ будем умножать за $\mathcal{O}(n)$. Улучшили $w^{\log 3}$ до $w \Rightarrow$ новое время работы в $w^{(\log 3)-1}$ раз меньше.

• **Над $\mathbb{Z}$, над $\mathbb{R}$, над $\mathbb{C}$**

Фурье за $\mathcal{O}(n \log n)$. Смотри главу про Фурье (FFT).

• **Над конечным полем**

Все конечные поля изоморфны $\Rightarrow$ умножить над $\mathbb{F}_p \Leftrightarrow$ умножить над $\mathbb{Z}/p\mathbb{Z}$.
Умножим в $\mathbb{Z}$ (Карацуба или FFT), затем возьмём по модулю.
Для некоторых p, например $p = 3 \cdot 2^{18} + 1$, можно напрямую применить «Фурье по модулю».

Лекция #2: Деление многочленов

2-я пара, весна 2025

2.1. Быстрое деление многочленов

Цель – научиться делить многочлены за $\mathcal{O}(n \log n)$.
Очень хочется считать частное многочленов $A(x)/B(x)$, как $A(x)B^{-1}(x)$. К сожалению, у многочленов нет обратных. Зато обратные есть у рядов, научимся сперва искать их.

• Обращение ряда

Задача. Дан ряд $A \in [[\mathbb{R}]]$, $a_0 \neq 0$. Найти ряд B: $A(x)B(x) = 1$.
Первые n коэффициентов B можно найти за $\mathcal{O}(n^2)$:
$b_0 = 1/a_0$
$b_1 = -(a_1 b_0)/a_0$
$b_2 = -(a_2 b_0 + a_1 b_1)/a_0$
...
А можно за $\mathcal{O}(n \log n)$.
Обозначим $B_k(x) = b_0 + b_1 x + \cdots + b_{k-1} x^{k-1}$. Заметим, что $\forall k$ $A(x)B_k(x) = 1 + x^k C_k(x)$.
$B_1 = b_0 = 1/a_0$. Научимся делать переход $B_k \to B_{2k}$ за $\mathcal{O}(k \log k)$.
$B_{2k} = B_k + x^k Z \Rightarrow A \cdot B_{2k} = 1 + x^k C_k + x^k A \cdot Z = 1 + x^k (C_k - A \cdot Z)$.
Выберем $Z = B_k \cdot C_k \Rightarrow C_k - A \cdot Z = C_k - C_k(A \cdot B_k) = C_k - C_k(1 + x^k C_k) = -x^k C_k^2$.
Итого $B_{2k} = B_k + B_k(x^k C_k) = B_k + B_k(1 - A \cdot B_k) \Rightarrow \boxed{B_{2k} = B_k(2 - A \cdot B_k)}$

Два умножения = $\mathcal{O}(k \log k)$. Общее время работы $n \log n + \frac{n}{2} \log \frac{n}{2} + \frac{n}{4} \log \frac{n}{4} + \cdots = \mathcal{O}(n \log n)$.
Конечно, мы обрежем B_{2k}, оставив лишь $2k$ первых членов.

• Деление многочленов

A^R – reverse многочлена. $a_0 + a_1 x + \cdots + a_n x^n \to a_n + a_{n-1} x + \cdots + a_0 x^n$.
Умножение: $A^R B^R = (AB)^R$ (доказательство: в c_{ij} += $a_i b_j$ поменяли индексы на $n-i$ и $m-j$).
Новое определение деления: по A, B хотим C: $A^R \equiv (BC)^R \bmod x^n$.
Здесь n – число коэффициентов у A, у B ровно столько же.

Обращение ряда нам даёт умение по многочлену Z: $z_0 \neq 0$ строить Z^{-1}: $Z \cdot Z^{-1} \equiv 1 \bmod x^n$.

$$C^R = (B^R)^{-1} A^R$$

Время работы: обращение ряда + умножение = $\mathcal{O}(n \log n)$.
Над кольцом делить странно, а вот над произвольным полем Фурье может не работать, тогда деление работает за $\mathcal{O}(\texttt{mul}(n) + \texttt{mul}(\frac{n}{2}) + \dots) = \mathcal{O}(\texttt{mul}(n))$ для $\texttt{mul}(n) = \Omega(n)$.

2.2. (*) Быстрое деление чисел

Для нахождение частного чисел, достаточно научиться с большой точностью считать обратно.
Рассмотрим метод Ньютона поиска корня функции $f(x)$:
$x_0 =$ `достаточно точное приближение корня`
$x_{i+1} = x_i - f(x_i)/f'(x_i)$
Решим с помощью него уравнение $f(x) = x^{-1} - a = 0$.
$x_0 =$ `обратное к старшей цифре` a
$x_{i+1} = x_i - (\frac{1}{x_i} - a)/(-\frac{1}{x_i^2}) = x_i + (x_i - a \cdot x_i^2) = x_i(2 - ax_i)$.
Любопытно, что очень похожую формулу мы видели при обращении формального ряда...
Утверждение: каждый шаг метода Ньютона удваивает число точных знаков x.
Итого, имея x_i с k точными знаками, мы научились за $\mathcal{O}(k \log k)$ получать x_{i+1} с $2k$ точными знаками. Суммарное время получения n точных знаков $\mathcal{O}(n \log n)$.

2.3. (*) Быстрое извлечение корня для чисел

Продолжаем пользоваться методом Ньютона.

$$x_{i+1} = \frac{1}{2}(x_i + \frac{a}{x_i})$$

Если у x_i k_i верных знаков, то $k_{i+1} = k_i + \Theta(1)$, а
$x_i \to x_{i+1}$ вычисляется одним делением многочленов длины k_i за $\mathcal{O}(k_i \log k_i)$.

2.4. (*) Обоснование метода Ньютона

Цель: доказать, что каждый шаг удваивает число точных знаков x.

Сдалем замену переменных, чтобы было верно $f(0) = 0 \Rightarrow$ корень, который мы ищем, – 0.
Сейчас находимся в точке x_i. По Тейлору $f(0) = f(x_i) - x_i f'(x_i) + \frac{1}{2}x_i^2 f''(\alpha)$ $(\alpha \in [0..x_i])$.
Получаем $\frac{f(x_i)}{f'(x_i)} = x_i + \frac{1}{2}x_i^2 \frac{f''(\alpha)}{f'(x_i)}$. Передаём Ньютону $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - x_i - \frac{1}{2}x_i^2 \frac{f''(\alpha)}{f'(x_i)}$.
Величина $\frac{f''(\alpha)}{f'(x_i)}$ ограничена сверху константой C.
Получаем, что если $x_i \leqslant 2^{-n}$, то $x_{i+1} \leqslant 2^{-2n+\log C}$.
То есть, число верных знаков почти удваивается.

2.5. Линейные рекуррентные соотношения

Задача. Дана последовательность $f_0, f_1, \ldots, f_{k-1}, \forall n \geqslant k\ f_n = f_{n-1}a_1 + \cdots + f_{n-k}a_k$, найти f_n.

Вычисления «в лоб» можно произвести за $\mathcal{O}(nk)$.

2.5.1. Через матрицу в степени

$$A = \begin{bmatrix} a_1 & a_2 & \ldots & a_k \\ 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix},\ \forall i\ A \cdot \begin{bmatrix} f_{i-1} \\ f_{i-2} \\ \ldots \\ f_{i-k} \end{bmatrix} = \begin{bmatrix} f_i \\ f_{i-1} \\ \ldots \\ f_{i-k+1} \end{bmatrix} \Rightarrow\ A^n \cdot \begin{bmatrix} f_{k-1} \\ f_{k-2} \\ \ldots \\ f_0 \end{bmatrix} = \begin{bmatrix} f_{n+k-1} \\ f_{n+k-2} \\ \ldots \\ f_n \end{bmatrix}$$

Умножать матрицы умножаем за $\mathcal{O}(k^3) \Rightarrow$ общее время работы $\mathcal{O}(k^3 \log n)$.

2.5.2. Через умножение многочленов

На самом деле можно возводить в степень не матрицу, а многочлен по модулю...
Умножение матриц за $\mathcal{O}(k^3)$ заменяется на умножение многочленов по модулю за $\mathcal{O}(k \log k)$.

Будем выражать f_n через $f_i \colon i < n$. В каждый момент времени $f_n = \sum_j f_j b_j$.

Изначально $f_n = f_n \cdot 1$. Пока $\exists j \geqslant k \colon b_j \neq 0$, меняем $f_j b_j$ на $\left(\sum\limits_{i=1}^{k} f_{j-i}a_i\right) \cdot b_j$. (*)

Посмотрим, как меняются коэффициенты b_j. Пусть $B(x) = \sum b_j x^j$, $A(x) = x^k - \sum\limits_{i=1}^{k} x^{k-i}a_i$.

Тогда (*) – переход от $B(x)$ к $B(x) - A(x)x^{j-k}b_j$.
Изначально $B(x) = x^n \Rightarrow$ наш алгоритм – вычисление $x^n \bmod A(x)$.
Возведение многочлена x в степень n по модулю $A(x)$ – $\mathcal{O}(\log n)$ умножений и взятий по модулю.

Итого: $\mathcal{O}(k \log k \log n)$.

Лекция #2: Применение умножения многочленов

2-я пара, весна 2025

2.6. Факторизация целых чисел

• **Вычисление** $n! \bmod m$

Возьмём $k = \lfloor\sqrt{n}\rfloor$, рассмотрим $P(x) = x(x+k)(x+2k)...(x+k(k-1))$.
$P(1)P(2)P(3)...P(k) = (k^2)!$, чтобы дополнить до $n!$, сделаем руками $\mathcal{O}(k)$ умножений.
Посчитать $P(x)$ мы можем методом разделяй и властвуй за $O(k \log^2 k)$.

• **FFT над** $\mathbb{Z}/m\mathbb{Z}$

Умножим в $\mathbb{Z}$ (то же, что $\mathbb{R}$, что $\mathbb{C}$), в конце возьмём по модулю m.
Если не хватает точности обычного вещественного типа ($m = 10^9 \Rightarrow$ `long double` уже слишком мал), то представим многочлен с коэффициентами до m, как сумму двух многочленов с коэффициентами до $k = \lceil m^{1/2} \rceil$: $P(x) = P_1(x) + k \cdot P_2(x), Q(x) = Q_1(x) + k \cdot Q_2(x)$.
Сделаем 3 умножения, как в Карацубе:

$$P(x)Q(x) = P_1Q_1 + k^2P_2Q_2 + k((P_1{+}Q_1) \cdot (P_2{+}Q_2) - P_1Q_1 - P_2Q_2)$$

• **Факторизация**

Найдём $\min d$: $gcd(d!, n) \neq 1$. Тогда d – минимальный делитель n. Можно искать бинпоиском, можно «двоичными подъёмами», тогда время $= \mathcal{O}(calc(n^{1/2}!)) = \mathcal{O}(n^{1/4} \log^2 n)$.

2.7. CRC-32

Один из вариантов хеширования последовательности бит $a_0, a_1, \ldots, a_{n-1}$ – взять остаток от деления многочлена $A(x) \cdot x^k$ на $G(x)$, где $G(x)$ – специальный многочлен, а $k = \deg G + 1$.

В *CRC-32-IEEE-802.3* (в v.42, mpeg-2, png, cksum) $k = 32$, $G(x) =$ 0xEDB88320 (32 бита).
Если размер машинного слова $\geqslant k$, CRC вычисляется за $\mathcal{O}(n)$, в общем случаем за $\mathcal{O}(n \cdot \lceil \frac{k}{w} \rceil)$.

```
1 for i = n-1..k:
2     if a[i] != 0:
3         a[i..i-k+1] ^= G
```

Упражнение 2.7.1. `CRC(A ^ B)` = `CRC(A) ^ CRC(B)`, `CRC(concat(A, 1))` = (`CRC(A)` $\cdot 2 + 1$) $\bmod G$

2.8. Кодирование бит с одной ошибкой

Контекст. По каналу хотим передать n бит.
В канале может произойти не более 1 ошибки вида «замена бита».

Детектирование ошибки. Передадим $a_1, a_2, \ldots, a_n$ и $b =$ `XOR`$(a_1, a_2, \ldots, a_n)$.
Если b' равно `XOR`$(a'_1, a'_2, \ldots, a'_n)$, ошибки при передачи не было.

Исправление ошибки.
Удвоение бит не работает – и из 0, и из 1 в результате одной ошибки может получиться 01.
Работает утроение бит ($3n$) или «удвоение бит и дополнительно передать `XOR`» ($2n{+}1$).
Раскодирование для $2n{+}1$: $00 \to 0,\ 11 \to 1,\ 01/10 \Rightarrow$ подгоняем, чтобы сошёлся `XOR`.

Исправление ошибки за $\lceil \log_2 n \rceil + 1$ дополнительных бит.
Передадим два раза $\texttt{XOR}(a_1, a_2, \ldots, a_n)$. Теперь мы знаем, есть ли ошибка среди $a_1, a_2, \ldots, a_n$.
Если ошибка есть, её нужно исправить. $\forall b \in [0, \lceil \log_2 n \rceil)$ передадим $\texttt{XOR}_{i:\, bit(i,b)=0}(a_i)$.
В итоге мы знаем все $\lceil \log_2 n \rceil$ бит позиции ошибки.

2.9. Коды Рида-Соломона

Задача. Кодируем n элементов конечного поля $\mathbb{F}_q$.
Канал допускает k ошибок. Хотим научиться исправлять ошибки после передачи.

Замечание 2.9.1. Конченые поля имеют размер p^k ($p \in Prime, k \in \mathbb{N}$). Поле размера p – $\mathbb{Z}/p\mathbb{Z}$.
Поле размера p^k – остатки по модулю неприводимого многочлена над $\mathbb{F}_p$ степени k.
Для кодирования битовых строк удобно использовать $q = 2^k$ при $k \in \{32, 64, 4096\}$.
Пример $\mathbb{F}_{2^{32}}$: остатки по модулю $x^{32}+x^{22}+x^2+x+1$. Один из алгоритмов поиска неприводимого степени n: ткнуть в случайный, попробовать факторизовать (Бэрликэмп за n^3).

Lm 2.9.2. Если обозначить расстояние Хэмминга как $D(s,t)$, а $f(s)$ – код строки s, канал передачи допускает k ошибок, то исправить ошибки можно iff $\forall s \neq t \;\; D(f(s), f(t)) \geqslant 2k+1$.

Доказательство. С учётом ошибок строка s может перейти в любую точку шара радиуса k с центром в $f(s)$. Если такие шары пересекаются, $\forall$ точку пересечения не раскодировать. ■

Код Рида-Соломона. Данные $a_0, \ldots a_{n-1}$ задают многочлен $A(x) = \sum a_i x^i$. Передадим значения многочлена в произвольных $n+2k+1$ различных точках (здесь мы требуем $p \geqslant n$).

Теорема 2.9.3. Корректность кодов Рида-Соломона.

Доказательство. $A(x) \neq B(x) \Rightarrow (A-B)(x)$ имеет не более n корней $\Rightarrow A(x)$ и $B(x)$ имеют не более n общих значений $\Rightarrow$ хотя бы $2k+1$ различных $\Rightarrow D(f(A), f(B)) \geqslant 2k+1$. ■

Выбор точек и q: хочется применить FFT. На практике можно отправлять данные такими порциями, что $n+2k+1 = 2^b$. Нужно q вида $a \cdot 2^b + 1$ и x: $ord(x) = 2^b$, точка $w_i = x^i, \;\; i \in [0, 2^b)$.

• **Декодирование**

Мы доказали возможность однозначного декодирования, осталось предъявить алгоритм.
Имели $A(x)$, $\deg A = n-1$, передали $A(w_0), A(w_1), \ldots A(w_{n+2k})$. Получили на выходе данные с ошибками $A'(w_0), A'(w_1), \ldots A'(w_{n+2k})$. Посчитали интерполяционный многочлен $A'(x)$.

Утверждение 2.9.4. Если у $A'(x)$ старшие $2k+1$ коэффициентов нули, ошибок не было.

Итого, если ошибок нет, декодирование при желании можно сделать за $\mathcal{O}(n \log n)$ через FFT.

Обозначим позиции ошибок $e_1, e_2, \ldots, e_k$.
Может быть, ошибок меньше. Главное, что в позициях кроме e_i ошибок точно нет.
Многочлен ошибок $E(x) = (x - w_{e_1})(x - w_{e_2}) \ldots (x - w_{e_k})$, $B(x) = A(x)E(x)$, $B'(x) = A'(x)E(x)$.
$\forall i \notin \{e_j\}\; A(w_i) = A'(w_i) \Rightarrow \forall i \; B(w_i) = B'(w_i)$. Запишем это, как СЛАУ $\forall i \; B(w_i) = A'(w_i)E(w_i)$, где неизвестные – коэффициенты многочленов B ($n+k+1$ штук) и E (k штук).

Теорема 2.9.5. Для записанной СЛАУ $\exists!$ решение.

Следствие 2.9.6. Декодирование: решим СЛАУ, найдём $A(x) = \frac{B(x)}{E(x)}$.

Асимптотика декодирования: Гаусс $\mathcal{O}((n+k)^3)$. Бэрликэмп-Мэсси $\mathcal{O}((n+k)^2)$.
Ссылка на более крутые алгоритмы декодирования в разд. 2.11.

2.10. Применения FFT в комбинаторике

• Возведение в степень

2^n бинарным возведением в степень вычисляется за $T(n) = T(\frac{n}{2}) + n \log n = \mathcal{O}(n \log n)$.

• Умножение многочленов от нескольких переменных

Посчитать $C(x, y) = A(x, y) \cdot B(x, y)$, пусть $\deg_x \leqslant n, \deg_y \leqslant m$, тогда возьмём $y = x^{2n+1}$ и вычислим $C'(x) = A'(x) \cdot B'(x)$, мономы вида $ax^{(2n+1)i+j}$, $j \leqslant 2n$ заменим на $ax^j y^i$. $\mathcal{O}(nm \log nm)$.

2.10.1. Покраска вершин графа в k цветов

Предподсчитаем за $\mathcal{O}(2^n)$ все независимые множества I (те, что можно покрасить в один цвет).

Рассмотрим любую корректную покраску в k цветов $I_1, I_2, \dots I_k$: $\sqcup I_j = V$,
заметим $\sum I_j = 2^n - 1, \sum |I_j| = n$.
Рассмотрим $P(x, y) = \sum_I x^I y^{|I|}$, внимательно посмотрим на $P^k(x, y)$:

Теорема 2.10.1. Коэффициент в $P^k(x, y)$ монома $x^{2^n-1} y^n$ – это число покрасок в k цветов.

Lm 2.10.2. $A \cap B = \varnothing \Leftrightarrow |A| + |B| = |A \cup B|$

Lm 2.10.3. $A \cap B \neq \varnothing \Leftrightarrow A + B > A \cup B$
(сложение/сравнение множеств – операции с битовыми масками)

Алгоритм – возведение многочлена степени $2^n n$ в степень k.
Одно умножение работает за $\mathcal{O}(2^n n^2)$, возведение в степень за $\mathcal{O}(2^n n^2 \log k)$.

2.10.2. Счастливые билеты

Задача. Массив из $2n$ цифр из множества $d_1, d_2, \dots, d_k$ называется счастливым, если $\sum$ первых n цифр совпадает с $\sum$ последних n. Найти число счастливых массивов из $2n$ цифр.

Рассмотрим $P(x) = (x^{d_1} + x^{d_2} + \dots + x^{d_k})^n$. Ответ – сумма квадратов коэффициентов P.
Алгоритм = возведение в степень. Время возведения в степень – $\mathcal{O}(\log n)$ умножений.
Точнее $\mathcal{O}(mul(N) + mul(\frac{N}{2}) + mul(\frac{N}{4}) + \dots) = \mathcal{O}(N \log N)$, где $\deg P = N = n \cdot \max d_i$.

2.10.3. 3-SUM

Задача. Даны n чисел $a_i \in \mathbb{Z} \cap [S]$, найти i, j, k: $a_i + a_j + a_k = S$.
Решение #1. Сортировка, далее $\forall i$ два указателя для j, k. $\mathcal{O}(n^2)$.
Решение #2. Возьмём $P(x) = \sum_i x^{a_i}$, рассмотрим P^3, возьмём коэффициент при x^S. $\mathcal{O}(S \log S)$.

2.10.4. Применение к задаче о рюкзаке

Простая задача. Subsetsum. Даны n целых $a_i > 0$, выбрать подмножество: $\sum_j a_j = S$.
Посчитаем $P(x) = \prod_i (1 + x^{a_i})$, возьмём коэффициент при x^S. n умножений $\Rightarrow \mathcal{O}(nS \log S)$.

Алгоритм 2.10.4. Сложная задача. *То же, но выбрать подмножество размера ровно k.*

Посчитаем $P(x, y) = \prod_i (1 + yx^{a_i})$, возьмём коэффициент при $x^S y^k$.
Будем вычислять разделяйкой: $T(n) = 2T(\frac{n}{2}) + nS \log nS$ (nS – степень многочлена).
Получаем $\mathcal{O}(nS \log nS \log n)$, что лучше базовой динамики за $\mathcal{O}(n^2 S)$ (`dp[i,size,sum]`).

2.10.5. (*) Сверхбыстрый рюкзак за $\tilde{O}(\sqrt{n}S)$

Решаем subsetsum. Пусть $A = \{a_i\}$. Если мы разобьём $A = A_1 \sqcup \cdots \sqcup A_k$ и для каждого A_i насчитаем массив f_{ij} = можем ли мы набрать вес j, используя предметы из A_i, то останется только за $\mathcal{O}(kS \log S)$ перемножить $F_1(x) \cdot \ldots \cdot \cdots \cdot F_k(x)$, где $F_i(x) = \sum f_{ij} x^j$.

Возьмём $k \approx \sqrt{n}$, $A_i = \{x \in A \colon x \bmod k = i\}$. $x \in A_i \Rightarrow x = ky + i \Rightarrow$
рассмотрим $B_i = \{y \colon ky + i \in A_i\}$ и для B_i воспользуемся 2.10.4, который насчитает $\sum f_{ijt} x^j y^t$, где f_{ijt} = число способов набрать сумму ровно j, используя ровно t предметов из B_i, при этом $jk + it \leqslant S \Rightarrow j \leqslant \lfloor \frac{S}{k} \rfloor \Rightarrow$ 2.10.4 отработает за $\mathcal{O}(\frac{S}{k} n_i \log n_i \log nS)$, где $n_i = |A_i|$.

Итого: решили subsetsum за $\mathcal{O}(kS \log S) + \sum n_i \mathcal{O}(\frac{S}{k} n_i \log n_i \log nS) =$
$\mathcal{O}(kS \log S) + \mathcal{O}(\frac{S}{k} n \log n \log S) \Rightarrow$ оптимальное $k = \sqrt{n \log n}$, subsetsum за $\mathcal{O}(\sqrt{n \log n} \cdot S \log S)$.
$\tilde{O} f = \mathcal{O}(f \cdot poly(\log))$.

2.10.6. (*) Сверхбыстрый рюкзак за $\tilde{O}(n + S)$

Будем решать задачу $f(A)$**:** определить $\forall s \in [0,S]$, можно ли набрать вес s.

Если есть ответы $f(A)$ и $f(B)$, то `fft` за $\mathcal{O}(S \log S)$ даёт ответ для $A+B$. Назовём это свёрткой. Если мы разделим множество предметов A на $A = A_1 \sqcup A_2 \sqcup \cdots \sqcup A_k$, мы можем для каждой части A_i решить задачу и свёртками за $\mathcal{O}(kS \log S)$ получить ответ для A.

Хорошее разделение: $m = \lceil \log n \rceil$, $A_i = A \cap (\frac{S}{2^i}, \frac{S}{2^{i-1}}], i \in [1, m]$, $A_{m+1} = A \cap [1, \frac{S}{2^m}]$.
Для A_{m+1} делаем разделяйку с `fft` даёт $T(n) = 2T(\frac{n}{2}) + \texttt{fft}(n\frac{S}{2^m}) = \mathcal{O}(S \cdot n \log n \cdot \log)$.
Для A_i рассмотрим множество-ответ X_i, заметим $|X_i| < 2^i$. Обозначим $k = 2^i$.
Поделим A_i случайным образом на k множеств: $A_i = \sqcup A_{ij}$, $\mathsf{E}(\max_j(X_i \cap A_{ij})) = \mathcal{O}(\log k) = \varepsilon$, чтобы решить задачу для A_{ij} разделим его на ε^2 случайных множеств A_{ijt}.

$Pr[\forall t \ |A_{ijt} \cap X_i| \leqslant 1] \geqslant \frac{1}{2} \Rightarrow$ ответ для A_{ijt} тривиален: мы можем взять $\leqslant 1$ предмета $\Rightarrow$ можем набрать только суммы $s \in A_{ijt}$. Ответ для $A_{ij} = \varepsilon^2$ свёрток, при этому в ответе нам нужны суммы не до S, а до $\frac{S}{2^i}\varepsilon$, так как $\max A_{ij} \leqslant \frac{S}{2^i}$ и $|A_{ij} \cap X_i| \leqslant \varepsilon$. Осталось свернуть ответы для A_{ij} в ответ для A_i – разделяйка длины 2^i, где в листьях `fft` от длины $\frac{S}{2^i}\varepsilon \Rightarrow$ время на свёртки для A_i: $2^i \log(2^i) M \log M$, где $M = \frac{S}{2^i}\varepsilon \Rightarrow \mathcal{O}(S \log 2^i \log M\varepsilon) = \mathcal{O}(S \log^3)$. Итак $\forall i \Rightarrow \mathcal{O}(S \log^4) = \tilde{O}(S)$.

2.11. Литература

Про FFT.
[Shuhong,Gao'2002]. Декодирование Рида-Соломона через расширенного Евклида.

Про рюкзаки (и отчасти FFT).
[Koiliaris,Xu'2017]. Subset Sum in $\tilde{\mathcal{O}}(\sqrt{n}S)$.
[Birmingam'2017]. Subset Sum in $\tilde{\mathcal{O}}(n + S)$.
[Jin,Wu'2018]. Subset Sum in $\mathcal{O}((n+S)\log^2)$. Улучшили log-факторы предыдущего решения.
[Pissinger'1999]. Практически эффективный knapsack за $\mathcal{O}(n \cdot \max a_i)$.
[Bringmann'2021]. Тут описывают, когда рюкзак решается за $\tilde{\mathcal{O}}(n)$ и дают нижние оценки.
[Becker'2011]. Рюкзак за $\tilde{\mathcal{O}}(2^{0.291n})$.

Лекция #3: Автоматы

3-я пара, весна 2025

3.1. Определения, детерминизация

Def 3.1.1. *Детерминированный автомат* – $\langle V, s, T, \Sigma, E\rangle, s \in V, T \subseteq V, D \subseteq V \times \Sigma,\ E\colon D \to V$. *Обозначим* $|V| = n$, $|E| = m$.

Def 3.1.2. *Детерминированный автомат называется полным, если* $D = V \times \Sigma$.

Замечание 3.1.3. Чтобы сделать автомат полным, добавим фиктивную вершину «тупик», все ∄ рёбра направим в «тупик», замкнём «тупик»: по всем символам из него торчат петли.

Def 3.1.4. *Недетерминированный автомат* – $\langle V, s, T, \Sigma, E\rangle,\ s \in V,\ T \subseteq V,\ E \subseteq (V \times \Sigma) \times V$

Def 3.1.5. *Автомат принимает строку* w, *если* $\exists s = v_0, v_1, \dots v_{|w|}\colon \forall i\ (v_i, s_i, v_{i+1}) \in E$.

Замечание 3.1.6. Принимает ли детерминированный автомат строку s, мы проверяем за $\mathcal{O}(|s|)$.

Алгоритм 3.1.7. *Принимает ли недетерминированный автомат строку* s*?*
После i *символов поддерживаем множество вершин «где мы можем сейчас находиться?».*
Переход $i \to i{+}1$ *за* $\mathcal{O}(m) \Rightarrow \mathcal{O}(m|s|)$.

• Детерминизация

Принимая строку s, недетерминированный автомат в момент времени t находится в одной из вершин множества A_t. «Множества вершин» – состояния детерминированного автомата $\langle V', E'\rangle$.

Можно взять $|V'| = 2^n$, можно оптимальнее – dfs-ом выбрать достижимые множества вершин.

Время детерминизации $\mathcal{O}(|V'| \cdot m)$.

3.2. Эквивалентность

• Простейший алгоритм

Проверяем эквивалентность детерминированных автоматов $\langle V_1, s_1, T_1, E_1\rangle$ и $\langle V_2, s_2, T_2, E_2\rangle$.
Найдём все пары состояний $v_1 \in V_1, v_2 \in V_2\colon v_1 \not\equiv v_2$. Все пары будем помещать в очередь.
База: $(v_1 \in T_1) \neq (v_2 \in T_2) \Rightarrow$ помечаем и помещаем $\langle v_1, v_2\rangle$ в очередь.
Переход: $v_1 \not\equiv v_2 \Rightarrow \forall c, x_1, x_2\colon E(x_1, c) = v_1, E(x_2, c) = v_2 \quad x_1 \not\equiv x_2$.
Реализация: `(v1,v2)= q.pop(); for c: for x1 in from(v1,c): for x2 in from(v2,c): toQueue(x1, x2)`
Каждую пару (v_1, v_2) переберём не более одного раза $\Rightarrow$ каждую пару рёбер $\Rightarrow \mathcal{O}(m^2)$.
Для корректности алгоритма автоматы должны быть полными.

• Через минимизацию

Чтобы проверить эквивалентность $A_1 = \langle V_1, E_1, T_1, s_1\rangle$, $A_2 = \langle V_2, E_2, T_2, s_2\rangle$, запустим минимизацию для $\langle V_1 \cup V_2, E_1 \cup E_2, T_1 \cup T_2\rangle$, и посмотрим попали ли s_1 и s_2 в один класс эквивалентности.

3.3. Минимизация

Задача: построить автомат, минимальный по числу вершин, эквивалентный данному. Перед тем, как рассматривать решения, поймём, как устроен минимальный автомат.

Def 3.3.1. $R_A(w)$ – *правый контекст строки* w. $R_A(w) = \{x \mid A \text{ принимает } wx\}$.

Теорема 3.3.2. Минимальный автомат, эквивалентный A, есть $A_{min} = \langle V, E, s, T\rangle$, где $V = \{R_A(w) \text{ по всем } w\},\ E = \{R_A(w) \xrightarrow{c} R_A(wc)\},\ s = R_A(\varepsilon),\ T = \{\varnothing\}$.

Для недетерминированных есть алгоритм Бржозовского [wiki][pdf]:

$$A_{min} = d(r(d(r(A)))) = drdrA$$

Где d – детерминизация автомата, r – разворот всех рёбер автомата и `swap(S, T)`.

Для детерминированных обычно пользуются алгоритмом Хопкрофта.

3.4. Хопкрофт за $\mathcal{O}(VE)$

```
# дополняем автомат до полного (next[v, char] - или конец ребра, или -1)
fictive = newVertex() # next[fictive, *] = -1, isTerminal[fictive] = 0
for v in [0, vertexN):
    for char:
        if next[v, char] == -1:
            next[v, char] = fictive

# строим обратные рёбра, инициализируем классы
for v in [0, vertexN):
    for char:
        prev[next[v, char], char].add(v)
    type[v] = isTerminal[v] # тип/класс вершины
    A[type[v]].add(v)       # A[type] - множество вершин типа type
typeN = 2 # изначально есть только терминалы и нетерминалы

# основной цикл с очередью
queue q; q.push(0); # любой из классов 0, 1
while !q.isEmpty():
    t = q.pop()
    for char:
        Split(t, char) # самая сложная процедура
```

Функция `Split(t, c)` должна разделить во всех существующих классах разделить вершины по предикату «*ведёт ли ребро по символу* c *в класс* t?» и положить в очередь новые классы.

Её несложно реализовать за $\mathcal{O}(E)$, тогда суммарное время работы алгоритма $\mathcal{O}(VE)$, так как `Split` вызовется не более $V - 2$ раз.

3.5. Хопкрофт за $\mathcal{O}(E \log V)$

Можно реализовать `Split` оптимальнее. Главная идея:
1. реализовать `Split(t)` за $\mathcal{O}$(просматра входящих рёбер в t),
2. при разбиении класса на два добавлять в очередь только меньшую половину.

```
1  def Split(t, char):
2      cc++ # очищаем vertexMark[] за O(1)
3      allTypes = []
4      types = []
5      for v in A[t]:
6          for u in prev[v, char]:
7              if vertexMark[u] != cc:
8                  vertexMark[u] = cc
9              t0 = type[u]
10             allTypes.add(t0)
11             B[t0].add(u) # для каждого типа t0 помним посещённую половину
12             if B[t0].size == 0:           types.add(t0)
13             if B[t0].size == A[t0].size: types.remove(t0)
14
15     for t0 in types: # те классы, которые поделились относительно (t, char)
16         if B[t0].size * 2 > A[t0].size: # если B[t0] - большая половина
17             B[t0].clear()
18             for u in A[t0]: # тратим времени O(B[t0].size), то есть, O(уже потраченного)
19                 if vertexMark[u] != cc:
20                     B[t0].add(u)
21         # теперь B[t0] - точно меньшая половина
22         for u in B[t0]: # перекрашиваем половину B[t0]
23             type[u] = typeN # номер нового класса - автоинкремент
24             A[typeN].add(u)
25             A[t0].add(u)
26         # Старый класс t0 разбит на 2 новых (t0,typeN), кладём в очередь меньшую половину
27         q.add(typeN++)
28
29     for t0 in allTypes: # быстрое обнуление массивов B[]
30         B[t0].clear()
```

Время работы. Блок строк [15-26] работает за $\mathcal{O}$([5-13]). Блок [5-13] – перебор вершин $v \in A[t]$ и входящих в них рёбер. Если вершина $v \in A[t]$ на строке (5), то следующий раз мы положим её в очередь в составе в два раза меньшего класса $\Rightarrow$ переберём её не более $\log V$ раз $\Rightarrow$ каждое входящее рёбро переберём $\log V$ раз $\Rightarrow$ время работы $\mathcal{O}(E \log V)$.

Замечание 3.5.1. Здесь E – количество рёбер в автомате, дополненном до полного $\Rightarrow$ $E = V \cdot |\Sigma|$.

3.6. Изоморфность

Def 3.6.1. *Изоморфизм автоматов: биекция на вершинах такая, что начальная → начальная, терминальные → терминальные, рёбра → рёбра.*

• **Проверка двух автоматов на изоморфизм за** $\mathcal{O}(m_1 + m_2)$

Пишем $dfs(v_1, v_2)$, который параллельно ходит по двум автоматам, запускаем $dfs(start_1, start_2)$.

• **Проверка автомата на эквивалентность минимальному за** $\mathcal{O}((m_1 + m_2))$

Пусть 1-й из двух минимальный. Оставим от 2-го только вершины, из которых достижимы

терминальные. Теперь каждая вершина 2-го лежит в одном из классов эквивалентности = вершин 1-го. Пишем $dfs(v_1, v_2)$, который параллельно ходит по двум автоматам, и для каждой v_2, понимает, в каком классе v_1 она лежит. Если какая-то v_2 должна лежать сразу в двух классах, не эквивалентны. Если в одном из автоматов нет парного ребра, не эквивалентны. Запускаем $dfs(start_1, start_2)$.

3.7. Литература

[hopcroft,motwani,ulman'2001]. Книжка про автоматы. Минимизация в разделе 4.4, стр. 154.

Лекция #4: Суффиксный автомат

4-я пара, весна 2025

4.1. Введение, основные леммы

Будем обозначать «v – суффикс u» как $\boxed{v \subseteq u}$

Def 4.1.1. *Суффиксный автомат строки s, $SA(s)$* – min *по числу вершин детерминированный автомат, принимающий ровно суффиксы строки s, включая пустой.*

Def 4.1.2. $R_s(u)$ – *правый контекст строки u относительно строки s.*

$$R_s(u) = \{x \mid ux \subseteq s\}$$

Пример: $s = abacababa \Rightarrow R_s(ba) = \{cababa, ba, \epsilon\}$
Мы будем рассматривать правые контексты только от подстрок $s \Rightarrow R_s(v) \neq \varnothing$.

Def 4.1.3. $V_A = \{u \mid R_s(u) = A\}$ – *все строки с правым контекстом A.*

Def 4.1.4. $V(w)$ – *вершина автомата, в которой заканчивается строка w ($w \in V_A$).*

Утверждение 4.1.5. $\forall A$ все строки V_A заканчиваются в одной вершине суффавтомата.
Собственно вершины автомата, как и в 3.3.2 – классы V_A.

Следствие 4.1.6. Рёбра между вершинами проводятся однозначно:
$(\exists\ x \in V_A,\ xc \in V_B) \Leftrightarrow$ (между вершинами V_A и V_B есть ребро по символу «c»).

Lm 4.1.7. $R_s(v) \cap R_s(u) \neq \varnothing,\ |v| \leqslant |u| \ \Rightarrow\ v \subseteq u.$

Доказательство. Возьмём $w \in R_s(v) \cap R_s(u)$, строки vw и uw – суффиксы s, отрежем w. ■

Lm 4.1.8. $R_s(v) = R_s(u),\ |v| \leqslant |u| \ \Rightarrow\ v \subseteq u.$

Lm 4.1.9. v – суффикс $u \Rightarrow R_s(u) \subseteq R_s(v)$ (у суффикса правый контекст шире).

Lm 4.1.10. $v \subseteq w \subseteq u,\ R_s(v) = R_s(u) \Rightarrow R_s(v) = R_s(w) = R_s(u)$ (непрерывность отрезания).

Следствие 4.1.11. $\forall A$ класс V_A определяется парой $s_{min} \subseteq s_{max}$: $V_A = \{w \mid s_{min} \subseteq w \subseteq s_{max}\}$.

Def 4.1.12. *Суффиксная ссылка $V(w)$ – вершина $V(z)$: $z \subseteq w,\ R_s(z) \neq R_s(w),\ |z| = \max$.*

`suf[V]` – суффиксная ссылка V_A
`len[V]` $= |s_{max}(V_A)|$

Lm 4.1.13. $|s_{min}(V_A)| =$ `len[suf[A]]` $+ 1$

Замечание 4.1.14. `suf[A]` корректно определена iff `len[A]` $\neq 0$.

Lm 4.1.15. У $SA(s)$ терминальными являются вершины `V(s)`, `suf[V(s)]`, `suf[suf[V(s)]]`, ...

Из 4.1.5 (вершины), 4.1.6 (рёбра), 4.1.15 (терминалы) мы представляем устройство суфавтомата.

Из лемм и 4.1.11 (вершина = отрезок суффиксов) 4.1.12 (суффссылка) мы получили инструменты для построения линейного алгоритма.

4.2. Алгоритм построения за линейное время

Алгоритм будет онлайн наращивать строку s. Начинаем с пустой строки $s = \varepsilon$.
Осталось научиться, дописывая к s символ a, от $SA(s)$ переходить к $SA(sa)$.

Будем в каждый момент времени поддерживать:

(a) `start` – $V(\varepsilon)$ (стартовая вершина)
(b) `last` – $V(s)$ (последняя вершина)
(c) `suf[V]` – для каждой вершины автомата суффссылку
(d) `len[V]` – для каждой вершины автомата максимальную длину строки
(e) `next[A,c]` – рёбра автомата

База: $s = \varepsilon$, `start` = `last` = 1.

Для того, чтобы понять, как меняется автомат, нужно понять, как меняются его вершины – правые контексты. Переход: $s \to sa \Rightarrow R_s(v) = \{z_1, \dots, z_k\} \to R_{sa}(v) = \{z_1 a, \dots, z_k a\}$ `+?` ε.

Пример 4.2.1. $s = abacabx,\ R_s(ab) = \{acabx, x\},\ R_{sa}(ab) = \{acabxa, xa\}$

Пример 4.2.2. $s = abacab,\ R_s(ba) = \{cab\},\ R_{sa}(ba) = \{caba, \varepsilon\}$

Lm 4.2.3. $(\varepsilon \in R_{sa}(v)) \Leftrightarrow (v \subseteq sa)$.

TODO

4.3. Реализация

```
template<const int N> // автомат от строки из N вершин
struct Automaton:
    static const int VN = 2 * N + 1; // число вершин будет 2N, и ещё 1 фиктивная.
    int root, last, n, len[VN], suf[VN];
    map<int, int> to[VN]; // храним рёбра по-простому, можно лучше: массив, хещ-таблица
    Automaton(): // конструктор
        n = 1, root = last = newV(0, 0); // 0 - фиктивная, 1 - корень
    int newV(int _len, int _suf): // создание новой вершины
        len[n] = _len, suf[n] = _suf; // уже знаем длину и суффссылку
        return n++;
    void add(int a): // добавляем один символ a, перестраиваем SA(s) -> SA(s+a)
        int r, q, p = last; // p указывает на старый last
        last = newV(len[last] + 1, 0); // s+a заканчивается в новом last
        while (p && !to[p].count(a)) // пропускаем вершины, из которых нет ребра по a
            to[p][a] = last, p = suf[p]; // создаём им ребро по a
        if (!p) // если мы дошли до фиктивной вершины 0
            suf[last] = root;
        else if (len[q = to[p][a]] == len[p] + 1) // если не нужно раздваиваться
            suf[last] = q;
        else:
            r = newV(len[p] + 1, suf[q]); // r - вершина для части q, суффиксов sa
            suf[last] = suf[q] = r;
            to[r] = to[q]; // r - просто копия q
            while (p && to[p][a] == q) // все суффиксы sa должны заканчиваться в r
                to[p][a] = r, p = suf[p];
Automaton SA;
for (char c : str) SA.add(c);
```

4.4. Линейность размера автомата, линейность времени построения

Теорема 4.4.1. При $n \geqslant 3$ в автомате не более $2n-1$ вершин.
Доказательство. Для $n = 2$ имеем базу «три вершины».
Переход $n \to n+1$: добавится две вершины. ■

Замечание 4.4.2. $2n-1$ достигается на тесте «`abbbbb...`».

Теорема 4.4.3. При $n \geqslant 3$ в автомате не более чем $3n-4$ ребра.
Доказательство. Назовём ребро $p \to q$ коротким, если $len[q] = len[p] + 1$.
Короткие рёбра образуют дерево $\Rightarrow$ их не более $2n-2$.
Длинным рёбрам $e\colon p \xrightarrow{c} q$ сопоставим строки $u+c+w$: u – длиннейший путь в p, w – длиннейший из q. Пути длиннейшие $\Rightarrow u+c+w$ из старта в терминал $\Rightarrow$ суффикс. Пути длиннейшие $\Rightarrow u$ и w состоят только из коротких рёбер $\Rightarrow \forall e$ строки $u+c+w$ различны $\Rightarrow$ их не больше непустых суффиксов $\Rightarrow \leqslant n-1$. Итого рёбер $\leqslant (2n-2) + (n-1) = 3n-3$.
Ещё -1 получаем т.к. число вершин $2n-1$ достигается *только* на тесте `abbbb...`. ■

Замечание 4.4.4. $3n-4$ достигается на тесте «`abbb...bbbc`».

4.5. Решение задач

4.5.1. LZSS за $\mathcal{O}(n)$

• LZSS

Мы можем использовать запись (n, i) «повторить n символов, начиная с i-й позиции» для сжатия данных. Например, «abababcbab» можно теперь записать, как «ab(4,0)c(3,1)».

Задача в том, чтобы выбрать код min длины. Чуть упростим задачу: пусть и один символ, и пара (n, i) записываются одинаковым числом байт, тогда (a) выгодно использовать только пары (n, i), (b) выбирать пару с максимальным n и любым i. Сделаем это суф.автоматом за $\mathcal{O}(n)$.

• Жадность

Построим автомат от всего текста t. $\forall$ вершины v автомата предподсчитаем $i[v]$ позицию начала самого длинного суффикса-терминала, достижимого из v.
Пусть мы уже выписали p символов. Пропускаем строку `t[p:]` через автомат, пока $i[v] < p$.
Пусть мы спустились в итоге n раз, остановились в v: $i[v] < p \Rightarrow$ возвращаем пару $(n, i[v])$.

• Практические нюансы

Исходный текст следует бить на куски, например 10^6 байт, чтобы автомат влезал в кэш.
Код, который мы нашли суф.автоматом, следует записать оптимально:
символ кодируется как $9 = 1 + 8$ бит, а пара (n, i) как $\min(9 \cdot n, 1 + \lceil \log(N-p) \rceil + \lceil \log p \rceil)$ бит, где p – сколько мы к паре (n, i) уже выписали символом.

4.5.2. Общая подстрока k строк

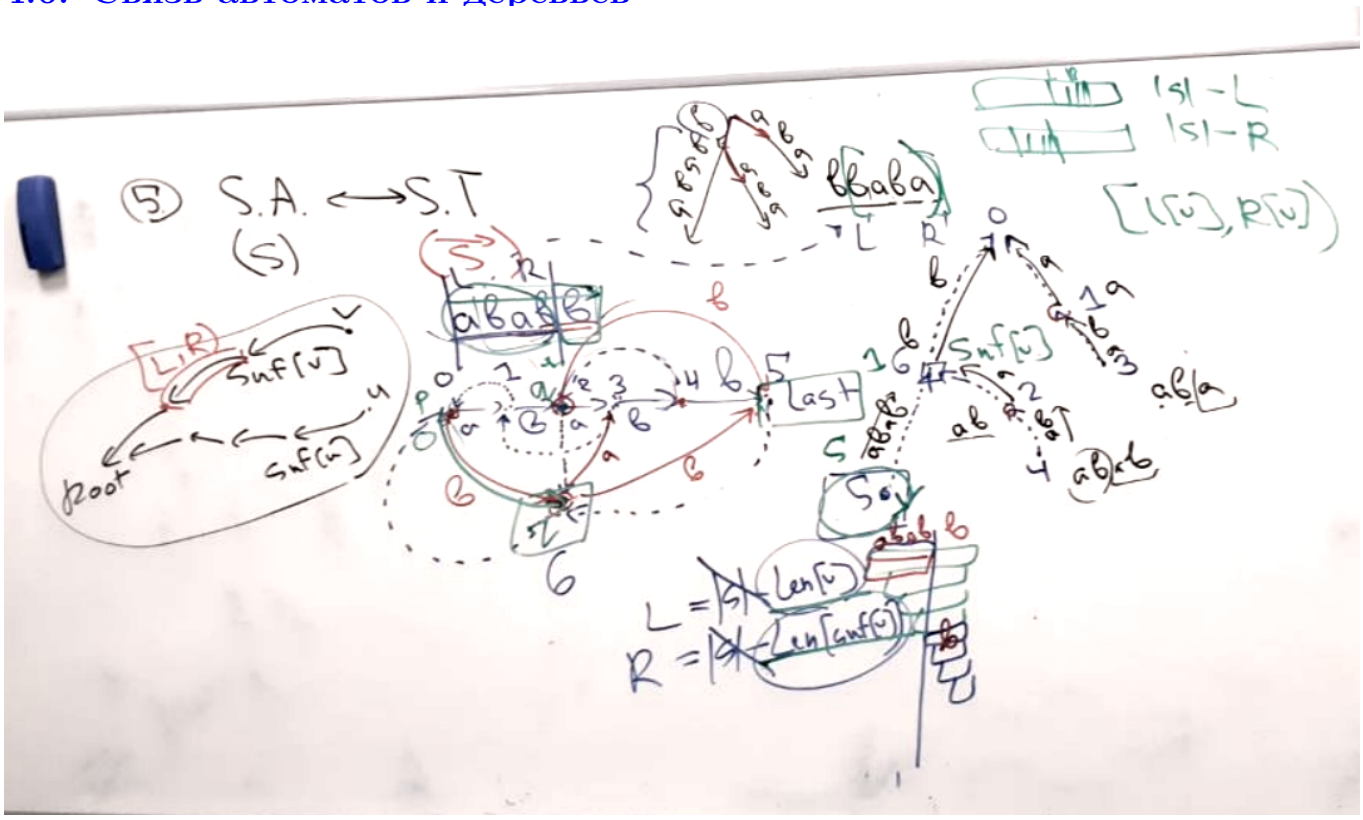
Построим суффавтомат A от минимальной из строк. Далее пропустим все остальные строки через A и для каждой вершины v, которой соответствует класс строк $(len[suf[v]], len[v]]$ будем поддерживать $m[v]$ – длину max общей подстроки, заканчивающейся в v.

Пересчёт $m[v]$. Пропускаем строку s через A:

```
v = root, k = 0
for (c in s)
    while (next[v][c] == 0) // нет ребра
        v = suf[v], k = len[v]
    v = next[v][c], k++
    // пропусктили p - префикс s через A
    // максимальный суффикс p, который есть в A, заканчивается в v
    // длина суффикса p ровно k
```

По всем вершинам v запоминаем $mk[v] = \max k$, а в конце проталкивкаем mk по суффиксным ссылкам. В итоге $m[v] = min(m[v], mk[v])$.

4.6. Связь автоматов и деревьев



4.7. Литература и история

История.

1973 | люди научились за $\mathcal{O}(n)$ растить суффдеревья, первым был алгоритм Вейнера
1983 | люди увидели связь дерева и автомата, $\mathrm{ST}(s).edges = \mathrm{SA}(s^R).suflinks$
1987 | заметили, что в автомате линия рёбер
1997 | придумали, как строить автоматы от строк напрямую, без деревьев
2001 | придумали, как строить автоматы уже от бора строк
2005 | суфф автоматы потихоньку переворачивают мир ICPC

[e-maxx]. Полное изложение суффавтомата.
[itmo]. Изложение суффавтомата без оценок размера и времени работы.
[wiki]. Полное изложение суффавтомата (автор adamant).
[codeforces]. Пост от adamant на тему суффавтоматов.
[cp-algorithms]. Ещё одно описание суффавтоматов.
[SK]. Код для копипаста.
[2009|pdf]. Статья Mohri,Moreno,Weinstein про «суффавтомат от бора» и «music identification».
[2009|pdf]. Preprint статьи выше.

Лекция #5: Линейное программирование

5-я пара 2025

5.1. Применение LP и ILP

Def 5.1.1. *Задача LP. Поиск* x: $Ax \leqslant b,\ x \geqslant 0,\ \langle c, x\rangle \to \max,\ x \in \mathbb{R}^n$

Def 5.1.2. *Задача ILP. Поиск* x: $Ax \leqslant b,\ x \geqslant 0,\ \langle c, x\rangle \to \max,\ \boxed{x \in \mathbb{Z}^n}$

При этом мы помним, как приводить к виду LP несколько похожих задач.

• LP: Кратчайшее расстояние в графе (от s до всех)

x_i — расстояние до вершины i, $x_s = 0$
Для каждого ребра $a \xrightarrow{w} b$ добавляем условие $x_b \leqslant x_a + w$
$\sum_i x_i \to \max$

• LP: Максимальный поток $s \to t$

Обратные рёбра не создаём, x_e — поток по ребру e.
$0 \leqslant x_e \leqslant capacity_e$ (второе добавляем в список неравенств)
Для каждой вершины $v \neq s, t$ добавляем условие $\sum x_{e \in in(v)} = \sum x_{e \in out(v)}$
$\sum_{e \in out(s)} x_e \to \max$

Немного магии: в симплекс-методе можно $0 \leqslant x_e$ бесплатно заменить на $0 \leqslant x_e \leqslant c_e$.

• LP: Поток размера k минимальной стоимости $s \to t$

$\sum_{e \in out(s)} x_e = k$
$\sum_e x_e cost_e \to \min$

• LP: Мультипродуктовый поток

Мы решаем на одной сети одновременно k задач «пустить из s_i в t_i F_i единиц потока». По каждому ребру e течёт одновременно $f_{e_1}, f_{e_2}, \ldots, f_{e_k}$, и должно выполняться общее ограничение $\forall e \sum_i f_{e_i} \leqslant capacity_e$. В отличии от предыдущих проще чем «сведём к LP» задача не решается.

• ILP: Два непересекающихся пути $A \to B, C \to D$

Задача NP-трудна, через поток размера два (склеить A и B, C и D) не делается.
Зато является частным случаем *целочисленного мультипродуктового потока.*

• ILP: Паросочетание в произвольном графе максимального веса

Каждому ребру сопоставляем переменную x_e — взяли ли мы ребро в паросочетание. $x_e \geqslant 0$.
Каждой вершине условие $\sum_{e \in adj(v)} x_e \leqslant 1$.
$\sum_e x_e cost_e \to \max$.

Замечание 5.1.3. На самом деле эту задачу можно решить за полином через LP

• LP: Паросочетание в двудольном графе максимального веса

Та же сеть, что в предыдущей, но благодаря двудольности матрица A задачи LP будет обладать свойством *тотальной унимодулярности*, из чего следует, что симплекс автоматически найдёт целочисленное решение.

• **LP: Вершинное покрытие минимального веса**

$0 \leqslant x_v$ – взяли ли вершину v, для каждого ребра (a, b) имеем $x_a + x_b \geqslant 1$. Цель: $\sum_v x_v w_v \to \min$.
$x_v \in \mathbb{Z} \Rightarrow$ решаем ILP, возвращаем $\{i \mid x_i = 1\}$, получили точное решение.
$x_v \in \mathbb{R} \Rightarrow$ решаем LP, возвращаем $\{i \mid x_i \geqslant \frac{1}{2}\}$, получили 2-приближение.

5.2. Сложность задач LP и ILP

Симплекс метод решает `LP` на большинстве тестов за полином, но в худшем всё же за экспоненту. Есть полиномиальные решения `LP`. Одно из них – *метод эллипсоидов*, им мы займёмся сегодня.

Lm 5.2.1. `ILP` $\in$ `NP-hard`

Доказательство. Сведём SAT. $0 \leqslant x_i \leqslant 1$, для каждого дизъюнкта $\sum x_{i_j} \geqslant 1$. ■

5.3. Нормальные формы задачи, сведения

Есть несколько форм задачи LP, равносильных стандартной форме $Ax \geqslant 0, x \geqslant 0, \langle c, x\rangle \to \max$:

- $Ax = b, x \geqslant 0$ (уравнения уместо неравенств).
- $\langle c, x\rangle \to \min$ (минимизация вместо максимизации).
- $Ax \leqslant b, \langle c, x\rangle \to \max$ (отсутствует $x \geqslant 0$).
- $Ax \leqslant b, x \geqslant 0$ (отсутствует максимизация/минимизация).
- $Ax \geqslant 0$.

Обозначим n – число неизвестных, m – число уравнений.
Будем сводить разные формы задачи друг к другу, следить, как меняются n и m.

$Ax = b \Leftrightarrow Ax \leqslant b \wedge -Ax \leqslant -b$. Свели равенства к неравенствам. $n, m \to n, 2m$.

$Ax \leqslant b \Leftrightarrow Ax + y = b, y \geqslant 0$. Добавили для каждого неравенства переменную $y_i \geqslant 0$, обращающую нер-во в равенство. Свели неравенства к равенствам. $n, m \to n + m, m$.

$\langle c, x\rangle \to \max \Leftrightarrow \langle -c, x\rangle \to \min$. Минимизация и максимизация равносильны.

Если мы умеем решать задачу $Ax \leqslant b$, то $Ax \leqslant b, x \geqslant 0$ – частный случай, а максимизацию можно прикрутить бинпоиском по ответу α, добавив одно неравенство: $\langle c, x\rangle \geqslant \alpha$.

Если у нас отсутствует $x \geqslant 0$, но очень хочется, можно ввести новые переменные:
$x_i = u_i - v_i$, $u_i, v_i \geqslant 0$. $n, m \to 2n, m$. На практике так, не делают, это теоретический трюк.

$Ax \geqslant 0$: **TODO**

5.4. Симплекс метод

5.4.1. Кошерный вид задачи

Пусть исходная задача была в стандартной форме $Ax \leqslant b, x \geqslant 0, \langle c, x\rangle \to \max$. И все $b_i \geqslant 0$.
Введём переменные y_j: $\langle a_j, x\rangle + y_j = b_j, y_j \geqslant 0$. Рассмотрим решение $x_i = 0, y_j = b_j$.

TODO

5.4.2. Поиск начального решения

Пусть $\exists b_i < 0$, возьмём $t = -\min b_i > 0$. Изменим наши неравенства вида $Ax \leqslant b$ на $Ax - t \leqslant b$ (из всех вычтем t). Заметим, что $x_i = 0$ под новые нер-ва подходит. Осталось решить $t \to \min$ при

$t \geqslant 0$: запустим симплекс-метод. Если $\min t = 0$, нашли начальное решение, иначе решений $\nexists$.

5.4.3. Основной шаг оптимизации

У базисных x_i $c_i =, 0$. Пусть у всех не базисных x_i $c_i \leqslant 0 \Rightarrow$ решение оптимально:
$\forall i$ $x_i \geqslant 0, c_i \leqslant 0 \Rightarrow \forall x, c$ $\langle c, x\rangle \leqslant 0 \Rightarrow \langle c, x\rangle = 0 = \max$ – оптимум.

5.5. Геометрия и алгебра симплекс-метода

Система неравенств $Ax \leqslant b$ — пересечение полупространств. Такой объект называется полиэдром. Полиэдр выпукл, максимум любой линейной функции на нём достигается в вершине.

Lm 5.5.1. Для системы $Ax = b, x \geqslant 0, \langle c, x\rangle \to \max$ из m уравнений $\exists$ оптимальное решение, содержащее не более m ненулевых x_i.

Доказательство. Рассмотрим оптимальное решение x^* с максимальным числом нулевых компонент. Пусть число нулей $k \geqslant m+1$. Мы хотим подвигать x^* так, чтобы нули остались нулями, а x^* осталось решением. Решение системы «m уравнений $Ax = b$ и $n-k$ уравнений $x_i = 0$» или пусто, или хотя бы прямая ($dim = n - m - (n-k) = k - m \geqslant 1$). x^* — решение $\Rightarrow \exists$ прямая, пойдём по ней в сторону увеличения $\langle c, x\rangle$, пока не упрёмся в ограничение $x_j = 0$.
Получили решение, у которого $\langle c, x\rangle$ не хуже, а нулей больше. Противоречие. ■

Теорема 5.5.2. Корректность симплекса

Доказательство. Чем занимается симплекс? Перебирает наборы из m столбцов, которые соответствуют ненулевым переменным. Зафиксировав эти m столбцов и занулив оставшиеся переменные, мы однозначно получаем кандидата на оптимальное решение.
Конечность алгоритма: есть всего $\binom{m}{n}$ выборок, важно, чтобы они не повторялись.
Для этого мы используем правило Блэнда (доказательство) — брать min столбец.
Оптимальность решения после остановки симплекса: $\forall i$ $c_i \leqslant 0 \Rightarrow \langle c, x\rangle \leqslant 0 \Rightarrow 0$ – оптимум. ■

• Симплекс перебирает вершины полиэдра

Если исходная задача имела форму $Ax \leqslant b, x \geqslant 0$, то область допустимых решений — полиэдр, а оптимум $\langle c, x\rangle$ достигается в вершине полиэдра. Симплексу мы скармливаем систему $Ax + y = b, x \geqslant 0, y \geqslant 0$. При этом и $y_i = 0$, и $x_i = 0$ означает, что одно из исходных неравенств обратилось в равенство (полупространство $\to$ плоскость). Из $n + m$ переменных x_i, y_i n обратятся в ноль, чем породят n плоскостей, пересечение которых — вершина полиэдра.

• Когда у симплекса есть вероятность застрять?

Если вершина полиэдра – не оптимум, из неё есть ребро, по которому функция увеличивается. Но симплекс может остаться на месте, лишь поменяв множество столбцов. Это значит, что вершина полиэдра есть одновременное пересечение больше чем n плоскостей.

5.6. Литература, полезные ссылки

[CF]. LP: геометрия, двойственность
[cormen]. Доступное для школьников описание симплекса.
[test]. Тест, на котором симплекс работает за экспоненту (Klee–Minty cube).
[efficiency]. Чуть подробнее про время работы симплекса (Hirsch Conjecture and so on).
[bland]. Правило Блэнда.
[max-babenko]. Хороший видео курс про линейное программирование от Максима Бабенко.

5.7. Перебор базисных планов

Понимание симплекса дало нам простейшее решение для LP – перебрать все $\binom{m}{n}$ базисных планов и для каждого пустить Гаусса. Такое решение можно использовать для тестирование. Заметьте, что до этого мы не знали вообще никаких решений задачи LP кроме симплекса.

5.8. Обучение перцептрона

В фундаменте нейронных сетей живёт один нейрон, он же перцептрон. Для решения некоторых задач классификации достаточно сети, состоящей лишь из одного нейрона.

Задача: даны точки $a_i \in \mathbb{R}^n$ и значения $y_i \in \{\pm 1\}$, найти гиперплоскость $b, x_1, \dots x_n$, что $\forall i\ sign(b + a_{i_1}x_1 + a_{i_2}x_2 + \dots + a_{i_n}x_n) = sign(y_i)$

Сразу упростим задачу
1. Добавим $\forall i\ a_{i_0} = 1$, обозначим $x_0 = b$.
2. Для всех $y_i = -1$ заменим a_i на $-a_i$, а y_i на 1. 3. Нормируем все a_i.

Теперь мы просто ищем такой вектор x, что $\forall i\ \langle x, a_i\rangle > 0$.

Решение: начнём с $x_0 = 0$, пока $\exists i_k\ \langle x_k, a_{i_k}\rangle \leqslant 0$, переходим к $x_{k+1} = x_k + a_{i_k}$.

$|a_i| = 1,\ |x_k|^2 = (x_{k-1} + a_{i_{k-1}})^2 = x_{k-1}^2 + 1 + 2\langle x_{k-1}, a_{i_{k-1}}\rangle \leqslant |x_{k-1}|^2 + 1 \leqslant k.$ (1)

x^* – искомый ответ, $|x^*| = 1$, $\langle x_k, x^*\rangle = \sum_j \langle a_{i_j}, x^*\rangle \geqslant k\alpha$, где $\alpha = \min_i \langle a_i, x^*\rangle > 0.$ (2)

$k\alpha \overset{(2)}{\leqslant} \langle x_k, x^*\rangle \leqslant |x_k| \overset{(1)}{\leqslant} \sqrt{k} \Rightarrow \sqrt{k} \leqslant \frac{1}{\alpha} \Rightarrow \boxed{k \leqslant \alpha^{-1/2}}$ ■

Проблема: уже при $n \geqslant 2$ мы можем построить тесты с α близким к нулю.

5.9. Метод эллипсоидов (Хачаян'79)

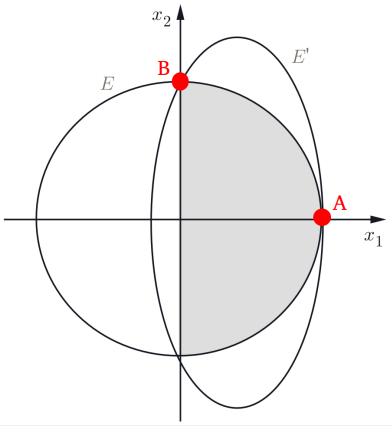
Решаем задачу $P = \{x \mid Ax \geqslant b\}$, найти $x \in P$, ограничение $P \neq \varnothing \Rightarrow Volume(P) > 0$.
Кроме A и b нам должны дать шар $E_0(x_0, r_0) \supseteq P$.

• **Алгоритм**

В каждый момент времени у нас есть эллипсоид $E_k(x_k, R_k)$: $x^T R x \leqslant 1$, содержащий P.
Если $x_k \in P$, счастье. Иначе выберем i_k: $\langle a_{i_k}, x\rangle < b$. $P \subseteq E_k \cap H_k$, где $H_k = \{x \mid \langle a_{i_k}, x\rangle \geqslant b\}$ (в половине эллипсоида по направлению a_{i_k} от центра). Теперь выпишем переход к $(k+1)$-му эллипсоиду в предположении, что E_k – единичная сфера, и $\forall i\ |a_i| = 1$.

$$x_{k+1} = x_k + \frac{1}{n+1} a_{i_k}$$
$$r_{k+1} = \left(\frac{n}{n+1}, \frac{n}{\sqrt{n^2-1}}, \dots, \frac{n}{\sqrt{n^2-1}}\right)$$
$$R_{k+1} = \begin{bmatrix} \frac{(n+1)^2}{n^2} & 0 & 0 \\ 0 & \frac{n^2-1}{n^2} & 0 \\ 0 & 0 & \ddots \end{bmatrix}$$

Где первая координата радиуса указана по направлению a_{i_k}.



• Математика: эллипсоиды

Эллипс: $\frac{x^2}{a^2} + \frac{y^2}{b^2} \leqslant 1$.
Добавим поворот и переход к $\mathbb{R}^n$: $z^t R z \leqslant 1$, где $z \in \mathbb{R}^n, R \in \mathbb{R}^{n \times n}, R \succ 0$.
Случай из $\mathbb{R}^2$ без поворота: $z = (x, y),\ R = \begin{bmatrix} \frac{1}{a} & 0 \\ 0 & \frac{1}{b} \end{bmatrix}$.

• Математика: матрицы

$R \succ 0$ (положительно определена) $\Rightarrow R = B^t D B$ (здравствуй, Жорданова форма), где
B – ортонормированная матрица собственных векторов,
D – диагональная матрица собственных чисел (с точки зрения геометрии $D_{ii} = \frac{1}{r_i^2}$).
Более того $B^t = B^{-1}$ (это нормально для ортонормированных матриц).

Lm 5.9.1. Растяжение системы координат по направлению **z**

(1) Важно помнить, что если $f(x, R) = x^t R x$, то
$f(x + y, R) = f(x, R) + f(y, R),\ f(x, R + \Delta R) = f(x, R) + f(x, \Delta R)$.
(2) Фокус: $\forall z \in \mathbb{R}^n$: $|z| = 1$ матрица $S = zz^t$ обладает свойством $f(z, S) = 1$, $\forall v \perp z\ f(v, S) = 0$.
(1),(2) $\Rightarrow \forall A, |z|{=}1, \alpha{\in}\mathbb{R}, S = A + \alpha z^t z \quad f(z, S) = f(z, A) + \alpha$, $\forall v \perp z\ f(v, S) = f(z, S)$

• Математика: системы координат

Почему мы предполагали, что E_k – единичная сфера?
Это удобно, и мы всегда можем перейти к такой системе координат, где это верно:
$R = B^t D B, x^t R x = 1 \Rightarrow S = B^T D^{-1/2}, x = Sy, y^2 = 1$ (y – точка на единичной сфере).

• Обоснование алгоритма

Радиус $r_1 = \frac{n}{n+1}$ подобран так, чтобы точка A была покрыта: $\frac{1}{n+1} + \frac{n}{n+1} = 1$.
Радиус $r_2 = \frac{n}{\sqrt{n^2-1}}$ подобран так, чтобы точка B была покрыта: $\frac{x_1^2}{r_1^2} + \frac{x_2^2}{r_2^2} = \left(\frac{1}{n+1}\right)^2 \left(\frac{n+1}{n}\right)^2 + \frac{n^2-1}{n^2} = 1$.
Посмотрим на частное объёмов. Объём эллипсоида равен $f(n) r_1 r_2 \ldots r_n$, поэтому $\frac{V_{k+1}}{V_k} = \frac{r_1 r_2 \ldots r_n}{1 \cdot 1 \cdot \ldots 1} =$
$= \frac{n}{n+1} \left(\frac{n}{\sqrt{n^2-1}}\right)^{n-1} = \ldots \leqslant e^{-1/2(n+1)} \Rightarrow$ за $2(n{+}1)$ шагов объём уменьшится хотя бы в e раз.

$\Rightarrow$ количество шагов не более $2(n{+}1) \cdot \ln(Volume(E_0)/Volume(P))$.
Хорошая новость: это полином от n.
Плохая новость: в худшем это $\mathcal{O}(n^4 \log(nU))$.
Пояснение: $Volume(E_0) \leqslant (nU)^{\Theta(n^2)}$, $Volume(P) \geqslant (nU)^{-\Theta(n^3)}$. U – ограничение на $a_{ij}, b_i \in \mathbb{Z}$.

• Время работы алгоритма

Один шаг работает за $mn + n^2$.
m – количество проверок $\langle a_i, x_k \rangle \geqslant b_i$.
n^2 – время пересчёта матрицы система координат.

Требуемая точность вычислений – $n^3 \log U$ цифр.

Итого: $\mathcal{O}^*(n^9)$, где $9 = 2 + 3 + 4$.

• Обобщение

Можно применить не только к $\cap$ полупространств, но и к $\cap\ \forall$ выпуклых множеств P_i.
Нам достаточно уметь проверять $x_k \in P_i$ и искать опорную плоскость к P_i.

Так получается полиномиальное решение для SDP (semidefinite programming) [wiki].

- **Псевдокод**

```
# Наш эллипсоид задаётся уравнением (x − x_0)^t D^{-1} (x − x_0) = 1
# Напомним, что D ≻ 0 ⇒ D = A^T R A, D^{-1} = A^T R^{-1} A, где R - диагональная
x_0 = [0,0,0,...,0]
D[i,i] = 10^20 # ∞
while True: # Предполагаем, что ответ существует
    find i : ∑_j a[i,j] x_0[j] < b[i] # O(nm)
    if i does not exist : return x_0 # Нашли точку, удовлетворяющую всем неравенствам
    z = D*a_i # O(n^2), z - нормаль a_i в другой системе координат
    len = (a_i^t * D * a_i)^{1/2} # O(n^2)
    x_0 += 1/(n+1) z/len # O(n)
    D = n^2/(n^2-1) * (D - 2/(n+1) z*z^t/len^2) # O(n^2), растянули всё в n/√(n^2-1) раз и
                                                # поменяли радиус по направлению z, см. 5.9.1
```

5.10. Литература, полезные ссылки

[MIT'09 | ellipsoid]. Краткое, но полное описание метода эллипсоидов.
[Florida'07 | ellipsoid]. Строгое описание метода эллипсоидов с кучей ссылок по теме.
[Кормен'2013, разделы 29.2 и 35.4]. Примеры задач на LP.

Лекция #6: Суффиксный массив

7 апреля 2025

6.1. Сортировка строк

[wiki]. Integer Sort.

Будем обозначать размер алфавита m, количество строк n, суммарную длину L.

Если все наши строки длины 1, то по сути мы сортируем числа $\mathbb{Z} \cap [0,m)$, и получаем нижнюю оценку $\Omega(\text{IntegerSort})$. Числа сейчас умеют сортировать за $\mathcal{O}(n\sqrt{\log\log n})$.

`QuickSort` отработает за $\mathcal{O}(L \cdot \log n)$.
`RadixSort` для строк одинаковой длины отработает за $\mathcal{O}(L+m)$.
Теперь наша цель научиться и строки произвольной длины сортировать за $\mathcal{O}(L+m)$.

• Сортировка Бором (trie-sort)
Алгоритм: сложим строки в бор + обойдём бор.
Нужно на каком-то этапе сортировать исходящие из вершины рёбра.

Можно через BST: `map` в каждой вершине, например, `SplayMap` даст $\mathcal{O}(L + n\log L)$.

Можно изначально хранить рёбра в `unordered_map`, а после построения бора отсортировать подсчётом пары $\langle$вершина бора, ребро$\rangle$ за $\mathcal{O}(L+m)$.

• Сортировка за $\mathcal{O}(L+m)$ без бора
Пусть наши строки $s[i]$.

(a) Отсортируем $A =$ тройки $\langle j, s[i,j], i\rangle$ подсчётом за $\mathcal{O}(L+m)$.
(b) Используя A, $\forall j$ сделаем «сжатие координат» для символов на j-й позиции.
(c) Теперь можем применить RadixSort к исходным строками:
$\forall j$ пусть есть n_j строк длины $\geqslant j$, отсортируем их подсчётом по j-му символу за $\mathcal{O}(n_j)$.

6.2. SA-IS за $\mathcal{O}(n)$

Все суффиксы s различны, suf_i – суффикс, начинающийся в i-й позиции.

`aabbaabbaccd#`: Пример строки s.
`++--++--+++-` : Для каждого i выпишем «+», если $suf_i < suf_{i+1}$ (next больше).
`*+--*+--*++-` : Отметим отдельно локальные минимумы.

Сделать это можно за $\mathcal{O}(n)$: $\forall i$ найти ближайшее справа j: $s[j] \neq s[i]$, если $s[j] > s[i]$, то «+».

Локальных минимумом $k \leqslant \frac{1}{2}n$.
Разобьём строку на k кусков между локальными минимумами $w_0, w_1, \ldots, w_{k-1}$:

```
a|abba|abba|ccd
*|+--*|+--*|++-
```

Отсортируем w_i# за $\mathcal{O}(n)$, где # больше всех символов, перенумеруем w_i числам $p_{w_i} \in [0,k)$.
Построим от строки $t = p_{w_0}p_{w_1}p_{w_2}\ldots$ суффиксный массив рекурсивным вызовом.

Важно, что не смотря на то, что w_i разной длины, сортировка суффиксов строки t сортирует и суффиксы исходной строки. Если w_i – префикс w_j, то $w_i\# > w_j\# \Rightarrow p_{w_i} > p_{w_j}$.

Пусть последний c символ w_i, тогда в строке s следующий, неравный $>c$, т.к. мы уже прошли локальный минимум, а в суффиксе, начинающимся с w_j первый неравный символ $<c$, т.к. в w_j мы в этой позиции ещё не дошли до локального минимумам.

Пример: `abcbbabcbbbc` → `a|bcbba|bcb|bbc`. `bcb#` > `bcbba#` и `bcb|bbc` > `bcbba|bcb|bbc`.

• Сортировка оставшихся суффиксов

```
aaabbbcccdddcccbbba|aabbbcccbb
+++++++++!----*|+++++---
```

Отсортируем суффиксы A, начинающиеся, от локального максимума (`!`) до локального минимума (`*`) в порядке возрастания.
$\forall$ буквы «`d`», суффикс $suf \in A$, начинающийся с «`d`» устроен как сколько-то букв «`d`», затем меньшая буква $\Rightarrow$ сортировать нужно сперва по количеству «`c`», затем по оставшейся части.

База. Есть порядок суффиксов, начинающихся в локальном минимуме = $\langle$буква, суффикс $t\rangle$.
Перебираем их по возрастанию, раскладываем по первой букве: $\forall$ «`d`»: `list[d]`. *Переход:*

```
for ch in {a,b,c,...}:
    for pos in list[ch]: // pos - позиция начала суффикса
        list[s[pos-1]].push_back(pos-1)
```

Заметим, что по ходу `for pos in list[ch]` вектор `list[ch]` растёт. Мы сперва положим в `list[ch]` суффиксы, у которых ровно одна буква «`ch`», затем две, три и т.д.

Итог. суффиксы от (`!`) до (`*`) отсортированы за $\mathcal{O}(n)$.
Можно аналогичным образом теперь отсортировать суффиксы от (`*`) до (`!`):

TODO

Замечание: w_i – строки вида «сперва возрастает, затем убывает», сортировать их можно также, как мы строим суффмассив (база, переход).

Память: существует реализация, которой кроме $n\lceil\log n\rceil$ бит на хранение ответа нужно $n + K\lceil\log n\rceil$ бит на всю рекурсию, где K – максимальный символ алфавита. Обычно $K \leqslant \frac{n}{2} \Rightarrow n(1 + \frac{1}{2}\log n)$ бит.

6.3. BWT и BWT^{-1} за $\mathcal{O}(n)$

Def 6.3.1. *BWT (Burrows–Wheeler transform) – преобразование строки s в последний столбец таблицы сортированных циклических сдвигов строки s#.*

Пример: $s = acabb \to acabb\# \to \{\#acabb, abb\#ac, acabb\#, b\#acab, bb\#aca, cabb\#a\} \to bc\#baa$.

Построения за $\mathcal{O}(n)$: у строки s# порядки суффиксов и циклических сдвигов совпадают $\Rightarrow$ берём любой алгоритм для суффмассива за $\mathcal{O}(n)$ (Каркайнен-Сандерс, SA-IS) и запускаем.

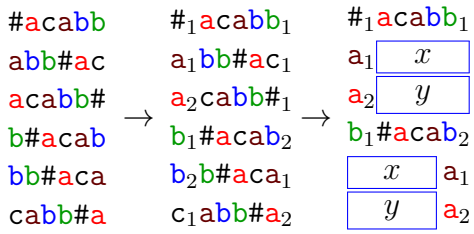
Lm 6.3.2. $\exists$BWT^{-1}: строку s можно однозначно восстановить по BWT(s).

Доказательство. Отсортируем последний столбец, получим первый, припишем за последним, получим все подстроки длины 2, отсортируем, припишем за последним, получим все подстроки длины 3, сортируем, и так далее. В конце s = циклический сдвиг, заканчивающийся на #. ■

BWT^{-1} можно просто получить за $\mathcal{O}(n)$. Давайте различать одинаковые символы строки: $s = \texttt{acabb}$, занумеруем символы в порядке первого столбца, поймём, где они в последнем:

Lm 6.3.3. $\forall$буквы порядок этой буквы в первом и последнем столбцах совпадает.

Доказательство. На примере «a»: возьмём циклические сдвиги, начинающиеся на «a»: $\texttt{a}_1\texttt{bb\#ac}_1 = \texttt{a}_1x$ и $\texttt{a}_2\texttt{cabb\#}_1 = \texttt{a}_2y$, заметим $x < y$ и что, если в последнем столбце $\texttt{a}_1$, в первом как раз начинается x. ■



• **Алгоритм** BWT^{-1} **за** $\mathcal{O}(n)$

Отсортируем подсчётом последний столбец, для каждой буквы в последнем столбце запомним, какая идёт за ней в первом столбце, выпишем буквы в таком порядке. На примере acabb#:

$$\texttt{b}_1 \to \#_1 \to \texttt{a}_2 \to \texttt{c}_1 \to \texttt{a}_1 \to \texttt{b}_2 \to \texttt{b}_1$$

$$\begin{array}{ll} \#_1 \ldots \texttt{b}_1 & \texttt{b}_1 \to \#_1 \\ \texttt{a}_1 \ldots \texttt{c}_1 & \texttt{c}_1 \to \texttt{a}_1 \\ \texttt{a}_2 \ldots \#_1 & \#_1 \to \texttt{a}_2 \\ \texttt{b}_1 \ldots \texttt{b}_2 & \texttt{b}_2 \to \texttt{b}_1 \\ \texttt{b}_2 \ldots \texttt{a}_1 & \texttt{a}_1 \to \texttt{b}_2 \\ \texttt{c}_1 \ldots \texttt{a}_2 & \texttt{a}_2 \to \texttt{c}_1 \end{array}$$

6.4. Поиск подстроки в строке и fm-index

Чтобы искать подстроки в тексте t, построим суффиксный массив $SA(t)$.
Запрос «найти s» будем обрабатывать бинпоиском по $SA(t)$ за $\mathcal{O}(|s| \log |t|)$.

Бинпоиск делает $\log |t|$ сравнений. Сравнение строк внутри бинпоиска можно ускорить с помощью LCP, получится поиск s за $\mathcal{O}(|s| + \log |t|)$:

Обозначим i-й суффикс в порядке суффмассива за p_i. Пусть бинпоиск сейчас на $[l,r]$ и $m = \frac{l+r}{2}$. Мы уже знаем $sl = \text{LCP}(s, p_l)$ и $sr = \text{LCP}(s, p_r)$, можем за $\mathcal{O}(1)$ найти $lm = \text{LCP}(p_l, p_m)$ и $rm = \text{LCP}(p_r, p_m)$. Хотим найти sm. Утверждение: $sm \geqslant sm_0 = \max(\min(sl,lm), \min(sr,rm))$.

Алгоритм: $\texttt{sm} = \texttt{sm}_0\texttt{; while (s[sm] == p}_m\texttt{[sm]) sm++;}$. Алгоритм сделает $\mathcal{O}(|s|)$ операций sm++.

Замечание 6.4.1. Оба описанных алгоритма работают для произвольного алфавита.

6.4.1. fm-index

Ещё один способ поиска подстроки в тексте t с помощью $SA(t)$ заточенный под маленькие алфавиты. Обычно применяется биоинформатиками для их любимого алфавита $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$.

Главная идея: будем прикладывать строку $s = s_1 s_2 \ldots s_n$ с конца, с s_n. Пусть мы уже обработали суффикс $(i,n]$ строки s и знаем отрезок $[l_i,r_i)$ массива $SA(t)$ — где может начинаться $s\,(i,n]$.

Допишем s_i в начало, получится, что это символ последнего столбца $SA(t)$.
Пусть $p[c,r]$ = сколько раз символ c встречается на префиксе $[0,r)$ последнего столбца $SA(t)$.
Пусть $st[c]$ – позиция первого c в первом столбце. Тогда $l_{i+1} = st_{s_i} + p[s_i,l_i]$ и $r_{i+1} = st_{s_i} + p[s_i,r_i]$.
Начинаем мы с отрезка $l_0 = 0, r_0 = |t|$ и n раз дописываем символ в начало за $\mathcal{O}(1)$.

Итого. Время поиска $\mathcal{O}(|s|)$. Время на предподсчёт $= \mathcal{O}(|t|) =$ время построения $SA(t)$.
Память на предподсчёт $= k \cdot n \log_2 n$ бит памяти, где k – размер алфавита.
$\forall r \sum_c p[c,r] = r \Rightarrow$ количество слоёв p можно уменьшить до $k-1$.

• Если мы хотим искать сразу «бор строк s», то время поиска автоматически $\mathcal{O}$(размера бора).

Лекция #7: Алгоритмы на графах

28 апреля 2025

7.1. Рёберная 3-связность за $\mathcal{O}(E)$

Def 7.1.1. *Рёберная k-связность – отношение эквивалентности на вершинах «$\exists k$ рёберно непересекающихся пути между вершинами». Компоненты k-связности – классы эквивалентности. m-разрез – m рёбер, при удалении которых теряется связность ($m = k - 1$).*

Задача. Найти все компоненты 3-связности и все 2-разрезы.

Алгоритм. Берём любой остов, рёбрам e не остова сопостовляем случайные числа w_e, рёбрам e остова такие числа w_e, чтобы `XOR`-ы в вершинах были 0, это можно сделать динамикой по дереву от листьев. Сумма `XOR`-ов всех вершин ноль независимо от $w_e \Rightarrow$ `XOR` в корне $= 0$.

Теперь рассмотрим $\forall$ разрез (S, T). $E(S, T)$ – мн-во рёбер, разделяющих 2 компоненты.

Lm 7.1.2. $\bigoplus\limits_{e \in E(S,T)} w_e = 0$.

Доказательство. $0 = \bigoplus\limits_{v \in S} \Big[\bigoplus\limits_{e \in edges(v)} w_e \Big] = \bigoplus\limits_{e \in E(S,T)} w_e$, т.к. остальные рёбра учтены по два раза. ■

Следствие 7.1.3. e – мост $\Rightarrow w_e = 0$

Следствие 7.1.4. $\{e_1, e_2\}$ – 2-разрез $\Rightarrow w_{e_1} = w_{e_2}$

Замечание 7.1.5. Треугольник – три пересекающихся 2-разреза, тогда все три веса равны.

64-битных числе достаточно, чтобы вероятность ошибки было пренебрежимо мала.

Итог. Рандомизированно за $\mathcal{O}(E)$ нашли все 2-разрезы и компоненты рёбрной 3-связности.

7.2. Алгоритм Эпштейна для k-го пути

[Eppstein'98]. Решение за $\mathcal{O}(m + n \log n + k)$. У нас чуть медленней.

Задача. Дан орграф, на котором мы умеем быстро строить дерево кратчайших путей (например, $w_e \geqslant 0$ или ацикличный), найти среди всех (необязательно простых) путей из s в t (концы фиксированы) k-ую статистику по суммарному весу рёбер.

Возможное применение для DP. Возьмём стандартную линейную динамику «дойти из 1 в n, переходя каждый раз вперёд или на 2, или на 3, собрать максимальную сумму». Попросим найти из всего множества способов дойти из 1 в n сразу k максимумов. Решение: строим ацикличный граф динамики, на нём Эпштейна.

Решение за $\mathcal{O}(m + n \log m + k \log(mk))$ (асимптотика для $w_e \geqslant 0$).

1. $\mathcal{O}(m + n \log n)$. Построим дерево кратчайших путей из t по обратным рёбрам.
$d[v]$ — расстояние от v до t, p_v – отец в дереве кратчайших путей (на шаг ближе к t).

2. $\forall v$ насчитаем кучу $h[v]$ возможных ответвлений от кратчайшего пути $v \rightsquigarrow t$, каждое ответвление описывается числом Δ — насколько путь длиннее кратчайшего $v \rightsquigarrow t$.

$h[v] = h[p_v] \cup \{(d[x]+w_e)-d[v] \mid e\colon v \to x, x \neq p_v\}$, в $h[p_v]$ уже лежат ответвления сделанные дальше по пути, $(d[x]+w_e) - d[v]$ это то, на сколько вес ответвления больше веса кратчайшего пути.

$\forall v\, |h[v]| \leqslant m$, кучу из чисел $\{(d[x]+w_e)-d[v]\}$ строим за линию $\Rightarrow$ общее время $\mathcal{O}(m + n \log m)$.

3. Изначально множество кандидатов $H = h[s]$, далее k раз
за $\mathcal{O}(\log(km))$ достаём из H очередной минимум и добавляем $\leqslant m$ новых кандидатов.

```
d[],p[] = Dijkstra(t) // насчитали расстояния и дерево кратчайших путей

for v: // динамика, перебираем вершины от корня дерева
    h[v] = h[p[v]].copy() // используем персистентность
    for e in g[v]: // e : v → b[e]
        h[v].add({(d[b[e]]+w[e])-d[v], e}) // запомним Δ веса и номер ребра
H = h[s]

for k:
    <Delta,e> = H.extractMin() // e : a[e] → b[e]
    // Если мы ответвимся только по e, а дальше пойдём оптимально, будет +Delta.
    // Если после e сделаем ещё ответвление, получим увеличение на +Delta+t, t ∈ h[b[e]].
    H = H.merge(h[b[e]].increase(Delta)) // увеличить всех на Delta
```

В качестве $h[v]$ нам нужна структура данных, которая быстро умеет

`add`, `extractMin`, `merge`, `copy`, `incrementAll`

Подойдёт персистентная leftist heap (левацкая куча) и хранение числа-увеличителя.

7.3. Алгоритм двух Китайцев (Chu, Liu)

Задача. Найти в орграфе ориентированное остовное дереве с корнем в `root` минимального веса.

• **Решение за $\mathcal{O}(m \cdot n)$.**

Пусть все вершины достижимы из корня, иначе решения нет.
Наблюдение. $\forall v \neq$ `root` в ответ входит ровно одно ребро, входящее в v $\Rightarrow$ если вычесть из всех входящих в v рёбер число x, вес любого остова уменьшится на x $\Rightarrow$ MST останется тем же.

$\forall v \neq$ `root` найдём $x_v = \min w(u, v)$ и вычтем x_v из входящих в v ребер.
Все рёбра стали $\geqslant 0$, и в каждую вершину v входит хотя бы одно ребро e_v веса 0.
Если теперь есть дерево из рёбер нулевого веса, оно минимально $\Rightarrow$ нашли ответ.

Иначе есть цикл из нулей. Сожмём цикл, получим новую вершину-цикл, вычтем из входящих в неё рёбер минимум, рекурсивно продолжим алгоритм. После выхода из рекурсии мы получим MST для графа, где одна вершина – сжатый цикл. Цикл нужно расжать: добавить в остовное дерево все рёбра цикла кроме одного (одна из вершин цикла уже имеет отца в дереве).

При расжатии добавлены только нулевые рёбра $\Rightarrow$ ответ оптимален.
Время работы: Фаза $\mathcal{O}(E)$, за фазу хотя бы на 1 вершину стало меньше $\Rightarrow$ $\leqslant n$ фаз $\Rightarrow \mathcal{O}(nm)$.

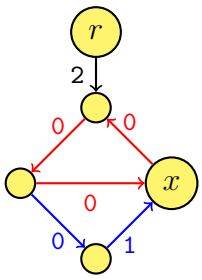
• **Решение за $\mathcal{O}(m \log n)$.**

Для каждой вершины будем хранить $h[v]$ – кучу всех входящих в v рёбер.
Построим все $h[v]$ мы за $\mathcal{O}(m)$. Далее от $h[v]$ нам нужны будут операции:
`+=x`, `merge`, `getMin`, `extractMin`; нам подойдёт skew heap и хранение числа-увеличителя.

Для сжатия циклов у нас будет `DSU`. Операция $next(v)$ для вершины v: выбрать min входящее ребро $e\colon x \to v$ (не петлю) из $h[v]$, уменьшить содержимое $h[v]$ на w_e `(*)`, вернуть начало e – вершину `DSU.get`(x). Как достать не петлю? Пропустить циклом `while` петли.
«Пропускать петли» — самая долгая часть, суммарно работает за $\mathcal{O}(m \log n)$.

Алгоритм. Начнём с пустого остова. Пока есть не подцепленная вершина v, берём её и пытаемся подцепить. Переходим из v в $next(v)$, пока или не дойдём до уже подцепленной вершины, или не зациклимся. Если зациклимся, снимаем цикл со стека и сжимаем (`DSU` и `merge` куч).
При `merge` вершин цикла важно, что мы уже сделали минимальные рёбра нулями (из разных куч вычитались разные числа). После сжатия цикла продолжаем движение к корню. $\mathcal{O}(n \log n)$.

Если нам нужен только вес остова, его мы знаем как сумму вычтенных величин в `(*)`.



Сами рёбра остова получить сложнее. Можно алгоритмом Прима, запущенным от корня дерева r на множестве выбранных в остов и поучаствовавших в сжатиях рёбер (на картинке такие все). Весом ребра будет «момент сжатия» или $+\infty$ для остовных. $\mathcal{O}(m \log n)$, при желании $\mathcal{O}(m + n \log n)$.

Сперва мы найдём красный цикл по нулям. После сжатия у вершины-цикла есть два входящих ребра – 1 и 2. Затем найдём синий цикл. При восстановлении MST алгоритмом Прима важно брать красные рёбра с б*о*льшим приоритетом, чтобы посетить x именно по красному ребру.

Другой способ получить рёбра. Помнить циклы в порядке сжатия, расжимать их в обратном порядке, добавляя в остов все рёбра кроме одного, которое идёт в «вершину у которой уже есть отец». В этом способе есть нюанс: нужно быстро «подняться по версии дерева DSU без сжатия путей», это задача LA, при желании делается за $\mathcal{O}(1)$. Итого $\mathcal{O}(m + n)$.

• **Ссылочки. Литературa.**

[CF]. Отличный текст от Олега Давыдова.
[wiki]. Chu–Liu/Edmonds algorithm.
[wiki.algocode.ru]. Описание на русском с коротким кодом.

7.4. Dynamic Graphs

[Erik'12|L20]. Лекция Эрика Демэйна с обзором темы.
[Erik'12|Full]. В курсе ещё много чего, например link-cut-tree и euler-tour-tree.

Dynamic Graph = динамически меняющийся граф. Алгоритмы на Dynamic Graphs должны обрабатывать изменения графов. Обычно это добавления и удаления рёбер. Если из задачи следует, что у рёбер есть веса, то и изменение весов. Часто отдельно различают update-time (запрос-изменение) и query-time (запрос-не-изменение), например, «достижимость», «связность».

Различают задачи incremental (только добавления), decremental и fully-dynamic.

Одно из первых нетривиальных достижений – удаление рёбер в неориентированном графе и ответы на запросы о связности за $\mathcal{O}(q+nm)$, где q – суммарное число всех запросов. [Shiloach'81]

За $\mathcal{O}(\log n)$ умеют поддерживать MST в динамически меняющемся планарном графе [Epp'92]

В статье про кратчайшие пути в dynamic планарном графе за update-time $\mathcal{O}(n^{4/5})$ и query-time $\mathcal{O}(\log^2 n)$ [Karczmarz'21] можно найти обзор про кратчайшие пути в обычных dynamic графах.

Связность умеют за $\mathcal{O}(\log^2 n)$, MST за $\mathcal{O}(\log^4 n)$, 2-связность за $\mathcal{O}(poly(\log n))$. [Thorup'01]
Все времена выше амортизированные, без амортизации умеют только $\mathcal{O}(\sqrt{n})$.

С ориентированным (directed) графами всё непросто. Есть три основных задачи – поддерживать матрицу достижимости, SSR (single-source-reachability) и поддерживать SCC (strong-connectivity-components). В быстрые решения для fully-dynamic-SCC люди не верят: если сделать и update, и query за $\mathcal{O}(m^{1-\varepsilon})$, получится прорыв для SETH.
Decremental-SCC недавно научились за $\tilde{\mathcal{O}}(m)$ в сумме на все update-ы [2019'Bernstein].
Набор красивых идей для задачи: [2008'Roddity].

7.5. Dynamic Connectivity в ориентированном графе

Мы уже умеем за $\mathcal{O}(nm/w)$ насчитывать матрицу достижимости.

Если рёбра только добавляются, можно все запросы добавления обработать в сумме за $\mathcal{O}(n^3/w)$:
`add(a,b):` $\forall x$: `d[x,a]=1 and d[x,b]=0 do d[x,b]=1, d[x]|=d[b];`.
Здесь «`|=`» выполнится не более n^2 раз, а `for` x переберёт не более $\mathcal{O}(n/w)$ лишних.

Если рёбра только удаляются, задача для неорграфа и SSR для орграфа делаются примерно одинакого за $\mathcal{O}(q+nm)$ [Shiloach'81]. Опишем SSR (single-source-reachability) из s:

Поддерживаем дерево bfs из s, когда удаляется ребро $a \to b$ из дерева кратчайших путей, ищем замену входящего в b ребра за $\mathcal{O}(deg_b)$, или находим, или вершина поднимается выше и вызывает рекурсивно подъём всех детей в дереве кратчайших путей. $\mathcal{O}(nm)$ на все подъёмы.

Алгоритм для fully-dynamic версии в DAG. Update $\mathcal{O}(n^2)$, *query* $\mathcal{O}(1)$ *(King,Sagert'99).*
Будем поддерживать $c[a,b]$ = число путей из a в b по модулю p,
где p — большое случайное простое число.
Ответ на запрос за $\mathcal{O}(1)$: `connected`$(a,b) = (c[a,b] \neq 0)$.

Пересчёт при добавлении и удалении за $\mathcal{O}(n^2)$, как во Флойде.
Для не DAG есть, например, такое обобщение: [King'99] (Ctrl+F: transitive-closure).

Теоретически крутая модификация алгоритма. Давайте не пересчитывать матрицу c сразу, а хранить k отложенных операций вида c `+=` $A_i \cdot B_i$: $(n \times 1) \times (1 \times n)$, тогда обращение к c работает за $\mathcal{O}(k)$, update работает за $\mathcal{O}(nk)$, и когда отложенных операций стало ровно k, нужно умножить матрицы $A_1A_2 \ldots A_k$ и $B_1B_2 \ldots B_k$: $(n \times k) \times (k \times n)$. Если $k = n^\varepsilon$, то
мы хотим $w(1, \varepsilon, 1) = 1 + 2\varepsilon \Rightarrow \varepsilon = 0.575$, query-time $\mathcal{O}(n^{0.575})$, update-time $\mathcal{O}(n^{1.575})$.

7.6. Dynamic Connectivity и MST в Offline

Пусть у нас есть массив из q запросов «`+`», «`-`», «`?`». Будем считать, что у каждого `+/-` запроса есть парный и изначально граф пустой (все рёбра можно добавлять через «`+`»).

Превратим рёбра в отрезки: ребро e добавлено, потому удалено, живо в моменты времени $[l_e, r_e)$. Есть моменты времени $[0, q)$, сделаем разделяй-и-властвуй (дерево отрезков) по времени:
`solve [l,r) { m=(l+r)/2 ; solve [l,m); solve[m,r); }`.

Что происходит с рёбрами? При входе в рекурсию $\forall e$ или $r \leqslant l_e$ или $r_e \leqslant l \Rightarrow$ ребро на отрезке времени $[l, r)$ всегда не в графе (мн-во A), или $l_e \leqslant l$ и $r \leqslant r_e \Rightarrow$ ребро всегда в графе (мн-во B), либо ребро остаётся «интересным» (мн-во E), интересных рёбер не более $k = r - l$.

Далее возможны два разных подхода «СНМ с откатами» и «сжать компоненты».

СНМ с откатами. Добавим все рёбра B в СНМ, сделаем два рекурсивных вызова, откатим рёбра B из СНМ за то же время, что добавляли. Работает за $\mathcal{O}(m \log m)$ операций с СНМ, каждая из которых за $\mathcal{O}(\log n)$ так как амортизация убивается из-за откатов. $\mathcal{O}(m \log m \log n)$.

Сжатие компонент. Наш граф G пустой. Добавим в наш пустой граф рёбра B, и выделим компоненты связности. Компоненты связности задают новый пустой граф G_1 с номерами вершин в $[0, k)$, перенумеруем вершины в E и запросах «`?`», и передадим в рекурсию $\langle G_1, E, \text{запросы «?»} \rangle$. Работает за $\mathcal{O}(m \log m)$: $T(m) = 2T(\frac{m}{2}) + m$.

• Метод откатов

Прелесть в том, что мы научились в offline отменять любой запрос изменения. Если мы умеем быстро без амортизации добавлять в online за $\mathcal{O}(t)$, то умеем удалять в offline за $\mathcal{O}(m \log m \cdot t)$.

Пример: 2D точки на плоскости добавляются и удаляются, поддерживать две ближайшие.

• Метод сжатия компонент

Если мы научимся, имея задачу размера $\leqslant n$ и $\leqslant k$ рёбер за $\mathcal{O}(n+k)$, сжатием строить меньшую задачу размера $\mathcal{O}(k)$, то получим offline-решение. Далее нетривиальный пример на тему.

• Dynamic MST Offline за $\mathcal{O}(m \log m)$.

Изменение веса ребра `=` удалить `+` добавить. Удаление рёбер получится через разделяйку.

Осталось научиться быстро спускаться в рекурсию. Пусть у нас уже есть MST на всех добавленных выше в рекурсии рёбрах, в рекурсию мы передаём рёбра E: $|E| = k$, у рёбер E есть $2k$ концов, которые индуцируют на MST дерево из $\leqslant 4k$ вершин. Возьмём пути этого $4k$-дерева, на каждом насчитаем max (пути не пересекаются $\Rightarrow$ в сумме за линию), передадим в рекурсию дерево из $\leqslant 4k$ вершин с весами «максимум на пути» и множество E с обновлёнными концами.

$\mathcal{O}(n + k)$ на сжатие MST при добавлении рёбер $\Rightarrow \mathcal{O}(m \log m)$ в сумме.

7.7. Dynamic Connectivity Online

[Thorup'98][Thorup'01] Оригинальная статья, две версии.
[Erik'07|L05]. Лекция профессора Эрика Демэйна.
[wiki]. Там есть «Cutset structure», работающий за $\mathcal{O}(poly(\log n))$ в худшем.
[D-Tree'2022]. Говорят, недавно научились лучше.

Задача. Структура данных, умеющая добавлять, удалять рёбра, и проверять достижимость.

Что мы уже умеем.
Решать задачу в offline за $\mathcal{O}(m \log m)$ (предыдущая глава).
Решать версию без удалений рёбер за $\mathcal{O}(\alpha(m, n))$ на запрос (просто DSU).
Решать версию задачи, в которой в каждый момент времени граф – лес за $\mathcal{O}(\log n)$ на запрос.
Решение: ETT, Euler Tour Trees (treap по неявному ключу на эйлеровом обходе).

Структура.
Для удаления заведем иерархию остовных лесов.
У каждого ребра есть уровень $\text{level}(e) \in [1, \log n]$. Исходно у всех ребер уровень $\log n$.
Обозначим за G_i граф из ребер уровня $\leqslant i$. То есть $G_{\log n}$ – исходный граф.
$F_{\log n}$ – min остовный лес (по сумме уровней ребер). $F_i = F_{\log n} \cap G_i$ – min остовный лес G_i.
Для всех F_i храним лес в виде ETT с прибамбасами.

Инвариант, за которым нужно следить: в G_i все компоненты размера $\leqslant 2^i$.

Добавление ребра. Просто попытаемся добавить ребро в $F_{\log n}$. $\mathcal{O}(\log n)$.

Алгоритм удаления ребра.
Удалили ребро $e = (v, u)$. Если $e \notin F_{\log n}$, ничего не происходит. Иначе $e \in F_{\text{level}(e)}, \dots, F_{\log n}$.
Нужно из всех них e удалить (в ETT cut работает за $\mathcal{O}(\log n)$), и искать замену для e:
Перебираем $i = \text{level}(e), \dots, \log n$ и ищем замену уровня i: пусть v и u лежат в деревьях T_v и T_u уровня i, до удаления e они были одной компонентой $\Rightarrow |T_v| + |T_u| \leqslant 2^i$.
Пусть $|T_v| \leqslant |T_u| \Rightarrow |T_v| \leqslant 2^{i-1}$ (взяли меньшую половину).

Понижаем до $i-1$ все ребра уровня i, лежащие в дереве T_v, $(i-1)$-рёбра добавляем в остов F_{i-1}.
Перебираем рёбра уровня i, у которых есть конец в T_v (для этого в вершине treap от ETT $\forall j$ поддерживаем «число рёбер ранга j в поддереве»).
Если ребро ведет в T_u, мы нашли замену, вставляем его в $F_i, \dots, F_{\log n}$ за $\mathcal{O}(\log^2 n)$ и останавливаемся. Иначе ребро ведёт в T_v, понижаем уровень ребра до $i-1$ (хотя бы на 1).

Время работы.
Выбираем из T_v и T_u меньшее за $\mathcal{O}(1)$ (знаем размеры ETT).
Уровень ребра никогда не понизится ниже 1 (на уровне 1 компоненты размера $2^1 = 2$).
Мы перебираем рёбра T_v, каждый просмотр ребра T_v понижает уровень ребра и работает за $\mathcal{O}(\log n)$: подняться от конца ребра к корню treap, чтобы понять, куда попали. Суммарно понижений уровня будет $m \log n$, каждое за $\log n$ $\Rightarrow$ амортизированно $log^2 n$ на ребро

Зачем нужно было именно MST?
Благодаря MST при поиске замены ребра e_0 мы уверены, что уровень замены неменьше $\Rightarrow$ просматриваем только e: $\text{level}(e) \geqslant \text{level}(e_0) \Rightarrow$ только рёбра, уровень которых можно понизить.

Замечание 7.7.1. Важно, что сначала мы понижаем уровень древесных рёбер T_v до $i-1$. Иначе при понижении *не*древесных до $i-1$ T_v могло перестать быть MST.

7.8. Дерево доминаторов

Дан орграф и выделенная вершина s. Нас интересуют пути из s до остальных вершин.

Def 7.8.1. *Говорят, что a доминирует b, если $\forall$ путь $s \rightsquigarrow b$ проходит через a.*

Заметим, что s доминирует всех, а все доминаторы b лежат на любом пути $s \rightsquigarrow b \Rightarrow$ среди них есть ближайший к b, обозначим его $idom(b)$ и обзовём «непосредственный доминатор b».

Задача. Построить «дерево доминаторов» — $\forall v$ найти $idom(v)$.

Простейшее решение. Попробовать удалять каждую вершину a, запускать dfs из s, проверять достижимость. Если $\exists s \rightsquigarrow b$ до удаления a и $\nexists$ после, то a – доминатор b. $\mathcal{O}(nm)$.

Чуть более умное решение. $N(v) = \{\text{доминаторов } v\}$, изначально пусть $N(s) = \{s\}, N(v) = V$, затем итеративно будем обновлять $N(v) = \cap_{x \to v} N(x) \cup \{v\}$, пока $N(v)$ меняются. Говорят, такое быстро сходится. Можно ускорить, взяв любой остов, и начав с $N(v) = \{\text{предки } v \text{ в остове}\}$.

Замечание 7.8.2. Даже для DAG-ов задача поиска дерева доминаторов не проще чем задача поиска LCA: $\forall$ алгоритм поиска дерева доминаторов решает и задачу LCA-offline. Доказательство: возьмём дерево, на котором нужно считать LCA, ориентируем его от корня, для запроса $i\colon \langle a_i, b_i \rangle$ создадим вершину i с входящими рёбрами из a_i и b_i, её доминатор это ровно $LCA(a_i, b_i)$.

Простое практически эффективное решение. Инициализируем дерево доминаторов, как любой остов dfs из s. Будем пересчитывать отцов: $p_v = LCA\{x\colon \exists \text{ребро } x \to v\}$, пока p_v будут меняться. Фазу изменений можно делать, как один dfs с DSU за $\mathcal{O}(m\alpha)$, а фаз часто $\mathcal{O}(1)$.

Ещё чуть лучше. $\mathcal{O}(m \log n)$. Будем использовать LCA на «текущем дереве доминаторов», каждое изменение $idom[v] = x$ меняет дерево, будем считать «LCA на меняющемся дереве». В таком случае dfs+DSU уже не подойдёт для LCA, зато фаз всегда будет не более 6 (эмпирический результат на всех тестах/задачах, до которых автор конспекта смог дотянуться).

Быстрое решение. $\mathcal{O}(m\alpha)$. Возьмём остовное дерево dfs из s. Перенумеруем вершины в порядке времени входа dfs. В искомом дереве доминаторов времена входа убывают на пути к корню.

TODO

[algocode]. Описание на русском от Филиппа Грибова.
[1979'Tarjan]. Оригинальная статья, $\mathcal{O}(m \log n)$.
[2004'Tarjan]. Сравнение всех реализаций на практике.
[wiki]. На wiki про всё это тоже чуть-чуть есть.

Лекция #8: Факторизация

3 апреля 2024

8.1. Метод Крайчика

Как обычно, наша задача – найти любой нетривиальный делитель x.
Метод Ферма: найдём v, u: $v^2 - u^2 = n \Rightarrow v \pm u$ – нетривиальный делитель n.
Обобщим: v, u: $v^2 - u^2 \equiv 0 \bmod n,\ v \pm u \not\equiv 0 \bmod n \Rightarrow gcd(n, v \pm u)$ – нетривиальный делитель n.

Суть Крайчика: взять $x_i = \lfloor \sqrt{n} \rfloor + i,\ y_i = x_i^2 - n$, найти подмножество y-ков, дающих в произведении точный квадрат, получается $x_{i_1}^2 \cdot \dots \cdot x_{i_k}^2 \equiv y_{i_1}^2 \cdot \dots \cdot y_{i_k}^2 \bmod n$, применяем Ферма.

Задача: дано множество $\{y_i\}$, найти подмножество, являющееся точным квадратом.
Решение: число является квадратом iff в его факторизации все степени простых чётны $\Rightarrow$ факторизуем y_i, получаем вектора z_i «чётности степеней простых», находим подмножество векторов z_i, в сумме над $\mathbb{F}_2$ дающее 0.
Реализация: Гаусс над разреженной матрицей.

Небольшая проблема – мы как раз учимся факторизовать, а тут нужно факторизовать y_i.
Чтобы разрешить эту неловкость, будем брать только b-гладкие y_i.

Def 8.1.1. *Число x называется b-гладким, если все простые в разложении x не больше b.*

Замечание 8.1.2. b-гладкое число x легко факторизовать за $\mathcal{O}(\frac{b}{\log b} + \log x)$ делений.

• Алгоритм `factorize(n)`

1. Фиксируем константу k ($b = \texttt{prime}[k]$).
2. Генерируем числа y_i, сразу факторизуем их за $\mathcal{O}(k + \log y_i)$.
3. Для всех y_i, оказавшихся b-гладкими, скармливаем Гауссу z_i, вектор над $\mathbb{F}_2$ длины k.
4. Каждый раз, когда Гаусс видит, что очередной вектор – линейная комбинация предыдущих, подставляем в тест Ферма $v = \prod x_{i_j}$ и $u = \left(\prod y_{i_j}\right)^{1/2}$.

• Время работы

Алгоритм зависнет для простого n и имеет шанс найти делитель для составного n.
При успешном завершении время работы $\mathcal{O}(k^3 + k^2 t + km)$, где t – число запусков теста Ферма, а m – общее число всех y_i.

Чтобы сделать алгоритм конечным и вероятностным, можно скармливать вектора Гауссу с вероятностью $\frac{1}{2}$ и останавливаться после первого же теста Ферма.
Тогда время $\mathcal{O}(k^3 + km)$, и мы верим в оценку на вероятность p: $0 < \varepsilon \leqslant p \leqslant 1$.

• Оптимизация

Можно для всех чисел $p_i = \texttt{prime}[i]$ заранее посчитать r_i: $(\pm r_i)^2 - n \equiv 0 \bmod p_i$.
Тогда множество таких j, что $p_i \mid y_j$ равно $\{\pm r_i + k \cdot p_i - \lfloor \sqrt{n} \rfloor \mid k \in \mathbb{Z}\}$. Факторизация:

```
1 for i for j while (p_i | y_j) do { y_j /= p_i; v_j[i]++; }
```

Научились факторизовать все y_j за $\sum\limits_{i=1..k} \frac{m}{i \log i} = \Theta(m \log \log k)$.

Предположим, что $\mathcal{O}(m \log \log k)$ единиц по m векторам распределены равномерно, тогда в выбранных $(k+1)$-ом векторе будет лишь $\mathcal{O}(k \log \log k)$ не нулей. Можно написать разреженного Гаусса, который хранит в строках лишь списки ненулевых элементов и операции над

строками-списками длин a и b делает за $\mathcal{O}(a+b)$. Можно воспользоваться алгоритмом Видеманна (Wiedemann), который решит систему за $\mathcal{O}(k \cdot z)$, где z – число не нулей в матрице.

Итого время $\mathcal{O}(k^3 + km)$ можно улучшить до $\mathcal{O}((k^2+m)\log\log k)$.
Чем меньше k, тем больше m. Нужно подобрать k: $k^2 = \Theta(m)$. Это можно сделать, например, последовательным удвоением (`for` $k \in \{1, 2, 4, 8, \dots\}$ `do` запускаем решение для $m = k^2$).

Теорема 8.1.3. Для числа n оптимальное $k = e^{\frac{1}{2}\sqrt{\log n \log\log n}}$

Следствие 8.1.4. Время Крайчика со всеми оптимизациями $T = e^{\sqrt{\log n \log\log n}}$, $\forall \varepsilon > 0 \;\; T = o(n^\varepsilon)$.

Замечание 8.1.5. Это первый, изученный нами, субэкспоненциальный алгоритм факторизации. Раньше мы умели за $\mathcal{O}(n^{1/4})$. Длина ввода есть $l = \log n \Rightarrow$ мы умели только за $exp(\Theta(l))$.

8.2. Оценки времени, математическая часть

Пусть k-гладкое число – в факторизации встречаются только первые k простых.
Грубо оценим количество k гладких от 1 до n: наши простые порядка $k \log k$, чтобы получить k-гладкое число, нужно взять произведение из $m \approx \log_{k\log k} n$ первых простых.

Число способов примерно $\frac{k^m}{m!} \approx \frac{k^m}{(m/e)^m}$, где $m \approx \frac{n}{\log(k\log k)} = \frac{\log n}{\log k + \log\log k} \approx \frac{\log n}{\log k}$. $k^{\log n/\log k} = n \Rightarrow$ k-гладких от 1 до n примерно $n \cdot \left(\frac{e\log k}{\log n}\right)^{\log n/\log k} = n \cdot \alpha$, чтобы найти k k-гладких чисел для Видемана (Гаусса), нужно рассмотреть k/α чисел, Видеман работает за $k^2 \Rightarrow$ ищем $k^2 = k/\alpha \Rightarrow$ $k = \left(\frac{\log n}{e\log k}\right)^{\log n/\log k} \Rightarrow \log k = \frac{\log n}{\log k} \cdot \log\frac{\log n}{e\log k} \Rightarrow \log^2 k \approx \log n \Rightarrow \log k \approx \sqrt{\log n}$, $k \approx 2^{\sqrt{\log n}}$ и время работы версии Крайчика с квадратичным решетом и Видеманом внутри $2^{\mathcal{O}(\sqrt{\log n})}$.

8.3. Литература

[logic.pdmi.ras]. Слайды от Николенко про Крайчика и Видемана.
[wiki|Wiedemann]. Короткое внятное описание решения системы $Ax = 0$ над $\mathbb{F}_p$ за $\mathcal{O}(n{\cdot}z)$.
[Pomerance]. Доказательства, связанные с b-гладкими числами и алгоритмом Крайчика.
[Факторизация]. Много про факторизацию на русском. И про Крайчика тоже.

Лекция #9: Паросочетание в произвольном графе

12 мая 2025

9.1. Полезные данные из прошлого

Lm 9.1.1. Лемма о дополняющем пути. Если паросочетание M не максимально, то существует дополняющий, чередующийся относительно M, путь (ЧДП), увеличивающий M.

Lm 9.1.2. Корректность Куна. Алгоритм последовательно увеличивает M дополняющими путями. $\forall v$ если в какой-то момент $\nexists$ дополняющего чередующегося пути из v, то это навсегда.

Lm 9.1.3. Симметрические разности.
M_1, M_2 – паросочетания, тогда $M_1 \triangle M_2$ – набор путей и циклов,
причём в $M_1 \triangle M_2$ есть хотя бы $\big||M_1| - |M_2|\big|$ ЧДП.
P – чередующийся относительно M_1 путь (не важно, дополняющий ли), тогда $M_1 \triangle P$ – тоже паросочетание, причём $|M_1 \triangle P| = |M_1| + \Delta P$, где $\Delta P = +1$ для ЧДП, 0 для чётного пути.

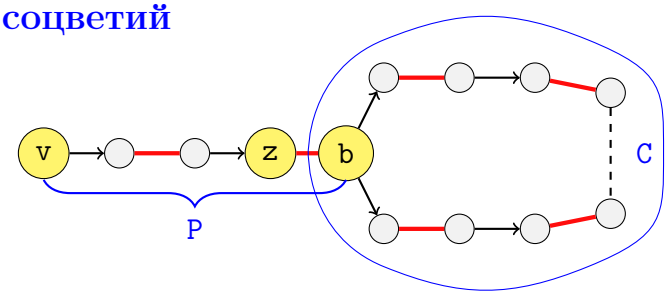
Lm 9.1.4. Поиск дополняющего пути в двудольном графе.
Чтобы из свободной вершины v 1-й доли найти дополняющий путь, запустим dfs/bfs на орграфе, где из 1-й доли ведут все рёбра, а из 2-й доли только рёбра паросочетания.

- Первые три леммы верны для произвольного графа.
- Последнюю мы сможем применить, добавив в алгоритм случай «найден нечётный цикл».

Упражнение 9.1.5. G – произвольный граф. M_1, M_2 – паросочетания в нём. Есть ли в $M_1 \triangle M_2$ нечётные циклы? *(конечно, нет, в сим.разности только чередующиеся циклы)*

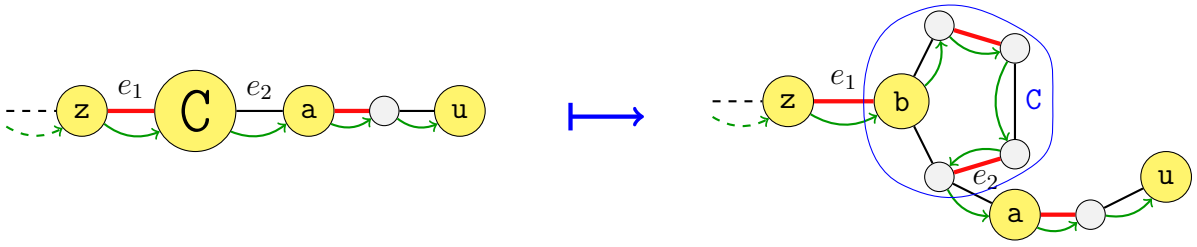
9.2. Алгоритм Эдмондса сжатия соцветий

Берём Куна и добавляем в `dfs/bfs` случай «найден нечётный цикл». Кстати, мы нашли не просто нечётный цикл, а чередующийся нечётный цикл C со стеблем P.



Теорема 9.2.1. Теорема Эдмондса⁺ [1]. Пусть стебель P начинается в v.
$G' = G/_C$ (стянули цикл C в одну вершину) $\Rightarrow$ в G есть ЧДП из v iff в G' есть ЧДП из v

Доказательство. В G' есть путь $T \Rightarrow$ в G есть.
Если путь T не проходит через вершину C, не меняем его.
Иначе заметим, что в T вершине C смежны два ребра – $e_1 = (z, b) \in M$ и $e_2 \notin M$. Расжимаем.



■

[1] На e-maxx используется слишком слабая версия теоремы, нам нужен путь именно из v.

Доказательство. В G есть путь $T\colon v \rightsquigarrow u \Rightarrow$ в G' есть путь $T'\colon v \rightsquigarrow u$.
(1) Пусть $v = b \Rightarrow P = \{b\}$. Возьмём в T последнее ребро, исходящее из цикла ($e_2\colon z \to a$, $z \in C$). $v = b$ не покрыта $M \Rightarrow$ все рёбра, исходящие из C, $\notin M \Rightarrow e_2 \notin M$.
Отрежем от T кусок до e_2, получили путь T'.

(2) В общем случае заменим свободную v на свободную b: $M \to M_1 = M \triangle P$. Относительно M_1 есть дополняющий путь $T_1\colon b \rightsquigarrow u$ ($T_1 = (M \triangle T) \triangle M_1$). По сути $T_1 = T \triangle P$, но $\triangle$ паросочетаний – пути и циклы, а $\triangle$ дополняющих путей – непонятно что. Применим (1), получили T_1' и берём $T' = (M_1' \triangle T_1') \triangle M'$, где M_1' и M' – соответствующие паросочетания в G'. ■

Доказательство закончилось. Но, возможно, вам нужны дополнительные пояснения:

Путь	Относительно	Где	Куда	Как появился?
T	M	G	$v \to u$	Дан по условию.
T_1	$M_1 = M \triangle P$	G	$b \to u$	Найден в $(M \triangle T) \triangle M_1$.
T_1'	$M_1' = M' \triangle P$	G'	$C \to u$	Отрезали от T_1 часть до *последнего ребра, исходящего из* C.
T'	M'	G'	$v \to u$	Найден в $(M_1 \triangle T_1') \triangle M$.

Lm 9.2.2. Для тех, чей мозг плавится под воздействием симметрических разностей.
Пусть $A(M)$ – свободные вершины относительно паросочетания M.
Концы всех путей в $M_1 \triangle M_2$ лежат в $A(M_1) \triangle A(M_2)$.

Таблица: почему именно такие пути мы нашли в $(M \triangle T) \triangle M_1$ и $(M_1' \triangle T_1') \triangle M'$:

Паросочетание	Свободные вершины	Покрытые вершины	Граф	Как получили
M и M'	v u	$b(C)$	G и G'	Дано по условию.
M_1 и M_1'	$b(C)$ u	v	G и G'	Поксорили M со стеблем P.
$M \triangle T$		v b u	G	Увеличили паросочетание M путём T.
$M_1' \triangle T_1'$		v C u	G'	Увеличили паросочетание M_1 путём T_1'.

9.3. Реализация за $\mathcal{O}(V^3)$

1. `for v=1..n` запустим `dfs/bfs(v)`, доставшийся нам от Куна, для поиска пути.
2. Поиск пути нашёл путь или нечётный цикл со стеблем (соцветие = цветок + стебель).
3. Сожмём нечётный цикл в одну вершину, продолжим поиск. На слове *продолжим* мы понимаем, что из `dfs/bfs` удобнее `bfs`.
   (a) Сжимаем нечётный цикл в новую вершину z за $\mathcal{O}(V \cdot$ `cycleLen`$)$.
   (b) Кидаем z в очередь, удаляем все вершины цикла из очереди.
   (c) Делаем рекурсивный вызов `continueBfs`, который возвращает путь в сжатом графе.
   (d) Если путь проходит через z, делаем расжатие пути. Возвращаем путь в исходном графе.

`continueBfs` – тот же `bfs`, но не обнуляем очередь и пометки.
Время работы = $V \cdot$ `bfs` = $V(E+V^2)$: E – просмотрели рёбра по разу, $\mathcal{O}(V^2) = \mathcal{O}(V \cdot \sum_i$ `cycleLen`$_i)$

9.4. Красивая простая реализация Эдмондса (Габов'1976)

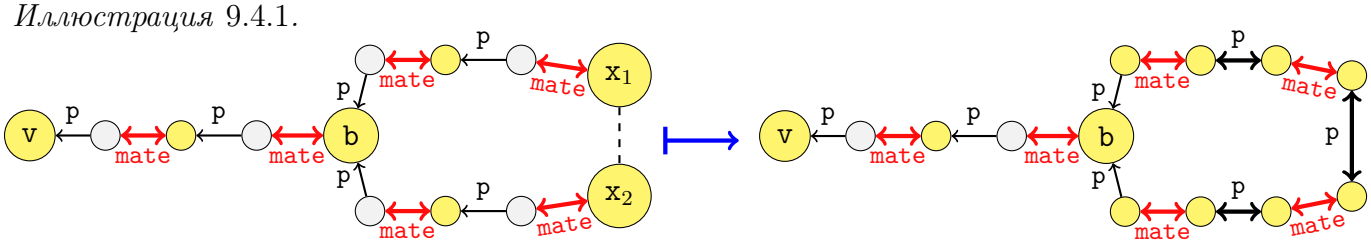
Основная схема та же, что в Куне:

```
mate = [-1,-1,...,-1] # mate[v] - пара в паросочетании
for v in 1..n
    if mate[v] == -1
        endOfPath = bfs(v)
        if endOfPath != -1
            mate = mate △ recoverPath(endOfPath)
```

`bfs` по ходу работы помечает вершины 1-й доли `used[v] = 1`.
Вершины 2-й доли – пары вершин 1-й доли.
Для ∀ вершины `w` 2-й доли храним предыдущую `p[w]` в 1-й доли.
Таким образом `w`, `p[w]`, `mate[p[w]]`, `p[mate[p[w]]]`, ... – путь $\mathtt{w} \rightsquigarrow \mathtt{v}$.
Рёбра $\{\mathtt{w} \to \mathtt{p[mate[w]]} \mid \mathtt{used[w]} = 1\}$ образуют дерево T с корнем в `v`.

Интересен случай, когда мы пытаемся пойти из помеченной вершины $\mathtt{x}_1$ в помеченную вершину $\mathtt{x}_2$. Тогда налицо нечётный цикл C, причём до всех вершин этого цикла кроме основания мы научились доходить, как до вершин 1-й доли.

Иллюстрация 9.4.1.



Замечание 9.4.2. Здесь жёлтым отмечены вершины, до которых мы умеем доходить, как до вершин первой доли. Они же лежат в очереди.

• Главная идея

Сжимая цикл C, вместо того, чтобы создавать новую вершину и видоизменять граф, лишь запомним, что все вершины C лежат именно в C. Для этого будем ∀v поддерживать `base[v]` – основание цикла, в котором лежит `v`.

• Алгоритм действий

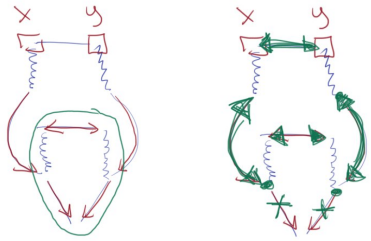
1. За $\mathcal{O}(n)$ выделить в дереве T вершину $\mathtt{b} = \mathtt{lca}(\mathtt{x}_1, \mathtt{x}_2)$.
2. Пусть $\mathtt{b}_i$ – первая вершина на пути $\mathtt{x}_i \rightsquigarrow \mathtt{b}$: $\mathtt{base}[\mathtt{b}_i] = \mathtt{base}[\mathtt{b}]$.
   Пройти по путям P_1: $\mathtt{x}_1 \rightsquigarrow \mathtt{b}_1$ и P_2: $\mathtt{x}_2 \rightsquigarrow \mathtt{b}_2$ и проставить новые ссылки `p[]`.
   $\forall \mathtt{z} \in P_1 \cup P_2$: `used[z]=0` пометить z: `used[z]=1` и добавить в очередь `bfs`-а.
   Важно: менять $\mathtt{p}[\mathtt{b}_1]$ и $\mathtt{p}[\mathtt{b}_2]$ нельзя, также нужно идти именно до b_1 и b_2 (до b нельзя, иначе мы можем случайно поменять $\mathtt{p}[\mathtt{b}_1]$).
3. $\forall \mathtt{u} \in P_1 \cup P_2$ $\forall \mathtt{w}$: $\mathtt{base}[\mathtt{w}] = \mathtt{base}[\mathtt{u}]$ сделать $\mathtt{base}[\mathtt{w}] := \mathtt{b}$ (`base[]` – простейшее DSU).

• Обоснование

1. В итоге мы пометим `used[v]=1` все v, которые достижимы, как вершины первой доли.
2. Ссылки `p[]`: пути $x_1 \rightsquigarrow b_1$ и $x_2 \rightsquigarrow b_2$ пересекаются только по $base[b_1] = base[b_2] \Rightarrow \forall w \in C$ откат по ссылкам `p[]` дойдёт до w: $base[w] = b$. Мы не меняли ссылки $p[i]$: $base[i] = b \Rightarrow$ откат

от w дойдёт до b. *Итого:* в каждый момент времени $\forall v\colon used[v]=1$
путь $v \to mate[v] \to p[mate[v]] \to mate[p[mate[v]]] \to \dots$ – корректный чередующийся путь.

Иллюстрация 9.4.3. Почему нельзя от $x_1{=}x$ и $x_2{=}y$ откатываться до $LCA(x_1,x_2){=}b$ (код бы упростился, можно было бы обойтись без `base[]`, без DSU)? Потому что ссылки `p[]` могут зациклиться, см.рис.



9.5. Оптимизации

Как и для Куна, основные оптимизации – не удалять пометки и жадно инициализировать паросочетание. Важно, что второе толком не работает без первого.

- Удаляем пометки, пока не нашли ЧДП $\Rightarrow$ $\mathcal{O}(V \cdot \texttt{bfs})$ превратилось в $\mathcal{O}(|M| \cdot \texttt{bfs})$.
- Жадно находим за $\mathcal{O}(E)$ паросочетание размером $\geqslant \frac{|M|}{2}$ $\Rightarrow$ ускорили ещё в два раза.

9.6. DSU и $\mathcal{O}(VE \cdot \alpha)$

Научимся случай «нечётный цикл» обрабатывать за $\mathcal{O}(k \cdot \alpha)$, где k – длина цикла в сжатом графе. Тогда суммарное время работы `bfs` будет $\mathcal{O}((V+E) \cdot \alpha)$.

• Поиск LCA: $\mathcal{O}(V) \to \mathcal{O}(k \cdot \alpha)$

Во-первых, мы будем переходить не `v` $\to$ `p[mate[v]]`, а сразу `v` $\to$ `p[mate[base[v]]]`.
Во-вторых, LCA(x_1, x_2) можно искать не за $\mathcal{O}(n)$, а за $\mathcal{O}(k)$, идя от x_1 и x_2 двумя указателями.

• Собственно сжатие цикла: $\mathcal{O}(V) \to \mathcal{O}(k \cdot \alpha)$

`base[v]` $\to$ `DSU.get(v)`. По ходу поиска LCA сохраним все пройденные вершины v, после LCA сделаем им `DSU.join` в b.

• Добавление вершин в очередь

Если вершина лежит в уже сжатом цикле, она точно в очереди. Остались только вершины вида `mate[base[v]]`. Каждую такую попробуем добавить.

• Обновление ссылок `p[]`

Если мы хотим только проверить наличие дополняющего пути, то ссылки `p[]` для вершин, уже сжатых в составе цикла, не нужны. Нужна только `p[base[v]]`. Такую обновить просто.

Для восстановлении пути основная идея: при сжатии цикла не присваивать ссылки `p[]` для внутренних вершин цикла w (вершин, которые мы изначально определили во 2-ую долю), а при добавлении w в очередь запомнить, что w была получена «при сжатии цикла, образованного ребром (x,y) при откате по ссылкам из x» $\Rightarrow$ $path(w \rightsquigarrow b) = path^{rev}(x \rightsquigarrow w) + (x,y) + path(y \rightsquigarrow b)$.

• Реализация

[rkolganov]. `dfs`-реализация за $\mathcal{O}(VE \cdot \alpha)$.
[skopeliovich]. `dfs`-реализация за $\mathcal{O}(|M| \cdot V^2)$.

9.7. Реализации через `dfs`

Если мы хотим вместо `bfs` использовать `dfs`, то при сжатии цикла все вершины цикла нужно

положить в `vector`, пометить, а потом от каждой сделать рекурсивный вызов.

У `dfs` есть огромный плюс – когда мы идём из `v` в `x` и обнаруживаем нечётный цикл, то `LCA(v, x) = inTime[v] < inTime[x] ? base[v] : base[x]`; $\Rightarrow$ ищем LCA за $\mathcal{O}(1)$.

9.8. Альтернативное понимание реализации

Попробуем вообще не думать про нечётные циклы. Будем поддерживать массивы `p[]`, `mate[]`. Ниже попытка неверно обобщить двудольный bfs на недвудольный граф:

```
v = q.pop()
for e : v --> x
    if mate[x] == -1: return # нашли путь, молодцы
    if !used[mate[x]]:
        p[x] = v
        used[mate[x]] = 1
        q.push(mate[x])
```

Проблема реализации только в том, что при восстановлении пути по ссылкам `p[]`, мы можем получить непростой путь. Эту проблему можно попробовать костыльно разрешать. Например, проверяя при `p[x] = v`, что `x` нет в пути от `v` до корня, тогда окажется, что тогда мы некоторые пути не найдём, но в простых случаях, как 9.4.1, отработаем корректно. Другие более мощные костыли, пытающие думать про нечётный цикл, спотыкаются о 9.4.3.

Массив `base[]`, DSU «база цикла» – простейший корректный способ решить проблемы с `p[]`.

9.9. Задача про чётный путь

Задача: найти кратчайший простой путь в неорграфе, состоящий из чётного числа рёбер.

Если бы путь был не простым, это просто Дейкстра/Гольдберг в раздвоенном графе.
Если веса отрицательные, это NP-трудно (пусть все веса -1, видим HAM-PATH).
Если граф ориентирован, тоже NP-трудно (можно свестись к «задаче про два пути»).
Простой путь, неор граф, $w_e \geqslant 0$ — интересный случай.

Решение. Ищем путь $a \rightsquigarrow b$ в графе G.
Создадим точную копию G', $\forall v \in G$ проведём ребро $v \to v'$ веса 0. Удалим a и b', заметим, что, если путь-ответ это $a = a_1 \to a_2 \to a_3 \to \cdots \to a_{2k+1} = b$, то min паросочетание на $G \sqcup G'$ это $(a_1,a_2), (a_2', a_3'), (a_3,a_4), \ldots, (a_{2k}', a_{2k+1}')$ и нули между вершинами не из пути.

Если попытаться получить двудольный граф: раздвоить вершины, провести рёбра $a_1 \to b_2$, мы получим «паросочетание = какой-то путь + циклы», в нашем же недвудольном графе путь ищется нужно чётности.

Замечание 9.9.1. Нечётный путь ищется также, для нужной чётности удаляем a и b.

9.10. Литература, полезные ссылки

[e-maxx]. Классическое описание алгоритма Эдмондса и реализации Габова.
[Galil'1986]. Экскурс во всё известное об алгоритмах для паросочетаний на 86-й год. В нашем курсе мы не выходим за рамки этих результатов и даже не затрагиваем самые крутые.
[Обзор'2023]. Статья, содержащая в частности обзор на тему «odd/even path».
[Gabow'2017]. Современная реализация алгоритма за $\mathcal{O}(EV^{1/2})$ на github.

9.11. Исторический экскурс

[Edmonds'1965]. Paths, trees, and flowers. Классический алгоритм за $\mathcal{O}(V^3)$.
[Gabow'1976]. Ph.D. Гарольда Габова'72. Простая и красивая реализация Эдмондса за $\mathcal{O}(V^3)$.

• Попытки обобщить Хопкрофта-Карпа для произвольного графа.

В двудольном графе есть идея за $\mathcal{O}(E)$ находить не один, а сразу «все кратчайшие пути длины d», фаз (разных d) будет $\leqslant\sqrt{V}$ и получится алгоритм Хопкрофта-Карпа за $\mathcal{O}(E\sqrt{V})$.
Идея известна с 1973-го года, с тех пор её старательно обобщали для произвольных графов:

[Hopcroft&Karp'1973]. Паросочетание в двудольном за $\mathcal{O}(E\sqrt{V})$.
[Even&Kariv'1975]. Научились делать одну фазу Хопкрофта-Карпа за $\mathcal{O}(V^2 + E\log V)$. Получили $\mathcal{O}(\min(V^{5/2}, EV^{1/2}\log V))$. Достойный претендент на *ACM Longest Paper Award*. Ивен был научником Харива, который через год оформил эту работу своим Ph.D.
[Micali&Vazirani'1980]. MV80. Научились одну фазу делать за $\mathcal{O}(E)$, получили $\mathcal{O}(EV^{1/2})$.
[Peterson'1988]. Автор утверждает, что смог сделать «понятное изложение» MV80.
[Blum'1990]. Другой подход, тоже $\mathcal{O}(EV^{1/2})$.
[Vazirani'2014]. Ребята наконец оформили строгое доказательство своего мегаалгоритма =)
[Gabow'2017]. Современная реализация алгоритма за $\mathcal{O}(EV^{1/2})$ на github.

Лекция #10: Геометрия

29 мая 2024

[DavidMount'2012]. Большой конспект по всем темам вычислительной геометрии.
[Eppstein'2009]. Связь с графами, увлекательные задачи.
[MotionPlanning]. Видео конспект для получения $\mathcal{O}(n \log^2 n)$ по сабж.
[David Mount'2020]. Триангуляция Делоне за $\mathcal{O}(n \log n)$.
[Скворцов'2002]. Всё и даже больше про триангуляцию Делоне.

10.1. Локализация точки за $[\mathcal{O}(n), \mathcal{O}(\log n)]$

Задача: дан плоский граф, отвечать на online-запросы «в какую грань попала точка?»

Решение. Персистентная вертикальная сканирующая прямая, внутри BST отрезков, пересекающих вертикальную прямую. Ответ на запрос: бинпоиск по x, чтобы найти нужную версию BST и lowerbound от этого BST, чтобы найти ребро грани.

Простейшая реализация даёт $[\mathcal{O}(n \log n), \mathcal{O}(\log n)]$. Чтобы соптимизировать время предподсчёта до $\mathcal{O}(n)$, в качестве BST используем AVL и fat-nodes для персистентности.

10.2. «Выпуклая оболочка» $\Leftrightarrow$ «Пересечение полуплоскостей»

Рассмотрим множество точек (x, y) и множество прямых (k, b): $y = kx + b \Leftrightarrow y - b = kx$.
Полуплоскость задаётся уравнением $y \leqslant kx + b \Leftrightarrow y - b \leqslant kx$.
Заметим, что оба выражения инварианты относительно замены $(x, y) \leftrightarrow (k, -b)$.

Рассмотрим преобразование мира, в котором прямые переходят в точки, а точки в прямые по правилу, описанному выше. Пусть p – точка, l – прямая, f – преобразование.

Тогда $p \in l \Leftrightarrow f(p) \ni f(l)$, и p под $l \Leftrightarrow f(p)$ под $f(l)$, в частности если $l(p_1, p_2)$ – прямая через две точки, то $f(l)$ – точка пересечения прямых $f(p_1)$ и $f(p_2)$.

Задача построения *верхней* огибающей части выпуклой оболочки: даны (x_i, y_i), найти набор прямых (k_j, b_j) на этих точках : $\forall i, j$ точка i под прямой j.

Задача построения *верхней* огибающей части пересечения полуплоскостей: даны полуплоскости (k_j, b_j), найти точки (x_i, y_i) – пересечением прямых (k_j, b_j): $\forall i, j$ точка i под прямой j.

При применении преобразования f задачи и ответы к ним переходят друг в друга.
f – биекция между задачами convexHull и semiplaneIntersection.

Следствие 10.2.1. $\forall$ алгоритм для convexHull подходит для пересечения полуплоскостей (поиска верхней огибающей) $\Rightarrow$ пересечь полуплоскости можно за $\mathcal{O}(n \log k)$, где k – размер ответа.

10.3. Алгоритмы построения выпуклой оболочки

10.3.1. Грэхем, Эндрю и $\mathcal{O}(sort(n))$

Грэхем. Возьмём самую левую-верхнюю точку, она точно лежит на выпуклой оболочке, остальные отсортируем относительно неё по углу и будем их в таком порядке пытаться добавлять в стек-выпуклой-оболочки. При добавлении новой точки сколько-то старых, возможно, выпилят-

ся с вершины стека. Время $\mathcal{O}(n \log n)$ или, точнее, $\mathcal{O}(sort_{\mathbb{R}}(n))$.

Эндрю. Зачем сортировать по углу, когда можно по x. Обозначим самую левую L, самую правую R, отсортируем точки по x, будем идти со стеком из L в R, строя верхнюю половину выпуклой оболочки. Домножим все y-и на -1, пройдём ещё раз, получим нижнюю половину.

Плюсы сортировки по x: если точки целые, мы остались в целых числах; если добавляются новые точки, порядок прежних не меняется. Время $\mathcal{O}(sort_{\mathbb{Z}}(n))$.

10.3.2. Джарвис и $\mathcal{O}(nk)$

«Заворачивание подарка». Возьмём самую левую-верхнюю точку, она точно лежит на выпуклой оболочке. Возьмём луч из неё вверх и будем «заворачиваться». Чтобы выбрать следующую точку на выпуклой оболочке, переберём n кандидатов и выберем того, до кого угол поворот минимален. Время работы $\mathcal{O}(nk)$, где k число вершин выпуклой оболочки.

10.3.3. Чен и $\mathcal{O}(n \log k)$

Пусть мы угадали k, разобьём точки на $\frac{n}{k}$ групп по k произвольным образом, от каждой группы за $\mathcal{O}(k \log k)$ строим оболочку, а теперь начинаем заворачивание: от самой левой точки k раз ищем следующую – или берём следующую в той же оболочке, или по касательной переходим к одной из $\frac{n}{k}$ оболочек. Касательная строится за $\mathcal{O}(\log k)$ чем-то вроде бинпоиска.
Суммарное время работы $\frac{n}{k} \cdot k \log k + k \cdot (1 + \frac{n}{k} \log k) = \mathcal{O}(n \log k)$.

Как угадать k? Перебираем $2 \to 4 \to 16 \to 256 \to \ldots (k \to k^2)$. Внутри перебора,
если заворачивание сделало больше k шагов и не замкнулось, прерываем процесс.
Время работы: $n \log k + n \log(k^{1/2}) + n \log(k^{1/4}) + \cdots = \mathcal{O}(n \log k)$.

10.3.4. Точнее оцениваем Чена

Давайте бить на группы по $k \log k$ точек (или больше). Зачем? Чтобы часть заворачивания работала за $k \cdot \frac{n}{k \log k} \cdot \log k = \mathcal{O}(n)$. За сколько работает часть до заворачивания? За $\frac{n}{k} sort(k)$.
Пусть $sort(k) = k \cdot f(k)$. Подбор k: подгоним рост k так, чтобы $f(k)$ увеличивалась в 2 раза.
Итого $\mathcal{O}(n \cdot f(k))$ на построение выпуклой оболочки от точек с целыми координатами, помещающимися в машинное слово. Например, sort за $\mathcal{O}(k\sqrt{\log \log k}) \Rightarrow$ convexHull за $\mathcal{O}(n\sqrt{\log \log k})$.

10.4. Рандомизированные алгоритмы

Общая идея: давайте добавлять объекты в случайном порядке.
Мы уже использовали её для нахождения пересечения n d-мерных полупространств за $\mathcal{O}(n \cdot d!)$.

10.4.1. Две ближайшие точки за $\mathcal{O}(n)$

Алгоритм 10.4.1. *Пусть уже добавлено i точек, расстояние между ближайшими равно d, и построена структура «разбиение плоскости на клеточки $d \times d$» = хеш-таблица.*
Добавляем $(i+1)$-ую точку, её клетка $= \langle \lfloor \frac{x_{i+1}}{d} \rfloor, \lfloor \frac{y_{i+1}}{d} \rfloor \rangle$, смотрим содержимое этой клетки и 8 соседних. В каждой не более двух точек (иначе нашлись бы две на расстоянии меньше d).

Итого за $\mathcal{O}(1)$ убеждаемся, что или от новой точки все на расстоянии $\geqslant d$, или находим ближайшую к ней, новое d и за $\mathcal{O}(i)$ перестраиваем разбиение на клетки. Точки берутся в случайном порядке $\Rightarrow$ вероятность последнего события $\frac{2}{i+1}$. Время работы: $\sum_i (8 \cdot 2 + i \cdot \frac{2}{i+1}) = \mathcal{O}(n)$.

10.4.2. Минимальный покрывающий точки круг за $\mathcal{O}(n)$

На даны точки A. Мы – процедура `Solve`, которая получает два множества точек: q – те, что точно должны лежать на границе и p – те, что должны лежать внутри или на границе, и возвращает покрывающий круг min радиуса. Изначально запускаем `Solve(`$\varnothing$`,` A`)`.

Алгоритм. $|q| \leqslant 3$. Инициализируем $C =$ описывающая окружность для q (для $|q| = 2$ строим на диаметре, для $|q| = 3$ пересекаем срединные перпендикуляры).
Добавляем точки p в случайном порядке. Если очередная точка $p_i \in C \Rightarrow$ ничего не делаем, иначе p_i точно лежит на границе круга-ответа $\Rightarrow$ запускаем `Solve(`$q + p_i$`,` $\{p_1, \dots p_{i-1}\}$`)`.
Если $|q| = 3$, можно сразу после инициализации вернуть C, по построению $\forall i\ p_i \in C$.

Время работы: $T(n, 3) = \mathcal{O}(1),\ T(n, |q|) = \sum_i \big(1 + Pr_{i,|q|} \cdot T(i, |q|{+}1)\big)$.
$Pr_{i,|q|}$ – вероятность того, что i-ая точка оказалось одной из двух-трёх, образующих ответ.
$Pr_{i,|q|} \leqslant \frac{3-|q|}{i} \Rightarrow T(n, 2) = \Theta(n) \Rightarrow T(n, 1) = \Theta(n) \Rightarrow T(n, 0) = \Theta(n)$.
Для d-мерной сферы мы получили бы время решения $\Theta(n \cdot d!)$.

10.5. Диаграмма Вороного и триангуляция Делоне

10.5.1. Диаграмма Вороного за $\mathcal{O}(n^2)$

Def 10.5.1. *Даны n точек $p_i \in \mathbb{R}^2$ (на плоскости). Определим $A_i = \{q \in \mathbb{R}^2 \mid i = argmin_j |p_j q|\}$. A_i – ячейка i-й точки, часть плоскости для которой, i-ая точка ближайшая из данных. Разбиение плоскости на ячейки – диаграмма Вороного.*

Диаграмма Вороного – планарный граф, каждой из n исходных точек соответствует грань. Рёбра – срединные перпендикуляры между исходными точками. $\forall v\ \deg_v \geqslant 3$. $\deg_v = k \Rightarrow$ среди исходных есть k точек на одной окружности $\Rightarrow$ для точек общего положения $\forall v\ \deg_v = 3$.

Оценим число вершин и рёбер: $E + 2 = V + G, G = n, E \geqslant \frac{3}{2}V \Rightarrow E = \mathcal{O}(n), V = \mathcal{O}(n)$.

• **Алгоритм построения за $\mathcal{O}(n^2 \log n)$.**

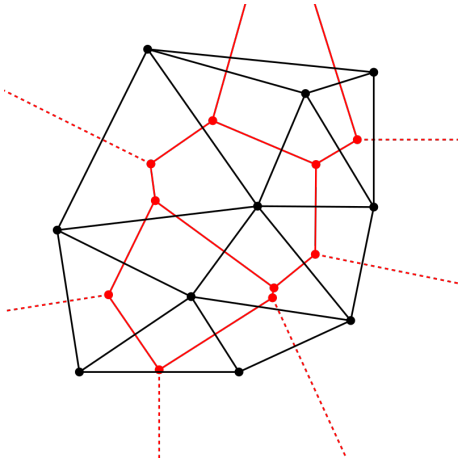
Каждая ячейка Вороного = пересечение n полуплоскостей.

• **Алгоритм построения за $\mathcal{O}(n^2)$.**

Для каждой точки p_i делаем заворачивание подарка: находим ближайшую к ней p_j, начинаем с $q = \frac{1}{2}(p_i + p_j)$ и направления v «срединный перпендикуляр $(p_i p_j)$», заворачиваемся – пересекаем луч (q, v) со всеми срединными перпендикулярами $(p_i p_k)$, выбираем ближайшую к q точку пересечения. При заворачивании каждое ребро мы получили за $\mathcal{O}(n) \Rightarrow$ суммарное время $\mathcal{O}(n^2)$.

10.5.2. Триангуляция Делоне

Даны n точек на плоскости.



Def 10.5.2. *Триангуляция n точек – плоский граф на точках, как на вершинах, в котором каждая грань – треугольник. Триангуляция Делоне обладает дополнительным свойством: описанные вокруг треугольников окружности не содержат внутри ни одной из n точек.*

Автор – Бори́с Никола́евич Делоне́.

Теорема 10.5.3. Триангуляция Делоне – двойственный граф к диаграмме Вороного.

Следствие 10.5.4. Триангуляция Делоне существует.

Следствие 10.5.5. Триангуляция Делоне единственна для точек общего положения (никакие 4 не лежат на одной окружности).

Доказательство. Существование. Диаграмма Вороного – граф G. Рассмотрим двойственный к G граф G^*. Для точек общего положения в G $\forall v$ $\deg_v = 3 \Rightarrow G^*$ – триангуляция. Если в G есть вершина степени k, то в G^* получаем грань-выпуклый-k-угольник, который можно как угодно порезать диагоналями на $k-2$ треугольника, получим G^{**} – триангуляцию. Почему выполнено свойство Делоне? От противного, пусть в окружность треугольника $ABC \ni P \Rightarrow$ для середины одной из сторон, например AB, точка P ближе любой из вершин. Противоречие с вороновостью диаграммы.

Единственность. В другую сторону: двойственный к триангуляции Делоне граф – диаграмма Вороного, а Вороного, очевидно, единственна. Почему двойственный граф – диаграмма Вороного? Рассмотрим ABC и $Q \in ABC$ для которой в двойственном графе ближайшая A, а на самом деле P. От противного, **TODO** ■

Теорема 10.5.6. Свойства Триангуляции Делоне.

1. Максимизирует минимальный угол треугольника.
2. Максимизирует сумму радиусов вписанных окружностей.
3. Расстояние по рёбрам Делоне $\leqslant 1.998\cdot$ евклидово.
4. [wiki] [вики]

Построение триангуляции Делоне за $\mathcal{O}(n^2)$. Построили Вороного, взяли двойственный граф.

10.5.3. Триангуляция Делоне за $\mathcal{O}(n \log n)$

Сразу отметим, что диаграмму Вороного мы автоматически тоже получим за $\mathcal{O}(n \log n)$.

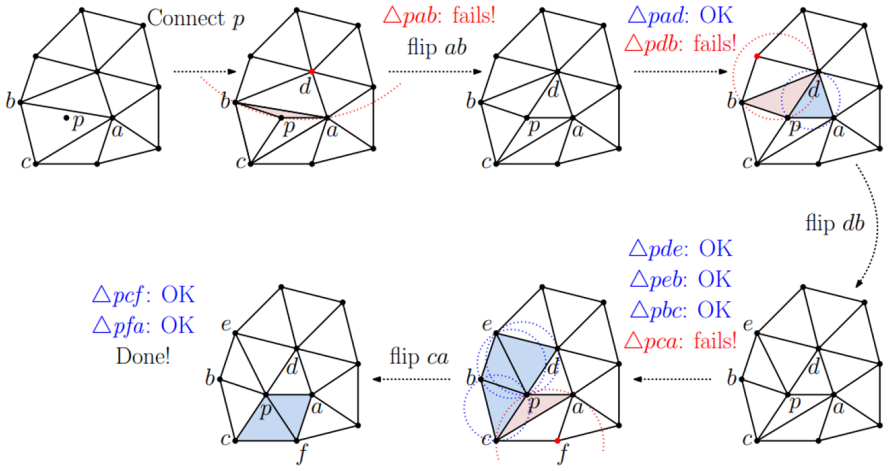
Алгоритм вкратце. Будем строить триангуляцию инкрементально в порядке random-shuffle. Добавление точки P: понять, в каком она $\triangle$-е, разбить его на 3, если нужно, сделать несколько флипов. $\forall \triangle$ поддерживаем «все ещё недобавленные точки в треугольнике». Матожидание перемещений для каждой точки $\mathcal{O}(\log n)$. Матожидание числа флипов на каждом шаге $\mathcal{O}(1)$.

Алгоритм. Добавляем точки по одной в случайном порядке.

При добавлении P хотим знать, в какой $\triangle$ из триангуляции попала P.
Для каждого $\triangle$ храним список $L[\triangle]$ всех ещё не добавленных точек, содержащихся в $\triangle$.

Когда $\triangle$ разбивается, для каждой $Q \in L[\triangle]$ в лоб ищем, в какой новый $\triangle$ попала Q.
$\forall P$ матожидание числа перекладываний P будет $\mathcal{O}(\log n)$.

Пусть точка P попала в $\triangle ABC$, бьём его на $\triangle PAB, \triangle PBC, \triangle PCA$, в каждом делаем проверку. Пример для $\triangle PAB$. Пусть за AB лежит D и $\triangle DAB$. Возможно диагональ PD лучше диагонали AB (D лежит в окружности PAB, а B не лежит в окружности APD), тогда делаем флип, меняем $\triangle PAB, \triangle DAB$ на $\triangle APD, \triangle BPD$, степень P при этом увеличивается. Продолжаем проверки для новых треугольников. Сколько времени потратим? Ровно $\deg P$ в правильной диаграмме. Какая степень P в диаграмме? Матожидание $\mathcal{O}(1)$. Итого флипов $\mathcal{O}(n)$ за всё время, а время на локализацию точек $\mathcal{O}(n \log n)$, эту часть люди умеют оптимизировать.



Почему средняя степень $\mathcal{O}(1)$? Потому что триангуляция – планарный граф, $E \leqslant 3V - 6$, средняя степень $\leqslant 6$, а мы последней добавили случайную из всех вершин графа.

Почему перекидываний точки $\mathcal{O}(\log n)$? Пусть после i-го шага мы перекинули точку, и она теперь живёт в $\triangle ABC$. Такое произошло iff последней мы добавили одну из A, B, C.
Вероятность такого события $= \frac{3}{i} \Rightarrow \mathsf{E}[\text{числа перекладываний}] = \sum_i \frac{3}{i} = \mathcal{O}(\log n)$.

10.6. Задачи

10.6.1. Применения диаграммы Вороного

• MST на плоскости

Задача. Даны n точек на плоскости, нужно найти их MST.
Решение. Рёбер в графе n^2, Краскал будет работать $\mathcal{O}(sort(n^2))$, Прим за $\mathcal{O}(n^2)$. А мы построим диаграмму Вороного и заметим, что рёбра MST обязательно между соседними ячейками (т.е. это рёбра триангуляции Делоне), а их уже $\mathcal{O}(n)$ и Краскал на них отработает за $\mathcal{O}(n \log n)$.

• Для каждой точки найти ближайшую

Аналогично строим Вороного, а ответ для i-й точки – одна из соседних ячеек Вороного $\Rightarrow$ кроме построения диаграммы Вороного мы тратим $\mathcal{O}(n)$ времени.

Решение без Вороного. Спроецируем все точки на случайную прямую, в поисках ближайшей для точки p_i, идём по прямой от неё, пока длина расстояния по прямой $\leqslant$ текущего оптимального расстояния. Работает за $\mathcal{O}(n^{3/2})$ в предположении, что координаты ограничены, для

экспоненциальных от n координат можно построить пример с $\Theta(n^2)$. Плюсы: легко пишется, обобщается на 3D. Минусы: диаграмма Вороного быстрее и может решить чуть более сложную задачу «$\forall a \in A$ найти ближайшую $b \in B$».

10.6.2. k-точек минимального диаметра

Даны n точек на плоскости. Выбрать k точек так: диаметр полученного множества $\to \min$.

Если сделать бинпоиск по ответу x, то, казалось бы, внутри бинпоиска нужно искать k-клику в графе «расстояние $\leqslant x$», что трудно (люди не умеют ни за полином, ни за экспоненту от k, только за n^k). Предположим, что мы угадали два конца диаметра множества-ответа – точки a и b, тогда все остальные точки множества-ответа отрезок ab делит на две доли, и любая другая потенциальная пара c, d: $|cd| > |ab|$ лежит по разные стороны $ab \Rightarrow$ граф c, d: $|cd| > |ab|$ двудольный, и мы можем найти в нём максимальное независимое множество. $\mathcal{O}(n^5)$.

10.6.3. Все пары точек на расстоянии не более R

Делаем, как и в 10.4.1. Бьём плоскость на клеточек $R \times R$.
Для каждой точки рассматриваем содержимое её клетки и 8 соседних.
Если мы в итоге перебрали k точек, то, даже, если в паре с данной нам все не подошли, перебранные образуют $\Theta(k^2)$ пар $\leqslant R \Rightarrow$ всё самортизируется в $\mathcal{O}(n + |ans|)$.

10.7. Сумма Минковского

Def 10.7.1. *Пусть A и B – мн-ва точек $\Rightarrow$ сумма Минковского* $A \oplus B = \{a + b \mid a \in A, b \in B\}$.

Для выпуклых многоугольников можно посчитать за $\mathcal{O}(|A| + |B|)$ двумя указателями: чтобы получить границу суммы, начинаем с самых левых точек, чтобы выбрать, в каком из многоугольников сделать шаг вперёд, смотрим на знак векторного произведение векторов-сторон.

• Задача про погоду

Летит облако A в форме выпуклого многоугольника с постоянной скоростью v. Есть аэропорт B в форме выпуклого многоугольника. Изначально $A \cap B \neq \varnothing$, найдите минимальный момент времени t: облако и аэропорт не пересекаются.

Решение. Считаем сумму Минковского $S = A \oplus (-B)$, $A \cap B \neq \varnothing \Rightarrow (0, 0) \in S$,
проводим из $(0, 0)$ луч в направлении v, пересекаем с границей S.
Аналогично можно найти v: $t(v) = \min$ – выбираем ближайшую к $(0, 0)$ точку границы.

10.8. Motion planning

Общий вид задачи. Мы – некий объект некой формы, который должен в неком пространстве с некими препятствиями (допустимыми состояниями) попасть из точки A в точку B.

Задача, которую сейчас решим. Мы – робот P в форме выпуклого многоугольника, живём на плоскости. Многоугольник P можно сдвигать в любую сторону, поворачивать нельзя. На плоскости есть несколько препятствий в форме невыпуклых многоугольников, препятствия не пересекаются. Нужно (a) найти какой-то путь из точки A в B, (b) найти кратчайший путь.

• Граф видимости

«Простейшее» решение – заметить, что менять направление нужно лишь в состоянии, что p_i (вершина P) совпадает с одной из вершин одного из препятствий $\Rightarrow$ если в P всего k вершин, а

в препятствиях суммарно N вершин, то мы строим граф на $k, \cdot N$ вершинах и каждое из $(k \cdot N)^2$ рёбер проверяем «ничего ли не пересекает» за $\mathcal{O}(k \cdot N)$, итого $\mathcal{O}((k \cdot N)^3)$ на построение графа.

Проверять можно и быстрее. Чтобы провести все рёбра из вершины v, можно пройтись заворачивающим лучом (sweep ray) и обработать события «отрезок препятствия начался», «отрезок препятствия закончился». Итого $\mathcal{O}((k \cdot N)^2 \log kN)$ на построение графа.

• Применяем сумму Минковского

Вместо A_i возьмём препятствие $A_i \oplus (-P)$ и будем считать, что мы – точка. Это отлично работает, если A_i – выпуклый многоугольник, если A_i невыпуклый, разобьём его на выпуклые куски (например, триангулируем). Далее считаем, что препятствия выпуклые, их n штук, в них суммарно N вершин $\Rightarrow$ вершин в графе получилось $N + k \cdot n$ (меньше Nk). При построении рёбер идеи те же, геометрия чуть проще т.к. мы – точка $\Rightarrow$ для проверки одного ребра достаточно проверить пересечение одного отрезка с исходными многоугольниками. $\mathcal{O}(V^2 \log V)$, $V = N + kn$.

• Кратчайший путь

Чем искать кратчайший путь? Идеально подойдёт A^* с ленивым построением графа (вычисляем только рёбра из тех вершин, куда завёл нас A^*). Для проверки себя, можно сдать задачу «Лоцман» [timus:1271] за <0.1 секунды.

• Разделяй и властвуй

Поиск кратчайшего пути мы улучшать не будем. А вот найти хоть какой-то путь можно быстрее. Если $A_i \oplus (-P)$ не пересекаются, то достаточно лишь взять планарный граф из $N + k \cdot n$ вершин, триангулировать его, часть $\triangle$-ов – препятствия, часть $\triangle$-ов – свободное пространство $\Rightarrow$ путь из A в B – $\forall$ путь в графе граней (ломаная строится, например, по центрам граней).

Но $A_i \oplus (-P)$ могут пересекаться. Поскольку сами A_i не пересекались, можно доказать, что суммарно граница объединения $A_i \oplus (-P)$ состоит из не более чем $V + 2V$ вершин, где $V = N + kn$. Заметьте, это не число точек пересечения всех пар $A_i \oplus (-P)$, а именно число вершин на границе множества-объединения. Как построить мега-препятствие (объединение всех препятствий)? Разделяй и властвуй, где merge двух – сканирующая прямая за $\mathcal{O}(V \log V)$, итого $\mathcal{O}(V \log^2 V)$.

10.9. Dynamic Convex Hull

Будем поддерживать верхнюю часть выпуклую оболчку между самой левой и самой правой точками. Поддерживаем точки в порядке, отсортированном по $\langle x, y \rangle$, на этом порядке BST, в каждой вершине выпуклая оболочка от поддерва в форме персистентного BST. Зная детей, посчитать себя – провести общую касательную и вырезать из левого и правого BST нужную часть, делается за $\mathcal{O}(\log n)$, итого $\mathcal{O}(\log^2 n)$ на `add/del` точки.

10.10. Число точек под прямой (в полуплоскости)

[CF]. Старое обсуждение на тему, приводящее к $o(n^{0.5})$ на запрос.

Задача. Даны n точек на плоскости. Хотим сделать предподсчёт и затем в онлайн отвечать на q запросов «сколько точек лежит под прямой $y = ax + by + c$».

Есть два решения. Первая – корневая. Вторая – что-то типа квадродерева.

• Корневая

Пусть все запросы с одним и тем же углом прямой $\langle a, b \rangle$. Тогда все исходные точки мы можем

отсортировать по $ax + by$ и при запросе бинпоиском искать $-c$ в этом массиве. $\mathcal{O}(n \log n)$ на предподсчёт, $\mathcal{O}(\log n)$ на запрос.

Хорошая новость: различных интересных углов не больше n^2 – проведём прямую через каждую пару точек, получим интересное направления. Отсортируем все интересные направления, порядок точек (сортировка по $ax+by$) храним в персистентном BST, каждое событие «изменение порядка» обрабатываем за $\mathcal{O}(\log n)$. Итого предподсчёт за $\mathcal{O}(n^2 \log n)$ и зарос за $\mathcal{O}(\log n + \log n)$ – сперва бинпоиском нашли угол $\langle a, b \rangle$ в массиве всех углов, затем спустились по нужному BST, который задаёт порядок для $\langle a, b \rangle$ в поисках $-c$.

Получили решение за $[\mathcal{O}(n^2 \log n), \mathcal{O}(\log n)]$, можно его сбалансировав, разбив исходные точки на k групп, получится $[\mathcal{O}(\frac{n^2}{k} \log n), \mathcal{O}(k \log n)]$, например, можно взять $k = \sqrt{n}$ и получить $[n\sqrt{n} \log n, \sqrt{n} \log n]$. BST по явному ключу $\Rightarrow$ можно использовать AVL + fat nodes и получить $n\sqrt{n}$ памяти вместо $n\sqrt{n} \log n$ (память – самое узкое место в решении).

• Квадродерево

Обычное квадродерево берёт точки на плоскости и бьёт их рекурсивно на четыре группы – четверти плоскости. Мы сделаем тоже самое, но крестик, который бьёт точки на 4 части, будет с произвольным центром и под произвольным углом. За $\mathcal{O}(n \log C)$ можно найти такое α и две прямые под углами α и $\alpha + \frac{\pi}{2}$, что строго внутри каждой из четвертей $\leqslant \frac{n}{4}$ точек.

Возьмём $line(\alpha)$ прямую, которая делит плоскость на части по $\frac{n}{2}$, $line(\alpha) \times line(\alpha + \frac{\pi}{2})$ задают 4 четверти, в них x_1, x_2, x_3, x_4 точек, при этом $x_1 + x_2 = x_2 + x_3 = x_3 + x_4 = x_4 + x_1 = \frac{n}{2} \Rightarrow x_1 = x_3 \wedge x_2 = x_4$, если $x_1 = x_2$, мы нашли α. Зададим функцию $f(\alpha) = x_1(\alpha) - x_2(\alpha)$, заметим, что f непрерывна и $f(\alpha + \frac{\pi}{2}) = -f(\alpha) \Rightarrow$ есть корень и он ищется бинпоиском. Находить медиану умеем за $\mathcal{O}(n) \Rightarrow$ весь бинпоиск работает за $\mathcal{O}(n \log C)$.

Строим структуру рекурсивно. Получается $\mathcal{O}(n \log n \log C)$ на построение. Ответ на запрос:
$\forall$ прямая пересекает не более трёх четвертей $\Rightarrow$ одна из 4 четвертей или целиком под прямой, или целиком над прямой $\Rightarrow$ делаем три рекурсивных вызова $\Rightarrow T(n) = 3T(\frac{n}{4}) = n^{\log_4 3} = n^{0.7925\ldots}$.

• Объединяем

На нескольких верхних уровнях сделаем «квадродерево», когда размер куска станет $\leqslant k$, сделаем предподсчёт за $\mathcal{O}(k^2 \log k)$. У нас $\frac{n}{k}$ групп, суммарный предподсчёт работает за $\mathcal{O}(nk \log k)$, ответ на запрос за $\left(\frac{n}{k}\right)^{0.7925} \log k$. Для случая $n = q$ можно приравнять $k \log k = \left(\frac{n}{k}\right)^{0.7925} \log k \Rightarrow k^{1+m} = n^m \Rightarrow k = n^{m/(1+m)} = n^{0.4421\ldots}$, итого n запросов мы обработаем за $\mathcal{O}(n^{1.4421} \log n)$.

10.11. Не раскрытые темы

[CG12.9]. Trapezoidal Maps (другой способ локализации точки).
[ppt]. 3D-Convex-Hull за $\mathcal{O}(n \log n)$.
[CF]. 3D-Convex-Hull за $\mathcal{O}(n \log n)$.
[MIT]. 3D-range-query за $\mathcal{O}(\log n)$.
[wiki]. BSP tree (для быстрой отрисовки трёхмерных сцен).

Лекция #11: Планарность

24 апреля 2024

11.1. Основные определения и теоремы

Def 11.1.1. *Планарным (planar) называется граф, который может быть изображён на плоскости без пересечения рёбер.*

Def 11.1.2. *Плоским (plane) граф – изображение графа на плоскости без пересечений рёбер.*

Плоский граф разбивает плоскость на области, которые называются гранями.
Внешней называется грань, содержащая точку на бесконечности.

Теорема 11.1.3. У графа есть укладка на сфере iff граф планарен.

Доказательство. Плоскость гомеоморфна проколотой сфере. Осталось проткнуть сферу. ■

Следствие 11.1.4. Для $\forall$ грани планарного графа есть укладка, в которой именно она внешняя.

Доказательство. Уложим на сфере. Проткнём нужную грань, получилась плоскость. ■

Теорема 11.1.5. Эйлера. Для плоского графа $E + C + 1 = V + G$, где V – число вершин, E – число рёбер, G – число граней, C – число компонент связности.

Доказательство. Индукция: удалим ребро, или увеличилось число граней, или уменьшилось число компонент связности. В конце $C = V$, $G = 1$. ■

Следствие 11.1.6. Для планарных графов $E \leqslant 3V - 6$, $E = \mathcal{O}(V)$.

Доказательство. У каждой грани хотя бы 3 ребра $\Rightarrow E \geqslant \frac{3}{2}G \Rightarrow E + 2 \leqslant V + \frac{2}{3}E$. ■

Упражнение 11.1.7. Покажите, что для двудольных планарных $E \leqslant 2V - 4$.

Lm 11.1.8. Графы $K_{3,3}$ и K_5 не планарны.

Доказательство. В K_5 вершин 5, рёбер $10 > 3{\cdot}5 - 6$. В $K_{3,3}$ вершин 6, рёбер $9 > 2{\cdot}6 - 4$. ■

Def 11.1.9. *Стягивание графа. Процесс, в котором каждая операция – удаление вершины, ребра или стягивание двух вершин, соединённых ребром, в одну.*

Теорема 11.1.10. Вагнер'1937. Граф планарен iff он не стягивается в $K_{3,3}$ или K_5.

Теорема 11.1.11. Куратовский'1930. Граф планарен iff не является подразбиением $K_{3,3}$, K_5.

Подразбиение – процесс обратный стягиванию, так что теоремы равносильны.
Короткое доказательство теоремы от Скопенкова [eng|arxiv] [rus].

Теорема 11.1.12. Фари'1948. $\forall$ планарный граф можно уложить прямыми отрезками.

Теорема 11.1.13. Schnyder'1989. $\forall$ планарный граф при $V \geqslant 3$ можно уложить с использованием только прямых отрезков на гриде размера $(V-2) \times (V-2)$. Укладку можно найти за $\mathcal{O}(V)$.

Теорема 11.1.14. Koebe–Andreev–Thurston. [wiki].
Можно нарисовать вершины непересекающимися кругами: ребро $\Leftrightarrow$ круги касаются.

11.2. Алгоритмы проверки на планарность

11.2.1. Исторический экскурс

[tarjan'1974]. Первый линейный алгоритм дан Тарьяном и Хопкрофтом.

Более современные алгоритмы развивались в направлении уменьшения сложности реализации, сложности доказательства, константы времени работы. Среди них можно выделить работы

[boyer'1999]. Boyer, Myrvold: A simplified $\mathcal{O}(n)$ planar embedding.
[brandes'2010]. Ulrik Brandes: The Left-Right Planarity Test.

11.2.2. Алгоритм Демукрона

Простейший алгоритм за $\mathcal{O}(n^2)$ в худшем, $\mathcal{O}(n \log n)$ в среднем при аккуратной реализации.

1. Выделим любой цикл C, уложим. Топологически есть ровно один способ уложить цикл.

2. После удаления рёбер и вершин C оставшиеся рёбра графа делятся на компоненты связности. Два ребра связны, если у них есть общий конец, не являющийся вершиной C.

3. Построим граф противоречий – для каждых двух компонент можно ли их уложить по одну сторону цикла. Время работы $\sum_{ij}(k_i + k_j) = 2m \sum_i k_i \leqslant 2mE$.
k_i – число вершин, которыми i-я компоненты зацепляется с циклом, m – число компонент.

TODO Картинка

4. Раскрасим граф противоречий в два цвета за $\mathcal{O}(m^2)$.
Если не красится, исходный граф не планарен.

5. Все компоненты укладываем независимо. Если компонента цепляет C не более чем одной вершиной, укладываем независимо от C. Иначе пусть цепляет вершинами a, b, ищем и рисуем любой путь $P\colon a \to b$ (топологически это можно сделать единственным образом). Оставшиеся рёбра компоненты делятся на подкомпоненты относительно P. $C \cup P$ образуют два цикла C_1 и C_2, каждая подкомпонента лежит в одном из них. Переходим к (3), не забываем, что про подкомпоненты, цепляющие $C \setminus P$ заранее известно в C_1 или C_2 они должны лежать.

Время работы: $T(E) = \mathcal{O}(E) + \mathcal{O}(mE) + T(e_1) + \cdots + T(e_m) = \mathcal{O}(E^2) = \mathcal{O}(V^2)$, $e_1 + \cdots + e_m \leqslant E$.

11.3. Алгоритмы отрисовки графа прямыми отрезками

Пусть у нас уже есть укладка графа.

Можно триангулировать граф: в грань из k вершин добавляем или $k - 3$ диагонали, или вершину-центр и k рёбер в центр.

Lm 11.3.1. Если все грани связного планарного графа – треугольники, граф трёхсвязен.
Доказательство. Пусть есть разрез $\{a, b\}$, удалим a и b, получим две компоненты связности G_1, G_2. Уложим графы $G_1 + a + b + (a, b)$ и $G_2 + a + b + (a, b)$ так, чтобы ребро (a, b) лежало на внешней грани. Соединим укладки. Получим внешнюю грань из $\geqslant 4$ вершин. Противоречие. ■

Теорема 11.3.2. Tutte'1963. Spring Theorem. Для $\forall$ трёхсвязного планарного графа.
Зафиксируем любую грань внешней, уложим её в виде выпуклого многоугольника.
Тогда остальная часть графа однозначно укладывается на плоскость таким образом, что каждая грань – выпуклый многоугольник, а любая внутренняя вершина – центр масс соседей.

Алгоритм 11.3.3. *Алгоритм отрисовки графа прямыми отрезками*

0. Вход: список рёбер. Выход: координаты вершин.
1. Укладываем граф Демукроном, получаем грани.
2. Триангулируем граф: в каждую грань добавляем центр и рёбра в центр.
3. Выбираем ∀ грань-△, называем внешней, укладываем как △ (-1,-1), (1,-1), (0,2).
4. Записываем систему уравнений из 11.3.2.
5. Устанавливаем координаты всех остальных вершин $(0,0)$, решаем систему методом итераций.
6. Удаляем лишние рёбра, добавленные в (2).

Замечание 11.3.4. Если изначально граф уже трёхсвязен, можно пропустить (1), (2), (6). Внешней гранью тогда можно выбрать, например, кратчайший цикл.

11.4. Планарный сепаратор

[lipton'1977]. Почти сразу после успешной проверки на планарность и укладки графа Тарьян и Липтон показали, что в ∀ планарном графе за $\mathcal{O}(n)$ можно найти сепаратор размера $f(n)=2\sqrt{2}\sqrt{n}$.

Def 11.4.1. *S – сепаратор графа $G=\langle V,E\rangle$, если*
$V = A \sqcup B \sqcup S$: $\max(|A|,|B|) \leqslant \frac{2}{3}n$, $n=|V|$, *между A и B нет рёбер.*

Lm 11.4.2. Если при удалении S все компоненты связности имеют размер $\leqslant \frac{2}{3}n$, S – сепаратор.
Доказательство. Жадно набираем множества A и B. Просматриваем компоненты от большей к меньшей, кидаем очередную компоненту в меньшее из A, B. ■

Lm 11.4.3. В планарном графе есть остов высоты $h \Rightarrow$ есть A: $|A| \leqslant 2h$, $A \cup \{root\}$ – сепаратор.
Доказательство. Триангулируем граф.
Каждое ребро e не из остова образует цикл, который делит множество вершин на A «внутри», B «снаружи». Выберем $e(u,v)$: $\max(|A|,|B|) \to \min = m$. Пусть $m > \frac{2}{3}n$. Пусть большая часть внутри цикла. Ребро e смежно двум граням-треугольникам. Возьмём тот, что внутри цикла, пусть его третья вершина x. Рассмотрим случаи – какие из рёбер (u,x), (v,x) лежат на цикле. В самом интересном «оба не лежат» покажем, что у одного из рёбер (u,x), (v,x) величина $\max(|A|,|B|)$ меньше. Противоречие. ■

Теорема 11.4.4. Тарьян-Липтон. В ∀ планарном графе G ∃ сепаратор S: $|S| \leqslant 2\sqrt{2}\sqrt{n}$.
Доказательство. Запустим bfs из любой вершины v, получим слои по расстоянию до v – $L_0, L_1, \ldots, L_d$. Найдём i: $|L_0 \cup \cdots \cup L_{i-1}| \leqslant \frac{n}{2}$, $|L_0 \cup \cdots \cup L_i| > \frac{n}{2}$.
Если бы $|L_i| \leqslant f(n)$, счастье уже наступило бы. Но нет ⇒

Ищем $j_0 < i$: $|L_{j_0}| + 2(i - j_0 - 1) \leqslant 2\sqrt{k}$, где $k = |L_0| + \cdots + |L_{i-1}|$.
Пусть $\nexists j_0 \Rightarrow |L_{i-1}| > \sqrt{k}$, $|L_{i-2}| > \sqrt{k} - 2, \ldots \Rightarrow |L_0| + \cdots + |L_{i-1}| > k$. Противоречие.
Аналогично ищем $j_1 \geqslant i$: $|L_{j_1}| + 2(j_1 - i) \leqslant 2\sqrt{n-k}$.
Пусть $\nexists j_1 \Rightarrow |L_i| > \sqrt{n-k}$, $|L_{i+1}| > \sqrt{n-k} - 2, \ldots \Rightarrow |L_i| + \cdots + |L_n| > n-k$. Противоречие.
Максимум величины $|L_{j_0}| + |L_{j_1}| + 2(j_1 - j_0 + 1)$ достигается при $k = \frac{n}{2}$ и равен $2\sqrt{2}\sqrt{n}$.

Обозначим $L_a \cup L_{a+1} \cup \cdots \cup L_b$, как $L[a,b]$.
Множество $S_1 = L_{j_0} \sqcup L_{j_1}$ делит граф на 3 части: $A = L[0, j_0 - 1]$, $B = L[j_0 + 1, j_1 - 1]$, $C = L[j_1 + 1, n]$.
$|A|, |C| \leqslant \frac{n}{2} \Rightarrow$ если $|B| \leqslant \frac{2n}{3}$, вернём сепаратор S_1.
Иначе рассмотрим граф $G' = L[j_0, j_1 - 1]/_{L_{j_0}}$ (стянули L_{j_0} в одну вершину).

В G' остовное дерево bfs имеет высоту $j_1 - j_0 \overset{11.4.3}{\Rightarrow} \exists$ сепаратор $X \cup \{root\} = X \cup L_{j_0}$: $|X| \leqslant 2(j_1 - j_0)$.
Итого $|(L_{j_0} \sqcup L_{j_1}) \cup X| \leqslant 2\sqrt{2}\sqrt{n}$ и является планарным сепаратором. ■

Замечание 11.4.5. В доказательстве теоремы планарность мы пользуемся только в лемме 11.4.3.

Замечание 11.4.6. [holzer'2009]. В журнале по экспериментальным алгоритмам говорят, что если оставить только последнюю часть алгоритм «*взять циклы, образованные не древесными рёбрами триангуляции*», то лучший из рассмотренных циклов на большинстве графов даст оценки лучше чем в теореме.

11.4.1. Решение NP-трудных задач на планарных графах

Lm 11.4.7. В $\forall$ планарном графе есть множество вершин S размера $\Theta(\sqrt{n})$:
$V = A \sqcup B \sqcup S$, $\max(|A|,|B|) \leqslant \frac{n}{2}$, между A и B нет рёбер.

Доказательство. У нас есть две корзины $A, B = \varnothing$ и множество $X = V$. В каждый момент времени $\max(|A|,|B|) \leqslant \frac{n}{2}$. Пока $\min(|A|,|B|) + |X| > \frac{n}{2}$ делим X его сепаратором на Y и Z, к меньшему из A и B добавляем меньшее из Y и Z, большее из Y и Z кладём в X.

Оценим суммарный размер сепараторов: $2\sqrt{2}(\sqrt{n} + \sqrt{\frac{2}{3}n} + \sqrt{\frac{4}{9}n} + \dots) = \frac{2\sqrt{2}}{1-\sqrt{2/3}}\sqrt{n}$. ■

• Поиск max IS в планарном графе за $2^{\Theta(n)}$.

Разделяй и властвуй. Выделим по лемме $(\frac{n}{2}, \frac{n}{2})$-сепаратор S.
Переберём $2^{|S|}$ вариантов, какие вершины сепаратора лежат в независимом множестве.
Для каждого из $2^{|S|}$ вариантов сделаем два рекурсивных вызова от половин.
Итого $T(n) \leqslant 2{\cdot}2^{C\sqrt{n}}{\cdot}T(n/2) \leqslant 2{\cdot}2^{C\sqrt{n}}{\cdot}2{\cdot}2^{C\sqrt{n/2}}{\cdot}2{\cdot}2^{C\sqrt{n/4}}{\cdot}\ldots = 2^{\Theta(\sqrt{n})}$.

• Раскраска в k цветов планарного графа за $2^{\Theta(\sqrt{n}\log k)}$.

Аналогично: переберём все $k^{|S|}$ раскрасок вершин сепаратора.
Итого: $T(n) \leqslant 2{\cdot}k^{C\sqrt{n}}{\cdot}T(n/2) = 2^{\Theta(\log k\sqrt{n})}T(n/2) = 2^{\Theta(\log k\sqrt{n})}$.

Замечание 11.4.8. Научившись искать сепаратор меньше, мы ускорим решение многих NP-трудных задач. К сожалению, для планарных графов оценка $\Theta(\sqrt{n})$ достигается: грид $\sqrt{n}\times\sqrt{n}$.

11.5. Системы уравнений

Вспомним 11.3.3 СЛАУ для укладки графа прямыми отрезками. В ней у нас есть две независимых СЛАУ – по x, по y. Каждая из них «СЛАУ на графе», переменные – значения вершин графа, ненулевые элементы матрицы – рёбра графа. Ниже в разделе «правило Кирхгофа» мы приведём ещё один пример «СЛАУ на графе». В обеих задачах граф планарный.

В этом разделе мы обсудим, как СЛАУ можно решать быстрее, чем Гауссом за $\mathcal{O}(n^3)$.

Во-первых, для $\mathbb{F}_p$ есть алгоритм Видемана за $\mathcal{O}(nA)$, где A – число ненулевых элементов в матрице. Для «СЛАУ на планарных графах» это автоматически даёт $\mathcal{O}(n^2)$ т.к. рёбер $\mathcal{O}(n)$.

Во-вторых, есть метод итераций. Опять вспомним 11.3.3, наши уравнения имеют вид $x_i = \frac{1}{k}\sum x_j$ (среднее арифметическое соседей). Метод итераций заключается в том, чтобы начать с любого приближения решения (например, все вершины в центре внешней грани). Далее одна итерация: пересчитать x_i по данным формулам (получить следующее приближение). Одна итерация работает за $E = \mathcal{O}(n)$. Сколько делать итераций? Заранее неизвестно. В целом метод может сходиться экспоненциально медленно, но для планарных графов эмпирически хорош.

Иногда как в разделе ниже «k-диагональная матрица» Гаусс автоматически работает сильно быстрее, если применять его как обычно «слева направо, сверху вниз».

А далее в разделе «nested dissection» мы покажем, как для планарного графа получить время $\mathcal{O}(n^{3/2})$ и память $\mathcal{O}(n\log n)$, используя наличие сепаратора размера $\Theta(\sqrt{n})$.

11.5.1. k-диагональная матрица

k-диагональная – матрица, состоящая из главной диагонали и нескольких соседних, всего k.

TODO

Если запустить обычного Гаусса, учитывающего, что строки – отрезки длины не более $2k$, автоматически получится решение за $\mathcal{O}(k^2 n)$. В частности метод для трёхдиагональной матрицы делает то же и работает за $\mathcal{O}(n)$.

Аналогично для верхнетреугольной матрицы + k диагоналей Гаусс будет работать за $\mathcal{O}(kn^2)$.

Если матрица состоит из побочных, а не главных диагоналей, нужно развернуть порядок строк, чтобы свести задачу к уже решённой.

11.5.2. Правило Кирхгофа

Рассмотрим задачу нахождение сопротивления между вершинами a и b в неорграфе, где у каждого ребра e есть сопротивление R_e.

Для всех вершин v обозначим потенциал вершины $u[v]$. Пусть $u[a] = 0$, $u[b] = 1$. Сила тока $I_{e\colon a\to b} = \frac{u[b]-u[a]}{R_e}$. Правило Кирхгофа гласит «*в каждую вершину втекает тока ровно столько, сколько вытекает*», т.е. $\forall v \neq a, b \sum\limits_{e\colon v\to x} I_e = \sum\limits_{e\colon v\to x} \frac{u[v]-u[x]}{R_e} = 0 \Leftrightarrow \left(\sum_e \frac{1}{R_e}\right) u[v] = \sum_e \frac{1}{R_e} u[x]$.

Получили систему линейных уравнений над $u[v]$. Её можно решить Гауссом за $\mathcal{O}(V^3)$.
Если схема задаётся планарном графом, то есть сепаратор размера $\mathcal{O}(V^{1/2}) \Rightarrow$
систему можно решить за $\mathcal{O}(V^{3/2})$ используя nested dissection. Подробно в [tarjan'1986].

11.5.3. Nested dissection

Наша система – симметричная матрица, она же матрица смежности взвешенного графа. Попытаемся элиминировать переменную-вершину x_i, с точки зрения матрицы мы повычитаем i-й столбец и строку так, чтобы $\forall j \neq i\ x_{ij} = x_{ji} = 0, x_{ii} = 1$. С точки зрения графа, мы удалим вершину x_i и попарно соединим рёбрами всех соседей x_i. Важно выбрать порядок элиминации переменных, чтобы $Space, Time \to \min$.

Алгоритм: выделим сепаратор, вершины сепаратора поставим в конец порядка элиминации, от компонент сделаем рекурсивные вызовы. *Утверждение:* $\sum$ время всех элиминаций $\mathcal{O}(n^{3/2})$, суммарное число ненулевых ячеек в матрице за всё время, т.е. требуемая память, $\mathcal{O}(n \log n)$.

11.6. Выделение граней плоского графа

• **Задача**

Даны координаты вершин плоского графа, уложенного прямыми отрезками.
Выделить грани графа.

• **Решение за $\mathcal{O}(sort + V)$**

Для каждого ребра $e(a, b)$ создадим две ориентированные копии $e_1 \colon a \to b,\ e_2 \colon b \to a$.
Проставим $rev[e_1] = e_2, rev[e_2] = e_1$. Добавим рёбра в списки смежности $g[a].add(e_1)$, $g[b].add(e_2)$.
Для каждой вершины v отсортируем $g[v]$ по углу. Сохраним позиции рёбер в списках смежности:

```
for v=1..V:
    for i=0..g[v].size-1:
        index[g[v][i]] = i
def next(e): # e: a -> b
    retrun g[a[e]][(index[e] + 1) % g[a[e]].size]
```

Тогда для каждого ориентированного ребра e следующим в грани будет $f_e = next(rev[e])$.
Граф $e \to f_e$ – ровно набор ориентированных циклов, осталось их выделить.

Замечание 11.6.1. Все грани кроме внешней будут ориентированы по часовой стрелке.

11.7. Поток в планарном графе

Величину max потока и min разреза между вершинами s и t одной грани легко найти. Берём такую укладку, что s и t на внешней грани, строим граф, где грани – вершины, ищем Дейкстрой путь-разрез между s и t. Время $\mathcal{O}(n + \texttt{Dijkstra}(n)) = \mathcal{O}(n \log n)$.

11.8. Литература

[Luca Vismara | graphbook]. Подробно про отрисовку графов.
[thomassen'2004]. Доказательство пружинной теоремы Татта, её физическое понимание.
[tarjan'1986]. Гаусс для планарных графов за $n^{3/2}$ и не только.
[tarjan'1977]. Существование и построение за $\mathcal{O}(n)$ планарного сепаратора.
[skopenkov]. Доказательство теоремы Куратовского.
[boyer'1999]. Короткий (2.5 страниц в 2 столбца) алгоритм проверки на планарность за $\mathcal{O}(n)$.
[Gauss'93]. Обобщение жадностей в Gauss Elimination для произвольных графов.