Алгосы, часть ii

Денис Осипов, Иван Ермошин, Егор Нечаев

23 августа 2020 г.

Введение

Этот проект – коллективный **конспект** по второй части курса «Математические основы алгоритмов», впервые прочитанного первокурсникам МКН СПбГУ в первой половине ii семестра 2020 года Эдуардом Алексеевичем Гиршем.

Актуальные исходники: `https://www.overleaf.com/read/hnbkrkyknbpk` и `https://github.com/gogochushij/algosi-hirsch`

Если вы хотите **принять участие** в написании билетов, или же **сообщить об ошибке**, напишите `http://vk.com/gogochushij`. Предполагается, что каждый автор напишет около 4 билетов, но мы будем рады любой посильной помощи. Здесь можно посмотреть, с какими билетами вы можете помочь проекту.

Последнее обновление публичной версии конспекта: 23 августа 2020 г.

Содержание

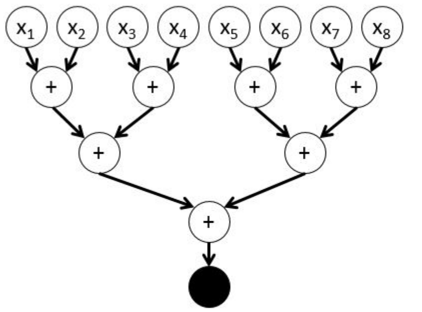
1 (1) Параллельные алгоритмы – i (Осипов Д.)

Параллельный алгоритм – предназначенный для исполнения на нескольких процессах.

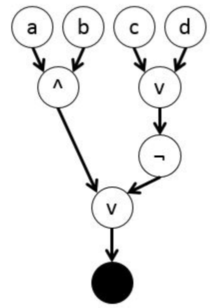
1.1 Булевы схемы как модель параллельного алгоритма

Определение. ***Булева схема*** *– ориентированный граф без циклов, где:*

- *вершины без входящих ребер соответствуют входным данным,*
- *вершины с входящими ребрами («гейты») соответствуют процессорам, которые выполняют операцию с данными, поступающими в вершину по входящим ребрам,*
- *вершины без выходящих ребер соответствуют выходным данным.*



Сумма массива «бинарным деревом»
за O(log n)



Вычисление формулы (a ^ b) v ¬(c v d)
за три шага

«Высота» схемы, т.е. длина наибольшего пути от вершины выходных данных – количество параллельных шагов алгоритма. Еще можно заметить, что если каждому гейту присвоить число – «номер этажа» так, что каждый переход осуществляется с «верхнего» этажа на «нижний», то максимальное число процессоров на одном этаже – достаточное количество процессоров для исполнения всего алгоритма. Так, в левом примере достаточно взять четыре процессора, а в правом – два.

Суммирование массива за $O(\log n)$ параллельных шагов в примере слева – уже хороший пример параллельного алгоритма. Хотя он интуитивно понятен, опишем его формально.

Задача. *Пусть дано n чисел. Вычислить их сумму.*

Решение за $O(\log n)$. Считаем, что n – степень двойки (если нет, дополним нулями). Разобьем все числа на $n/2$ пар и поручим каждому процессору одну пару, чтобы он вычислил ее сумму. Получившиеся $n/2$ чисел разобьем на $n/4$ пар и так же вычислим суммы этих пар. Повторяем до тех пор, пока не останется одно число. Ясно, что всего будет выполнено $\log_2 n = O(\log n)$ параллельных шагов. □

NB: *Сложить*[1] *n чисел быстрее, чем за $\log n$ шагов, нельзя.* В самом деле, если можно, то булева схема такого алгоритма как граф-дерево имеет высоту $h \leq \log_2 n - 1$. Но каждый гейт принимает на вход не больше двух чисел, т.е. входная степень каждой вершины не больше 2. Значит верхних входных гейтов не может быть более $2^h \leq n/2$ чисел, а надо n.

При проектировании параллельных алгоритмов в качестве меры их эффективности возникает аж три параметра: *количество параллельных шагов (время работы)*, *количество используемых процессоров* и *общая работа* (определение дано далее). К счастью, об одном из них – количестве процессоров – можно не задумываться, о чем говорит нам следующее утверждение.

[1] Имея в распоряжении только «+»-гейты, принимающие ровно два числа

1.2 Принцип Брента

Теорема. ***принцип Брента*** *Рассмотрим параллельный алгоритм, выполняющий t параллельных шагов, где на $i-$том шаге задействовано w_i процессоров (т.е. выполняется w_i операций). Обозначим $W = \sum\limits_{i=1}^{t} w_i$ и назовем эту величину* <u>*общей работой алгоритма*</u>. *Тогда алгоритм можно перепрограммировать так, чтобы на P процессорах он работал не более, чем за $\frac{W}{P} + t$ параллельных шагов.*

Доказательство. Перераспределим все W операций на P процессорах наиболее равномерно. Тогда i-тый шаг изначального алгоритма можно выполнить за $\lceil \frac{w_i}{P} \rceil$ новых шагов. Оценим общее число шагов нового алгоритма:

$$t' = \sum_{i=1}^{t} \left\lceil \frac{w_i}{P} \right\rceil \leq \sum_{i=1}^{t} \left(\frac{w_i}{P} + 1 \right) = \sum_{i=1}^{t} \frac{w_i}{P} + t = \frac{W}{P} + t$$

Таким образом, получили алгоритм с искомым временем работы. □

NB: Принцип Брента позволяет при проектировании параллельных алгоритмов **не думать**, на скольких процессорах будет работать алгоритм. Именно: пусть был создан алгоритм, работающий на неизвестном (лень считать) числе процессоров $P_0(n)$ и совершающий общую работу $W(n)$ за $t(n)$ параллельных шагов. Тогда его можно перепроектировать на любое число процессоров $P(n)$ такое, что

$$\frac{W(n)}{P(n)} = O(t(n)),$$

и асимптотически не потерять во времени, так как тогда новое время работы все еще $t'(n) \leq \frac{W(n)}{P(n)} + t(n) = O(t(n))$. Поэтому в дальнейшем при изучении параллельных алгоритмов считаем, что у нас **сколь угодно много процессоров**, а затем количество нужных процессоров будем вычислять по принципу Брента.

1.3 Параллельное умножение булевых матриц

NB: *булевые – только чтобы арифметика с числами была $O(1)$.*

Задача. *Даны две матрицы A и B размера $n \times n$ над $\mathbb{F}_2$. Вычислить их произведение, то есть числа $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$ для всех $i, j = 1..n$ (всего n^2 чисел)*

Непараллельное решение. Вычислить все n^2 чисел C_{ij}, каждое считается за $O(n)$, значит общая сложность $O(n^3)$. □

Несмотря на то, что в прошлом разделе мы условились не думать о количестве процессоров, конкретно здесь на всякий случай приведем два решения. Второе решение – просто пример того, как работает принцип Брента.

Решение за $O(\log n)$ времени на n^3 процессорах. Занумеруем все n^3 процессоров тройками чисел (i, k, j), где $i, k, j = 1..n$. Сначала на каждом процессоре (i, k, j) посчитаем $A_{ik}B_{kj}$. Теперь хотим получить число $C_{ij} = \sum_{k=1}^{n}(i, k, j)$, Сделаем это за $\log n$ шагов (суммирование бинарным деревом). Задача решена за $1 + \log n = O(\log n)$ шагов на n^3 процессорах. □

<u>*Вместо второго решения*</u>*, можно, наверное, просто привести первое решение, а затем вычислить оптимально возможное количество процессоров, на которое это решение можно перепроектировать.* Общая работа этого решения $W(n) = O(n^3)$. Действительно, ведь данное решение просто считает n^2 сумм $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$, переставив слагаемые в другом порядке, таким образом, n^2 раз совершено n действий сложения. Тогда количество процессоров P вычисляется из $\frac{W(n)}{P(n)} = O(t(n)) \implies P(n) = \frac{W(n)}{t(n)} = O\left(\frac{n^3}{\log n}\right)$

Решение за $O(\log n)$ ***времени на*** $O\left(\frac{n^3}{\log n}\right)$ ***процессорах.*** Модифицируем алгоритм выше. На первом шаге вычислить все числа $A_{ik}B_{kj}$ получится не за 1, а за $O(\log n)$ шагов: за каждый шаг просто посчитаются очередные $\frac{n^3}{\log n}$ чисел $A_{ik}B_{kj}$. Получать из них C_{ij} за $O(\log n)$ мы уже умеем. Итоговая сложность $O(\log n) + O(\log n) = O(\log n)$. □

1.4 Параллельная достижимость в графе

Задача. *Дан граф, заданный матрицей смежности* $\{a_{ij}\}$*. Построить его матрицу достижимости.*

Решение за $O(\log^2 n)$ ***времени*** Будем булево умножать матрицы: вместо $\cdot$ возьмем $\wedge$, вместо $+$ возьмем $\vee$. Из формулы перемножения матриц несложно видеть, что A^k – матрица $k-$шаговой достижимости. Тогда матрица достижимости – любая матрица A^k, где $k \geq n$. Умеем возводить матрицу в квадрат за $O(\log n)$. Для получения матрицы достижимости A^n. возведем матрицу A в квадрат $\log n$ раз. Итоговая сложность $O(\log n) \cdot O(\log n) = O(\log^2 n)$. Общая работа W(n) $= O(n^3 \log n)$, так как $\log n$ раз перемножили матрицы за $O(n^3)$ работы. Число процессоров: $P(n) = O\left(\frac{n^3 \log n}{\log^2 n}\right) = O\left(\frac{n^3}{\log n}\right)$. □

2 (2) Параллельные алгоритмы – ii (Осипов Д.)

2.1 Параллельное вычисление всех префиксных сумм

Задача. *Дан массив* $A[0 \ldots n-1]$*. Вычислить все его префиксные суммы*

Решение за $O(\log n)$ Рекурсивный алгоритм. Предположим, что умеем считать префиксные суммы массивов меньшего размера.

Заведем вспомогательный массив $B[0..\frac{n}{2}-1]$, в котором положим $B[i] = A[2i] + A[2i+1]$ (один параллельный шаг). Посчитаем (по предположению) все префиксные суммы B и заменим ими сам массив B. После этой операции для всякого $0 \leq k \leq \frac{n}{2} - 1$ верно $B[k] = \sum_{j=0}^{2k+1} A[j]$.

Теперь делаем на месте A массив префиксных сумм A следующим образом. Если $i > 0$ четное, то полагаем $A[i] = B[\frac{i}{2}-1] + A[i]$ (проверьте подстановкой, что $= \sum_{j=0}^{i} A[j]$). Если же $i > 0$ нечетное, то просто полагаем $A[i] = B[\frac{i-1}{2}]$ (снова проверьте, что $= \sum_{j=0}^{i} A[j]$). Эта операция – снова один параллельный шаг. Таким образом, на месте массива A был построен массив префиксных сумм A. Описание алгоритма закончено.

Соответствующий псевдокод:

```
PrefixSum(&A[0..n-1]):
if n > 1 then
    B = [0] * (n/2 - 1)
    parallel for i = 0..(n/2 - 1 do
        B[i] = A[2i] + A[2i+1]
    PrefixSum(B)

    parallel for i = 0..(n-1) do
        if i > 0 then
            A[i] = B[i/2 - 1] + A[i]
        else
            A[i] = B[(i-1)/2]
```

Время работы оценивается просто: из кода следует соотношение $T(n) = T(n/2) + C$, и далее можно написать $= T(n/4) + 2C = T(n/8) + 3C = \ldots = C \cdot \log n = O(\log n)$. Общая работа: $W(n) = W(n/2) + O(n)$, откуда по мастер-теореме $W(n) = O(n)$. По принципу Брента количество процессоров можно взять $P(n) = \frac{W(n)}{T(n)} = O(\frac{n}{\log n})$. $\square$

2.2 Параллельное сложение чисел

Задача. *Даны два (длинных) двоичных числа в виде* $a = \sum_{i=0}^{n} a_i 2^i$ *и* $b = \sum_{i=0}^{n} b_i 2^i$. *Вычислить их сумму в виде* $c = \sum_{i=0}^{n} c_i 2^i$.

Решение за $O(?????)$**.** Для удобства считаем $a_n = b_n = 0$, остальные $a_i, b_i = 0$ или 1

Формализуем алгоритм сложения столбиком. Через z_i обозначим число (0 или 1), которое при сложении столбиком переносится из $i-$того разряда в $(i+1)-$тый. Если бы мы знали все переносы z_i, то c_i можно бы было вычислить по формуле[2] $c_i = (a_i + b_i + z_{i-1})\%2$.

Положим:

- $g_i = a_i \wedge b_i$ – «генератор переноса»,
- $p_i = a_i \vee b_i$ – «продолжатор переноса».

Перебирая все возможные случаи, когда в $i-$том разряде может возникнуть перенос, получаем формулу для z_i (опустим знак $\wedge$ для наглядности):

$$z_i = g_i \vee p_i z_{i-1}$$

Обратите внимание, что это похоже на «линейную рекурренту» на z_i. Распишем дальше z_{i-1}:

$$z_i = g_i \vee p_i(g_{i-1} \vee p_{i-1} z_{i-2})$$

$$= g_i \vee p_i g_{i-1} \vee p_i p_{i-1} z_{i-2}$$

Итак, при одной «итерации» «свободный член» g_i заменился на $g_i \vee p_i g_{i-1}$, а «коэффициент» p_i – на $p_i p_{i-1}$. Определим операцию на парах битов:

$$(a, b) \odot (a', b') = (a' \vee b'a,\ b'b)$$

Поверим **(нужно уметь проверять!)**, что эта операция ассоциативна. Тогда если умеем вычислять вектора

$$(v_{k1}, v_{k2}) = (0, 0) \odot (g_1, p_1) \odot (g_2, p_2) \odot \cdots \odot (g_k, p_k) \text{ для всех } k = 1..n,$$

то имеем $z_k = v_{k1} \vee v_{k2} z_{-1} = v_{k1}$. Но вектора (v_{k1}, v_{k2}) суть просто префиксные «суммы» последовательности $(0, 0), (g_1, p_1), \ldots, (g_n, p_n)$ относительно ассоциативной операции $\odot$. К ним применим алгоритм нахождения префиксных сумм за $O(\log n)$ выше.

Время и работа алгоритма: сначала посчитали g_i и p_i за время $O(1)$ и работу $O(n)$, потом префиксы за $O(\log n)$ и работу $O(n)$, наконец вычислили z_i и c_i за время $O(1)$ и работу $O(n)$. Итоговое время $O(\log n)$, итоговая работа $O(n)$, процессоров $O(\frac{n}{\log n})$. $\square$

2.3 Параллельное умножение чисел

Задача. *Даны два (длинных) двоичных числа в виде* $a = \sum_{i=0}^{n} a_i 2^i$ *и* $b = \sum_{i=0}^{n} b_i 2^i$. *Вычислить их произведение в виде* $c = \sum_{i=0}^{2n} c_i 2^i$.

[2] Полагая $z_{-1} = 0$ по определению

Решение за $O(\log n)$. Ясно, что $ab = \sum\limits_{i=0}^{n} ab_i 2^i = \sum\limits_{i\colon b_i=1} a2^i$. Таким образом мы свели умножение двух чисел с сложению не более чем n чисел. Но на этом не все.

Трюк «Два по цене трёх». Пусть нам даны три числа x, y, z. Как за $O(1)$ времени сделать из них два числа с той же суммой? Для каждого i число $x_i + y_i + z_i$ есть некоторое двубитовое число $2p_i + q_i$. Составим числа p, q из таких p_i, q_i. Тогда верно $x+y+z = 2p+q$. Итак, мы свели сложение трех чисел к сложению двух чисел за $O(1)$ времени[3] и $O(n)$ работы.

Итак, как быстро складывать много чисел? Разбиваем их на тройки (возможные лишние 1-2 числа игнорируем), применяем к каждой тройке трюк. Делаем так, пока не останется одно или два числа (в последнем случае просто сложим их).

Оценим время и работу. Один трюк требует $O(1)$ времени и $O(n)$ работы. На каждом параллельном шаге трюк применяется $\sim n/3 = O(n)$ раз, т.е. общая работа на одном параллельном шаге $O(n^2)$. На каждом шаге количество чисел уменьшается в 3/2 раза, откуда $T(n) = T\left(\frac{n}{3/2}\right) + O(1)$ и $W(n) = W(\frac{n}{3/2}) + O(n^2)$. По мастер-теореме получаем $T(n) = O(\log_{3/2} n) = O(\log n)$ и $W(n) = O(n^2)$.[4] Процессоров можно брать $O\left(\frac{n^2}{\log n}\right)$. $\square$

3 (3) Параллельные алгоритмы – iii (Осипов Д., Нечаев Е.)

3.1 (В РАЗРАБОТКЕ) Параллельное вычисление всех расстояний до конца списка

Задача. *Дан список $a_1, \ldots, a_n$ в следующем формате. Про каждый элемент a_i известно, какой элемент за ним следует. Обозначим его номер за* `next[i]`. *Если за элементом ничего не следует, считаем* `next[i] == nil`. *Предположим, что указатели* `next[i]` *действительно образуют список. Найти расстояние до конца списка для каждого элемента.*

У нее есть решение за $O(\log n)$.

Каждому элементу a_i сопоставим процессор p_i. Заведем массив d_i, проинициализируем его следующим образом. На первом параллельном шаге для концевого i ($next[i] ==$ `nil`) положим $d_i = 0$, для всех остальных положим $d_i = 1$. В дальнейшем указатели будут изменяться (таким образом, структура списка будет нарушаться), и тогда d_i будет означать расстояние между a_i и $a_{next[i]}$ в исходном списке.

Далее на каждом параллельном шаге происходит пересчет расстояний. Именно, каждый процессор `i`, для которого `next[i]` $\neq$ `nil`, делает следующее (порядок важен!): запоминает $d[$`next[i]`$]$, затем увеличивает $d[i]$ на запомненное значение. После этого (снова порядок важен!) процессор `i` запоминает `next[next[i]]`, затем присваивает это значение к `next[i]`. Алгоритм останавливается, когда все `next[i] == nil`.

Алгоритм корректно находит ответ. Действительно, только что описанный цикл сохраняет инвариант «d_i – расстояние между a_i и $a_{next[i]}$ в исходном списке», а в конце алгоритма все $next[i] ==$ `nil`.

Про корректность обращений к памяти читайте Cormen'а, я нифига не понимаю, наверное это и не нужно?????

Время работы алгоритма $O(\log n)$. Это следует из того, что начальная инициализация и каждая итерация цикла проходят за $O(1)$ времени, а сам цикл выполняется $\log n$ раз: все значения, для которых ?????. $\square$

[3] Именно $O(1)$, так как мы разобрались с каждым из n битов по отдельности. Ни о каких «переносах» и сложении длинных чисел здесь речи не идет.

[4] Оценка из «Computational Complexity» Пападимитроу $W(n) = O(n^2 \log n)$ тоже верна, но грубее.

3.2 Параллельное вычисление всех глубин дерева

Задача. *Дано подвешенное неориентированное дерево на n вершинах, занумерованных* $\{0, \dots, n-1\}$ *в следующем формате. Имеются три массива* $left[0..n-1], right[0..n-1], parent[0..n-1]$*, для каждого* i $left[i], right[i]$ *и* $parent[i]$ *суть номера левого потомка, правого потомка, родителя вершины* i *(при отсутствии какого-то из параметров присвоено* `nil`*). Предположим, что эти массивы действительно задают дерево. Вычислить глубины всех вершин относительно корня.*

Решение за $O(\log n)$***.*** Сопоставим каждой вершине i три процессора A_i, B_i, C_i[5]. Перестроим дерево в ориентированный граф, вершины которого будут этими процессорами. Именно, проведем ребро:

- $A_i \to A_{left[i]}$, либо $A_i \to B_i$, если $left[i] ==$ `nil`;
- $B_i \to A_{right[i]}$, либо $B_i \to C_i$, если $right[i] ==$ `nil`;
- $C_i \to \dots$
  - $\dots B_{parent[i]}$, если i – левый потомок,
  - $\dots C_{parent[i]}$, если i – правый потомок,
  - $\dots$`nil`, если $parent[i] ==$ `nil` (i – корень) (можно, наверное, считать, что у корневой вершины нет процессора C).

Можно проверить[6], что у этого графа существует эйлеров обход, начинающийся в A корня и заканчивающийся в C корня.

Поместим теперь[7] в процессоры A_i число 1, в B_i число 0, в C_i число -1. эйлеров обход графа превратился в последовательность чисел, у которой мы умеем параллельно вычислять частичные суммы (мы учились это делать в 2.1).

Теорема. *Частичная сумма, вычисленная до* C_i *— это глубина вершины* i*.*

Доказательство. Заметим из устройства нашего дерева, что процессор C_i встречается всегда перед процессором A_i, A_i потомка встречается позже A_j предка, а с C_i, C_j наоборот. Процессор A_i добавляет 1, процессор B_i добавляет 0, процессор C_i добавляет -1, поэтому вклад в сумму от посещения поддеревьев нулевой. До C_i C-процессоры могут встречаться только в поддеревьях предков вершины i и поддеревьях самой вершины i, но они не вносят вклад в сумму. Поэтому в сумму добавляют только A-вершины предков (а их на 1 больше, чем глубина вершины), а отнимает только C_i, то есть вся сумма – это ровно глубина. □

4 (4) Приближенный алгоритм для задачи о рюкзаке (Осипов Д.)

Напомним сначала классическое, точное решение задачи о рюкзаке методом динамического программирования.

Задача (о рюкзаке с повторениями). Пусть есть n видов вещей, $i-$тая вещь имеет вес w_i и стоимость v_i. Количество каждого вида вещей неограничено. Пусть W – максимальный вес, который выдерживает рюкзак. Найти максимальную стоимость по всем наборам вещей, суммарный вес которого не превышает W.

---

[5] Я, честно говоря, так и не понял, по какой причине они называются процессорами. Как мне кажется, намного легче интерпретировать это как просто 3 числа, сопоставленные каждому узлу дерева.

[6] Например, вспомнить критерий полуэйлеровости: для всех вершин v, кроме двух, $in(v) = out(v)$, а для особых двух вершин $in(v_1) = out(v_1) + 1$ и $in(v_2) = out(v_2) - 1$.

[7] Вот видите, это просто числа а никакие не процессоры. Кто это название вообще придумал?

Точное решение за $O(n\sum v_i = nV)$. Динамическое программирование. Пусть $K[v]$ – минимальный вес набора стоимостью ровно v. Тогда $K[0] = 0$, $K[v] = \min\limits_{i=0}^{n}\{K[v - v_i] + w_i\}$. Ответ на задачу: $\max\{v : K[v] \le W\}$.

Таким образом заполняется массив длины $V + 1$, на каждый поиск минимума уходит $O(n)$ времени, на поиск ответа $O(n)$, всего $O(nV)$. $\square$

Теперь рассмотрим решение, которое может выдавать решение с заданной точностью. Именно, для всякого $0 < \varepsilon < 1$ это решение будет работать за $O(\frac{n^3}{\varepsilon})$ времени, а ценность найденного набора будет отличаться от оптимальной на множитель, не превышающий $(1 - \varepsilon)$.

Приближенное $\frac{1}{1-\varepsilon}$***-оптимальное решение за*** $O(\frac{n^3}{\varepsilon})$***.***

Зафиксируем параметр $\varepsilon > 0$. Заменим все v_i на:

$$\hat{v_i} = \left\lceil \frac{n}{\varepsilon} \cdot \frac{v_i}{v_{max}} \right\rceil$$

Запустим на новом наборе алгоритм ДП выше. Описание алгоритма закончено.

Оценим время работы. $V = \sum v_i \le n \cdot \frac{n}{\varepsilon} = \frac{n^2}{\varepsilon}$. Поэтому время $O\left(n \cdot \frac{n^2}{\varepsilon}\right) = O\left(\frac{n^3}{\varepsilon}\right)$.

Теперь точность.

Пусть оптимальное решение исходной задачи – набор S, его стоимость с точки зрения старой задачи $K^* = \sum\limits_{i \in S} v_i$.

С точки зрения новой задачи сумма этого набора оценивается как:

$$\sum_{i \in S} \hat{v_i} = \sum_{i \in S} \left\lceil \frac{v_i n}{\varepsilon v_{max}} \right\rceil \ge \sum_{i \in S} \left(v_i \cdot \frac{n}{\varepsilon v_{max}} - 1 \right) \ge K^* \frac{n}{\varepsilon v_{max}} - n$$

С точки зрения новой задачи набор S необязательно оптимален. То есть, если $\hat{S}$ – оптимальный с точки зрения новой задачи набор, то имеем

$$\sum_{i \in \hat{S}} \hat{v_i} \ge \sum_{i \in S} \hat{v_i} \ge K^* \frac{n}{\varepsilon v_{max}} - n$$

Нужно оценить, насколько стоимости наборов S и $\hat{S}$ отличаются с точки зрения старой задачи, т.е. сравнить величины $\sum\limits_{i \in S} v_i = K^*$ и $\sum\limits_{i \in \hat{S}} v_i$. Что же, так как $\hat{v_i} \le \frac{v_i n}{\varepsilon v_{max}}$,

$$\sum_{i \in \hat{S}} v_i \ge \sum_{i \in \hat{S}} \hat{v_i} \frac{\varepsilon v_{max}}{n} \ge \left(K^* \frac{n}{\varepsilon v_{max}} - n \right) \frac{\varepsilon v_{max}}{n} = K^* - \varepsilon v_{max} \ge K^* - \varepsilon K^* = K^*(1 - \varepsilon)$$

Таким образом, полученное решение хуже оптимального не более, чем в $\frac{1}{1-\varepsilon}$ раз. $\square$

5 (5) Set Cover - i (Осипов Д.)

Задача (о покрытии множествами, Set Cover). *Пусть дано множество* $\{e_1, \dots, e_n\} = E$ *и несколько подмножеств* $S_1, \dots, S_m \subseteq E$*, каждому* S_j *присвоен неотрицательный вес* w_j*. Необходимо выбрать из* $S_1, \dots, S_m$ *набор, полностью покрывающий* E*, с минимальным суммарным весом. Более формально: требуется найти такое* $I \subseteq \{1, \dots, m\}$*, что:*

$$\bigcup_{j \in I} = E \text{ и } \sum_{j \in I} S_j \text{ минимально.}$$

В этом билете представляются различные подходы к приближенным решениям этой задачи.

5.1 Сведение к задаче линейного программирования

Определение. *Задача линейного программирования* – задача минимизации (максимизации) некоторой линейной функции $h = h(x_1, \ldots, x_n)$ при ограничениях вида $g_k \wedge b_k$, где $\wedge$ есть один из знаков $\leq, \geq$, а $g_k = g_k(x_1, \ldots, x_n)$ – линейные функции; b_k – числа. Функция h называется *целевой* функцией.

Далее в тексте сокращение *ЛП-задача* будет означать *задача линейного программирования.*

Обозначим за f_j количество множеств среди $S_1, \ldots, S_m$, в которые входит элемент e_j. Положим $f = \max\limits_{i=1..n} f_j$. Оказывается, что эти параметры играют решающую роль в следующем решении.

Переформулируем задачу Set Cover. Каждому множеству S_j сопоставим переменную x_j, принимающую значение 1, если S_j взято в набор I, и 0 – иначе. Столбец $x = (x_1, \ldots, x_m)^T$ взаимно однозначно кодирует любой набор индексов I. Тогда целевая функция – суммарный вес покрытия – выглядит как $\sum\limits_{j=1}^{m} w_j x_j$. Ограничение на то, что набор I – покрытие, записывается так: каждый элемент e_i покрыт хотя бы одним элементом I, или же что условие $\sum\limits_{j\colon e_i \in S_j} x_j \geq 1$ выполнено для всех $i = 1..n$. Итак, формулировка задачи:

$$\sum_{j=1}^{m} w_j x_j \to \min,$$

$$\sum_{j\colon e_i \in S_j} x_j \geq 1, \quad i = 1..n,$$

$$x_j \in \{0, 1\}, \quad j = 1..m$$

Это <u>почти</u> ЛП-задача. Если бы умели решать такие задачи точно, решили бы и нашу – это просто ее переформулировка. «Ослабим» задачу до настоящей ЛП-задачи:

$$\sum_{j=1}^{m} w_j x_j \to \min,$$

$$\sum_{j\colon e_i \in S_j} x_j \geq 1, \quad i = 1..n,$$

$$x_j \geq 0 \quad j = 1..m$$

Мы перестали требовать, что x_j обязательно должен быть целым и не превышать единицы. Отметим, что если обозначить минимум ЦФ в исходной задаче за OPT, а в ослабленной – за Z^*, то будет справедлива оценка

$$Z \leq OPT,$$

так как фактически вторая задача – следствие первой.

Приближенное f-оптимальное решение (методом прямой ЛП-задачи, primal).

<u>Считаем, что ослабленную ЛП-задачу мы решать умеем.</u> Пусть $x^* = (x_1^*, \ldots, x_m^*)^T$ – оптимальное решение ослабленной ЛП-задачи, т.е. $Z = \sum\limits_{j=1}^{m} w_j x_j^*$. Сконструируем из нее решение исходной

задачи (и задачи Set Cover) следующим образом:

$$x_j = 1\,(j \in I) \iff x_j^* \geq 1/f;$$

Итак, алгоритм заключается в следующем: мы находим минимум x_j^* ослабленной ЛП-задачи, а далее в покрытие берем все S_j, для которых получилось $x_j^* \geq 1/f$. Теперь докажем его корректность и точность.

Найденный набор S-ок действительно покрывает всё E.

Именно, докажем, что каждый элемент $e_i \in E$ покрыт какой-то S-кой. Найденное решение x^* удовлетворяет ослабленной ЛП-задаче, то есть для данного e_i имеем $\sum\limits_{j:\, e_i \in S_j} x_j^* \geq 1$. В этой сумме по определению $f_i = |\{j : e_i \in S_j\}| \leq f$ членов, значит, хотя бы один из них $x_k^* \geq 1/f$. Значит, соответствующий $x_k = 1$, что доказывает то, что e_i покрыт S_k.

Теперь докажем f-оптимальность.

Обозначим (снова) минимальное значение целевой функции исходной почти-ЛП задачи за OPT, а ослабленной ЛП-задачи за $Z \leq OPT$. (То есть, в обозначениях x^* имеем $Z = \sum\limits_{j=1}^{m} w_j x_j^*$). Для всякого $j \in I$ имеем $x_j^* \geq 1/f$, или же $x_j^* \cdot f \geq 1$. Тогда значение целевой функции в найденном решении исходной почти-ЛП задачи оценивается как:

$$\sum_{j=1}^{m} w_j x_j = \sum_{j \in I} w_j \leq f \sum_{j \in I} w_j x_j^* \leq f \sum_{j=1}^{m} w_j x_j^* = f Z^* \leq f \cdot OPT.$$

Таким образом, найденное решение хуже оптимального не более, чем в f раз. □

5.2 Следствие для задачи вершинного покрытия (Vertex Cover)

Задача (о вершинном покрытии, Vertex Cover). *Пусть дан неориентированный граф $G = (V, E)$, каждой вершине i которого сопоставлен неотрицательный вес w_i. Найти минимальный по весу набор вершин $C \subseteq V$ такой, что всякое ребро графа хотя бы одним из двух концов лежит в C.*

Приближенное 2-оптимальное решение. Это частный случай задачи Set Cover: основное множество – множество ребер графа E, а каждой вершине $i \in V$ сопоставляется множество S_i веса w_i, состоящее из ребер, смежных с i. Причем в обозначениях предыдущего раздела каждое ребро (i, j) содержится ровно в двух множествах: S_i, S_j, поэтому $f = 2$, а значит, алгоритм становится 2-оптимальным. □

5.3 Двойственная задача

`От автора: к сожалению, получился не очень приятный для чтения параграф. Автор не смог вникнуть в «экономический смысл» двойственной ЛП-задачи, поэтому все рассуждения построены на противной формалистике с матрицами и суммами. Возможно, вы лучше поймете эту тему, прочитав ее здесь ("1.4 Rounding a dual solution")`

Задачи линейного программирования можно записывать в матричном виде. Вспомним нашу ослабленную ЛП-задачу:

$$\sum_{j=1}^{m} w_j x_j \to \min,$$

$$\sum_{j:\, e_i \in S_j} x_j \geq 1, \quad i = 1..n,$$

$$x_j \geq 0 \quad j = 1..m$$

Положим $w = (w_1, \dots, w_m)^T$ – столбец весов, тогда, очевидно, первое условие переписывается как:

$$w^T x \to \min$$

Со вторым условием разберемся так. Введем матрицу $\mathcal{E}$ размера $n \times m$:

$$\mathcal{E}_{ij} = \begin{cases} 1, & \text{если } e_i \in S_j \\ 0, & \text{иначе} \end{cases}$$

Тогда для фиксированного $1 \leq i \leq n$ условие $\sum\limits_{j: e_i \in S_j} x_j \geq 1$ переписывается как $\mathcal{E}_{i*} x \geq 1$. Ясно, что все такие n условий можно заменить одним матричным:

$$\mathcal{E}\, x \geq \mathbb{I}_n,$$

где за $\mathbb{I}_n$ обозначен столбец из единиц высоты n.

Наконец, третье условие, очевидно, просто заменяется на

$$x \geq \mathbb{O}_m,$$

где за $\mathbb{O}_m$ обозначен столбец из нулей высоты m.

Итак, мы получили задачу $w^T x \to \min$, $\mathcal{E} x \geq \mathbb{I}_n$, $x \geq \mathbb{O}_m$.

Определение. *Пусть дана ЛП-задача вида* $c^T x \to \min$ *с ограничениями* $Ax \geq b$, $x \geq \mathbb{O}$. ***Двойственная*** *к ней ЛП-задача ставится следующим образом:* $b^T y \to \max$ *при ограничениях* $A^T y \leq c$, $y \geq \mathbb{O}$.

В обозначениях определения имеем $c = w$, $A = \mathcal{E}$, $b = \mathbb{I}_n$. Поэтому двойственная к нашей ЛП-задаче такова:

$$\mathbb{I}_n^T y \to \max,$$

$$\mathcal{E}^T y \leq w,$$

$$y \geq \mathbb{O}_m$$

Использование этой двойственной задачи на самом деле приведет нас к алгоритму той же эффективности, но далее в билете она пригодится лучше.

«Разворачиваем» матричные обозначения. Перепишем первое ограничение:

$$(\mathcal{E}^T y)_i = \sum_{j=1}^{n} \mathcal{E}^T_{ij} y_j = \sum j = 1^n \mathcal{E}_{ji} y_j = \sum_{j=1}^{n} [e_j \in S_i] y_j = \sum_{j:\, e_j \in S_i} y_j, \quad i = 1..m$$

Разверните ЦФ и второе ограничение самостоятельно и убедитесь, что вы получили:

$$\sum_{j=1}^{n} y_j \to \max,$$

$$\sum_{i: e_i \in S_j} y_i \leq w_j,\ j = 1..m,$$

$$y_i \geq 0,\ i = 1..n$$

Мы наконец-то готовы к созданию приближенного алгоритма на основе двойственной задачи.

Приближенное f-оптимальное решение (методом двойственной ЛП-задачи, dual).

Аналогично первому разделу, считаем, что эту задачу мы решать умеем. Пусть $y^* = (y_1^*, \ldots, y_n^*)^T$ – оптимальное решение двойственной ЛП-задачи. Сконструируем из нее решение I' исходной задачи (и задачи Set Cover) следующим образом:

$$j \in I' \iff \sum_{i\colon e_i \in S_j} y_i^* = w_j,$$

т.е. берем только те S_j, для которых первое ограничение ЛП-задачи обращается в равенство. Описание алгоритма закончено.

Найденный набор S-ок действительно покрывает все E.

Действительно, пусть какое-то e_k оказалось не покрытым. Тогда в I' не взяты все j такие, что S_j содержит e_k, т.е. для всех $S_j \ni e_k$ справедливо

$$\sum_{i\colon e_i \in S_j} y_i^* < w_j.$$

Обозначим $\varepsilon = \min\limits_{j\colon e_k \in S_j} \left(w_j - \sum\limits_{i\colon e_i \in S_j} y_i^* \right) > 0$. Определим столбец y' следующим образом: $y'_k = y_k^* + \varepsilon$, а все остальные $y'_j = y_j^*$. Покажем, что это решение подходит в нашу ЛП-задачу.

1. Для всякого $S_j \ni e_k$ имеем $\sum\limits_{i:e_i \in S_j} y'_i = \varepsilon + \sum\limits_{i:e_i \in S_j} y_i^* \overset{\text{def } \varepsilon}{\leq} \left(w_j - \sum\limits_{i\colon e_i \in S_j} y_i^* \right) + \sum\limits_{i\colon e_i \in S_j} y_i^* = w_j$
2. А для всякого $S_j \not\ni e_k$ имеем просто $\sum\limits_{i:e_i \in S_j} y'_i = \sum\limits_{i:e_i \in S_j} y_i^* \leq w_j$.

Таким образом, проверено первое ограничение задачи. Второе ограничение $y_i \geq 0$ тривиально, тем самым, y' – решение ЛП-задачи. При этом решении значение ЦФ оказывается лучшим, чем при y^*: $\sum\limits_{j=1}^{n} y'_j = \varepsilon + \sum\limits_{j=1}^{n} y_j^* > \sum\limits_{j=1}^{n} y_j^*$, но мы предполагали, что y^* – оптимальное решение. Противоречие.

Теперь докажем f-оптимальность. Распишем суммарный вес найденного набора I':

$$\sum_{j \in I'} w_j = \sum_{j \in I'} \sum_{i:e_i \in S_j} y_i^* = \sum_{j=1}^{m} \sum i = 1^n [j \in I'][e_i \in S_j] y_i^* = \sum i = 1^n |\{j \in I' : e_i \in S_j\}| \cdot y_i^*$$

Оценим сверху в терминах $f_i = |\{j : e_i \in S_j\}|$ и $f = \max\limits_{i=1..n} f_i$:

$$\leq \sum_{i=1}^{n} f_i y_i^* \leq f \sum_{i=1}^{n} y_i^*$$

Последняя сумма равна оптимальному значению ЦФ двойственной задачи. Воспользуемся без доказательства следующим фактом:

Теорема (о сильной двойственности). *Рассмотрим ЛП-задачу и двойственную к ней. Если хотя бы у одной из двух задач есть оптимальное решение, то оно есть и у второй задачи, причем оптимальные значения целевых функций совпадают.*

Значит, $\sum\limits_{i=1}^{n} y_i^*$, будучи равной оптимальному значению прямой ЛП-задачи, не превосходит OPT. Таким образом,

$$\sum_{j \in I'} w_j \leq f \cdot OPT$$

□

5.4 Прямо-двойственный метод

Алгоритмы, которые решают задачи ЛП, довольно быстры. Но мы хотим еще быстрее.

Приближенное f-оптимальное решение (прямо-двойственный метод, primal-dual).

Вспомним, как мы из решения двойственной ЛП-задачи построили приближенное решение исходной, и как мы доказали, что это решение. Идея доказательства – если данное I не покрытие, то можем увеличить переменную, отвечающую за непокрытый элемент, – порождает следующий алгоритм:

`PrimalDual(`$E = \{e_1, \ldots, e_n\}, S_1, \ldots, S_m$`):`
$y = [0] * n$
I = []
while $\exists e_i \notin \bigcup\limits_{j \in I} S_j$ **do**
    $l =$ **все** индексы, для которых $e_i \in S_l$ **и** $\varepsilon = (w_l - \sum\limits_{k: e_k \in S_l} y_k)$ **минимален**
    $y_i \mathrel{+}= \varepsilon$
    Добавить в I все элементы l

Итераций внешнего цикла **while** не более n штук, так как каждый раз в I добавляем не менее одного элемента. Ясно (из раздела про двойственную задачу), что это корректный f-оптимальный алгоритм. □

6 (6) Set Cover – ii (Осипов Д.)

6.1 Жадный приближенный алгоритм

Условие задачи все еще `в том билете`.

Сейчас окажется, что обычный жадный подход часто дает результат лучше, чем все подходы к Set Cover, описанные до этого. Именно, если обозначить $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$, то получим:

Приближенное H_n-оптимальное решение.

Вот вполне интуитивный «жадник»:

`Greedy(`$E = [e_1, \ldots, e_n]$,$S = [S_1, \ldots, S_m]$,$w = [w_1, \ldots, w_m]$`):`
    $I = []$
    $\hat{S}_1, \ldots, \hat{S}_m = S_1, \ldots, S_m$
    while *I is not a set cover* **:**
        $l =$ any index j such that $\hat{S}_j \neq \varnothing$ and $\frac{w_j}{|\hat{S}_j|}$ is minimal
        I.`append`(l)
        for $j = 1 \ldots m$ **:**
            $\hat{S}_j = \hat{S}_j \setminus S_l$

Ясно, что этот алгоритм действительно дает покрытие всего E. Нужно доказать точность.

Доказываем H_n-оптимальность. Пусть алгоритм сделал l итераций. За n_k обозначим количество непокрытых элементов E перед k-той итерацией (полагаем по определению $n_{l+1} = 0$). Так, $n = n_1 > \ldots > n_{l+1} = 0$.

Пока поверим, что если на k-той итерации выбрано множество S_i, то справедливо неравенство:

$$w_i \le \frac{n_k - n_{k+1}}{n_k} OPT$$

По модулю этого факта доказываем H_n-оптимальность. Пусть I – множество индексов, найденное жадным алгоритмом. Тогда суммарный вес всех выбранных множеств оценивается как:

$$\begin{aligned}
\sum_{j \in I} w_j &\le \sum_{k=1}^{l} \frac{n_k - n_{k+1}}{n_k} OPT \\
&= OPT \cdot \sum_{k=1}^{l} \left(\underbrace{\frac{1}{n_k} + \dots + \frac{1}{n_k}}_{n_k - n_{k+1} \text{ раз}} \right) \\
&\le OPT \cdot \sum_{k=1}^{l} \left(\frac{1}{n_k} + \frac{1}{n_k - 1} + \dots + \frac{1}{n_{k+1}+1} \right) \\
&= OPT \cdot \sum_{k=1}^{n} \frac{1}{k} = OPT \cdot H_n
\end{aligned}$$

Это и требовалось. Теперь доказываем неравенство $w_i \le \frac{n_k - n_{k+1}}{n_k} OPT$.

Для данной итерации k и выбранного на ней элемента S_i обозначим I_k – множество индексов, выбранных на итерациях $1, \dots, k-1$, а для всякого $j = 1 \dots m$ положим $\hat{S}_j = S_j \setminus \bigcup_{p \in I_k} S_p$ – множество элементов из S_j, которые были покрыты на k-той итерации. Заметьте, что получается ровно те $\hat{S}_j$, которые фигурируют в псевдокоде. По смыслу алгоритма получается

$$\frac{w_i}{|\hat{S}_i|} = \min_{j \colon \hat{S}_j \ne \emptyset} \frac{w_j}{|\hat{S}_j|}.$$

Обозначим за O множество индексов в оптимальном решении (т.е. соответствующее суммарному весу OPT). Ясно, что $j \in O \implies \hat{S}_j \ne \emptyset$, так что:

$$\min_{j \colon \hat{S}_j \ne \emptyset} \frac{w_j}{|\hat{S}_j|} \le \min_{j \in O} \frac{w_j}{|\hat{S}_j|}$$

Вспомним такое неравенство из курса анализа. *Пусть $a_1, \dots, a_q, b_1, \dots, b_q$ – положительные числа. Тогда*

$$\min_{j=1..q} \frac{a_j}{b_j} \le \frac{\sum\limits_{j=1..q} a_j}{\sum\limits_{j=1..q} b_j} \le \max_{j=1..q} \frac{a_j}{b_j}$$

Применим его первую часть для чисел $w_j, |\hat{S}_j|$, где $j \in O$, получим:

$$\min_{j \in O} \frac{w_j}{\left|\hat{S}_j\right|} \le \frac{\sum\limits_{j \in O} w_j}{\sum\limits_{j \in O} |\hat{S}_j|}$$

Числитель просто равен OPT по определению, а знаменатель не меньше $n_k = |\bigcup_{j \in O} \hat{S}_j|$ (это просто количество оставшихся непокрытых элементов!). Резюмируя, имеем:

$$\frac{w_i}{|\hat{S}_i|} \le \frac{OPT}{n_k}$$

А так как на k-той итерации покрываем $|\hat{S}_i| = n_k - n_{k+1}$, получаем наконец:

$$w_i \le \frac{|\hat{S}_i| \cdot OPT}{n_k} = \frac{(n_k - n_{k+1}) \cdot OPT}{n_k}$$

□

7 (7) Транспортные сети. Задача о максимальном потоке. Разрез. Теорема о максимальном потоке и минимальном разрезе. Алгоритм Форда-Фалкерсона (Нечаев Е.)

Фактически эта глава — просто пересказ параграфов из кормена в правильном порядке. Начнем, с того, что это вообще такое. Итак,

7.1 Транспортные сети. Задача о максимальном потоке

Определение 7.1. ***Транспортной сетью*** *называется ориентированный граф* $G = \langle V, E\rangle$ *с функцией* $c\colon V \times V \to \mathbb{N}$*, которая называется* ***пропускной способностью****, причем* $c(u,v) = 0 \iff (u,v) \notin E$*, а также двумя выделенными вершинами —* ***источником*** s *и* ***стоком*** t*.*

Внимание! В источник могут входить ребра, а из стока выходить.

Для удобства предполагается, что любая вершина находится на некотором пути от источника к стоку (то есть граф связный).

Пример 7.1. *Здесь нужна картинка*

Определение 7.2. ***Потоком*** *называется функция* $f\colon V \times V \to \mathbb{R}$*, для которой выполняются следующие свойства:*

1. $\forall (u,v) \in V \times V\colon f(u,v) \le c(u,v)$
2. $\forall (u,v) \in V \times V\colon f(u,v) = -f(v,u)$
3. $\forall u \in V \setminus \{s,t\}\colon \sum_{v\in V} f(u,v) = 0$

Величиной *потока называется число* $|f| \overset{\text{def}}{=} \sum_{v \in V} f(s,v)$*.*

Одна из возможных интерпретаций этого – электрическая цепь. Тогда все свойства потоков превращаются в правила Кирхгофа.

Обратите внимание, что если есть ребро (u,v) и с потоком $f(u,v) \neq 0$, но нет ребра (v,u), то тем не менее $f(v,u) = -f(u,v) \neq 0$. Подумайте, почему если между вершинами нет ребра ни в каком направлении, то поток между ними нулевой[8].

Задача 7.1. *(о максимальном потоке) Дана транспортная сеть. Нужно найти в ней поток максимальной величины.*

Базовая идея: взять какой-нибудь (например, тривиальный) поток и увеличивать его, пока можно. Осталось только научиться все это делать, но нужно еще несколько определений.

Определение 7.3. *Для сети* G *и потока* f ***остаточной пропускной способностью*** *ребра* (u,v) *называется величина* $c_f(u,v) = c(u,v) - f(u,v)$*.* ***Остаточной сетью*** $G_f = \langle V, E_f\rangle$ *называется сеть на вершинах графа* G *с множеством ребер* $E_f = \{(u,v) \in V \times V | c_f(u,v) > 0\}$ *с пропускной способностью* c_f *и теми же источником и стоком.*

[8] потому что $0 = c(u,v) \ge f(u,v) = -f(v,u) \ge -c(v,u) = 0$

Обратите внимание, что если в G есть ребро (u, v), но нет ребра (v, u) (то есть его пропускная способность 0), то остаточная пропускная способность $c_f(v, u) = c(v, u) - f(v, u) = f(u, v)$, то есть если между вершинами есть одно из ребер с ненулевым потоком, то в остаточную сеть попадут оба. Получается, что $|E_f| \le 2|E|$.

Лемма 7.1. *Пусть $\langle G, c\rangle$ — транспортная сеть, f — поток в ней, G_f – остаточная сеть и в ней задан поток f'. Тогда $f + f'$ — поток в G, а его величина $|f + f'| = |f| + |f'|$.*

Доказательство. Проверим условия на потоки:

1. $$(f + f')(u, v) = f(u, v) + f'(u, v) \le f(u, v) + (c(u, v) - f(u, v)) = c(u, v)$$

2. $$(f + f')(u, v) = f(u, v) + f'(u, v) = -f(v, u) - f'(v, u) = -(f + f')(v, u)$$

3. $$\sum_{v\in V}(f + f')(u, v) = \sum_{v\in V} f(u, v) + \sum_{v\in V} f'(u, v) = 0$$

Поэтому это поток.

С величиной все понятно:

$$|f + f'| = \sum_{v\in V}(f + f')(s, v) = \sum_{v\in V} f(s, v) + \sum_{v\in V} f'(s, v) = |f| + |f'|$$

□

Определение 7.4. ***Увеличивающим путем*** *называется простой путь между s и t в G_f.*

Лемма 7.2. *G, c, s, t — сеть с потоком f, p — увеличивающий путь в G_f. Определим $f_p \colon V \times V \to \mathbb{R}$.*

$$f_p(u, v) = \begin{cases} c_f(p), & (u, v) \in p, \\ -c_f(p), & (v, u) \in p, \\ 0, & \text{otherwise} \end{cases}$$

где $c_f(p) = \min\{c_f(u, v) | (u, v) \in p\}$. Тогда f_p — поток в G с величиной $c_f(p) > 0$.

Доказательство.

1. $$f_p(u, v) \le c_f(p) \le c_f(u, v) = c(u, v) - f(u, v) \le c(u, v)$$
   если $f(u, v) \ge 0$. Остальные случаи тривиальны.

2. ...

3. Заметим, что для любой вершины v (не источника и не стока) в путь входит ровно одно ребро (u, v) и ровно одно ребро (v, w), то есть у всех остальных ребер потоки будут нулевые, а у этих они отличаются знаком, поэтому сумма потоков $\sum_{v\in V} f_p(u, v) = 0$.

□

Из лемм 7.1 и 7.2 следует, что поток на каждом ребре пути может быть увеличен на величину $c_f(p)$ (которая называется ***пропускной способностью пути***), чтобы не нарушить условия на сумму потоков и ограничение пропускной способности.

Теперь осталось научиться определять, чем максимальный поток отличается от немаксимального. Для этого нужно еще несколько определений и важная теорема, а именно

7.2 Разрез. Теорема о максимальном потоке и минимальном разрезе

Определение 7.5. ***Разрезом*** *сети G называется разбиение $V = S \sqcup T$, что $s \in S, t \in T$.* ***Чистым потоком*** *потока f через разрез (S,T) называется $f(S,T) \stackrel{\text{def}}{=} \sum_{x\in S}\sum_{y\in T} f(x,y)$.* ***Пропускной способностью разреза*** *называется $c(S,T) \stackrel{\text{def}}{=} \sum_{x\in S}\sum_{y\in T} c(x,y)$.* ***Минимальный разрез*** *— это тот, у которого пропускная способность минимальна.*

Лемма 7.3. *Чистый поток через любой разрез равен величине потока.*

Доказательство. Заметим, что $\sum_{x\in S}\sum_{y\in S} f(x,y) = 0$.

$$\sum_{x\in S}\sum_{y\in T} f(x,y) = \sum_{x\in S}\sum_{y\in V} f(x,y) - \sum_{x\in S}\sum_{y\in S} f(x,y) = \sum_{x\in S}\sum_{y\in V} f(x,y) = \sum_{y\in V} f(s,y) + \sum_{x\in S\setminus\{s\}}\sum_{y\in V} f(x,y) = \sum_{y\in V} f(s,y) = |f|$$

□

Лемма 7.4. *Величина любого потока не превышает пропускную способность любого разреза.*

Доказательство.

$$|f| = \sum_{x\in S}\sum_{y\in T} f(x,y) \le \sum_{x\in S}\sum_{y\in T} c(x,y) = c(S,T)$$

□

Теорема 7.1. ***(О максимальном потоке и минимальном разрезе)*** *G, c, s, t — транспортная сеть с потоком f. Следующие утверждения эквивалентны:*

1. *f — максимальный поток в G.*
2. *Остаточная сеть G_f не содержит увеличивающих путей.*
3. *$|f| = c(S,T)$ для некоторого разреза (S,T).*

Доказательство.

$1 \Rightarrow 2$ Если есть увеличивающий путь, то по лемме 7.2 можно построить поток со строго большей величиной, то есть f не максимальный.

$2 \Rightarrow 3$ Предположим, что нет увеличивающего пути. Определим $S = \{v \in V | \exists p\colon s \to v \text{ in } G_f\}, T = V \setminus S$. Понятно, что это разрез. В нем для любой пары $(u,v) \in S \times T$ выполняется $f(u,v) = c(u,v)$, потому что иначе бы ребро (u,v) попало бы в E_f (напомню, что там находятся только те ребра, у которых положительная остаточная пропускная способность) а значит существовал бы путь из s в v, это противоречит $v \in T$.

$$|f| = \sum_{x\in S}\sum_{y\in T} f(x,y) = \sum_{x\in S}\sum_{y\in T} c(x,y)$$

$3 \Rightarrow 1$ Из леммы 7.4 следует, что $|f| \le c(S,T)$. Поэтому если достигается равенство, то f — максимальный.

□

Теперь мы умеем все доказывать, чтобы описать алгоритм из базовой идеи, который называется

7.3 Алгоритм Форда-Фалкерсона

```
FFA(G = ⟨V, E⟩, c, s, t):
    foreach (u, v) ∈ E :
        f(u, v) := 0
        f(v, u) := 0
    while ∃p: s → t, p ⊆ E_f :
        c_f(p) := min{c_f(u, v) | (u, v) ∈ p}
        foreach (u, v) ∈ p :
            f(u, v) += c_f(p)
            f(v, u) := −f(u, v)
```

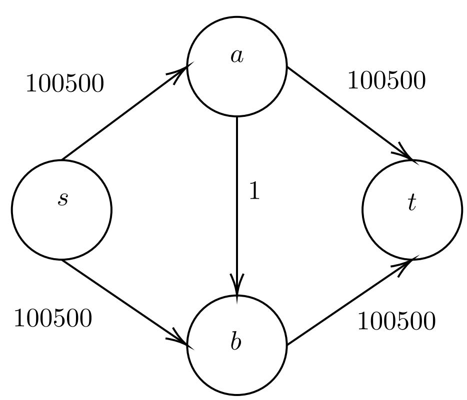
На практике, понятно, он возникает в основном только с целыми числами. Проблема этого алгортима в том, что не указано, как именно нужно искать увеличивающий путь. Если искать его неудачно[9], то алгоритм может и зависнуть.

В предположении, что числа рациональные (их можно свести к целым) и при использовании поиска в глубину или поиска в ширину для нахождения увеличивающего, время его работы составляет $O(|E||f^*|)$, где f^* — максимальный поток (в случае использования поиска в ширину этот алгоритм называется ***алгоритмом Эдмондса-Карпа***, для него в секции 8 мы докажем более точную оценку).

Проанализируем время работы. Первый цикл выполняется за время $\Theta(|E|)$, второй цикл выполняется не более $|f^*|$ раз (потому что величина потока в каждую итерацию увеличивается хотя бы на 1).

Время работы поиска $O(|V| + |E|) = O(|E|)$ (так как наш граф связный, а в нем $|E| \geq |V| - 1$, поэтому весь цикл выполняется за время $O(|E||f^*|)$.

Пример 7.2. *Алгоритм работает плохо, если найдется неудачный увеличивающий путь:*



Поиском в глубину находится путь $s \to a \to b \to t$, поэтому за одну итерацию поток увеличивается всего лишь на единицу.

В следующей итерации может найтись путь $s \to b \to a \to t$ (так как остаточная пропускная способность $b \to a$ теперь $0 - (-1) = 1$) и он опять уменьшится всего лишь на 1, поэтому нужный поток найдется за 100500 итераций.

[9] а еще если значения пропускных способностей иррациональные, пример можно найти в англоязычной википедии

7.4 Применение к паросочетаниям

Этим алгоритмом можно пользоваться, чтобы найти в двудольном графе $G = \langle V, E \rangle$ максимальное паросочетание. Для этого нам нужно теперь построить транспортную сеть и научиться сопоставлять потокам паросочетания.

Напомню, что мощностью паросочетания называется количество ребер в нем.

Дан двудольный граф $G = \langle V = L \sqcup R, E \rangle$, L, R — доли. Добавим еще две выделенные вершины s, t (источник и приемник) и построим сеть $G' = \langle V' = V \cup \{s, t\}, E' \rangle$, где $E' = \{(s, u) | u \in L\} \cup \{(u, v) | (u, v) \in E\} \cup \{(v, t) | v \in R\}$. У каждого ребра единичная пропускная способность.

Определение 7.6. *Поток называется* ***целочисленным****, если* $\forall (u, v) \in V \times V : f(u, v) \in \mathbb{Z}$.

Лемма 7.5. *Каждому паросочетанию в G взаимно однозначно соответствует целочисленный поток f в G' мощности $|f| = |M|$.*

Доказательство. Для начала построим поток по паросочетанию: если $(u, v) \in M$, то $f(s, u) = f(u, v) = f(v, t) = 1$, $f(t, v) = f(v, u) = f(v, s) = -1$, для всех остальных (u, v) $f(u, v) = 0$. Понятно, что это поток, и так как чистый поток через разрез $(L \cup \{s\}, R \cup \{t\})$ равен M, то и величина всего потока равна $|M|$.

Пусть теперь f – поток в G'. Определим

$$M = \{(u, v) | u \in L, v \in R, f(u, v) > 0\}$$

Поскольку пропускная способность каждого ребра равна 1, в вершину $u \in L$ входит не больше одной единицы потока. Так как она обязана куда-то выходить и поток целочисленный, она выходит по одному ребру. Так что единица положительного потока входит в u, согда существует единственная вершина $v \in R$, в которую эта единица входит. То же самое можно сказать про любую вершину $v \in R$, поэтому это паросочетание. Понятно, что величина этого потока равна $|M|$: по построению нашей сети $f(s, v) = 0 \forall v \in R \cup \{s, t\}$, поэтому

$$|f| = \sum_{v \in V' \setminus \{s\}} f(s, v) = \sum_{v \in L} f(s, v) = |M|$$

□

Понятно, что максимальному паросочетанию M соотвествтует максимальный поток (поскольку иначе существует паросочетание M', для которого $|M'| = |f'| > |f| = |M|$).

Чтобы применять условия этой леммы, нужно убедиться, что

Лемма 7.6. *Алгоритм Форда-Фалкерсона в сети с целочисленной пропускной способностью действительно строит целочисленный поток.*

Доказательство. Индукция по количеству итераций цикла. □

8 (8) Алгоритм Эдмондса-Карпа (Нечаев Е.)

Вариант реализации алгоритма Форда-Фалкерсона, где в качестве алгоритма поиска пути используется поиск в ширину (предполагается, что у всех ребер единичная длина), называется ***алгоритмом Эдмондса-Карпа***. Для него есть хорошая оценка времени работы $O(|V||E|^2)$. Она хорошая, потому что не зависит от величины максимального потока.

Обозначим как $\delta_f(u, v)$ кратчайшее расстояние между вершинами u и v в остаточной сети G_f.

Лемма 8.1. *Для всех вершин $v \in V \setminus \{s,t\}$ длина кратчайшего пути $\delta_f(s,v)$ в остаточной сети G_f монотонно возрастает при выполнении алгоритма.*

Доказательство. Будем доказывать от обратного. Предположим, что существует такое увеличение потока, которое приводит к уменьшению длины кратчайшего пути из s к некоторой вершине v. f – поток перед этим увеличением по пути, f' – поток после этого увеличения. Выберем v, чтобы $\delta_{f'}(s,v)$ было минимальным. Тогда $\delta_{f'}(s,v) < \delta_f(s,v)$. Пусть u – вершина перед v в этом пути в $G_{f'}$, то есть $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$. По выбору v $\delta_{f'}(s,u) \geq \delta_f(s,u)$ (иначе противоречие с минимальностью).

Теперь предположим, что $(u,v) \in E_f$. Но тогда

$$\delta_{f'}(s,v) = \delta_{f'}(s,u) + 1 \geq \delta_f(s,u) + 1 = \delta_f(s,v)$$

Теперь рассмотрим случай, когда $(u,v) \notin E_f$ (но $(u,v) \in E_{f'}$). Заметим, что такое может произойти только в том случае, когда поток на ребре $f'(u,v) < f(u,v)$ ($c_{f'}(u,v) > 0 = c_f(u,v)$), а значит, поток на ребре (v,u) увеличился. Алгоритм увеличивает поток только вдоль кратчайших путей, а это значит, что в G_f кратчайший путь $s \to u$ содержит ребро (v,u). Поэтому

$$\delta_f(s,v) = \delta_f(s,u) - 1 \leq \delta_{f'}(s,u) - 1 = \delta_{f'}(s,v) - 1 - 1$$

Опять получили противоречие с условием $\delta_{f'}(s,v) < \delta_f(s,v)$. □

Теперь мы можем посчитать ограничение на количество итераций основного цикла (того, в котором проводятся увеличения пути) алгоритма.

Определение 8.1. ***Критическим*** *назовем ребро (u,v) в пути p, для которого выполняется $c_f(u,v) = c_f(p)$ (ребро с наименьшей пропускной способностью из леммы 7.2).*

Лемма 8.2. *Количество итераций основного цикла – $O(|V||E|)$.*

Доказательство. Понятно, что в каждом увеличивающем пути есть критическое ребро, поэтому нам нужно посчитать, сколько раз каждое ребро может побывать критическим. Докажем, что не больше $\frac{|V|}{2} - 1$ раз.

Пусть $(u,v) \in E$ – критическое ребро. Так как увеличение проходит по кратчайшему пути, $\delta_f(s,v) = \delta_f(s,u) + 1$. После этого это ребро пропадет из остаточной сети и появится обратно, только если (v,u) появится в увеличивающем пути. Если f' – это такой новый поток, то $\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$. По лемме 8.1, $\delta_f(s,v) \leq \delta_{f'}(s,v)$, а значит

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \geq \delta_f(s,v) + 1 = \delta_f(s,u) + 2$$

Так что между случаями, когда ребро становится критическим, расстояние от источника вырастает как минимум на 2. Промежуточными вершинами на пути от s к u не могут быть s, u, и t. А это значит, что пока вершина u станет недостижимой из источника, расстояние до нее не превысит $|V|-2$. Поэтому ребро (u,v) станет критическим не более $\frac{|V|-2}{2}$ раз (делим на 2, потому что половину случаев ребро (v,u) становится критическим). Так как всего ребер в остаточной сети может быть $O(|E|)$ (обоснование есть на странице 17), количество критических ребер будет $O(|E||V|)$.

Так как внутренность цикла выполняется за время $O(|E|)$, то общее время работы алгоритма Эдмондса-Карпа — $O(|V||E|^2)$. □

9 (9) Алгоритм проталкивания предпотока (Нечаев Е.)

Этот алгоритм также находит максимальный поток, но отличается от предыдущих описанных алгоритмов тем, что не является вариантом алгоитма Форда-Фалкерсона, а также другой оценкой времени работы – $O(|V|^2|E|)$.

Определение 9.1. ***Предпотоком*** *называется функция* $f\colon V \times V \to \mathbb{R}$ *на вершинах транспортной сети* $G = \langle V, E\rangle, s, t$*, для которой выполняются следующие свойства:*

1. $\forall (u, v) \in V \times V\colon f(u, v) \le c(u, v)$
2. $\forall (u, v) \in V \times V\colon f(u, v) = -f(v, u)$
3. $\forall u \in V \setminus \{s\}\colon \sum_{v \in V} f(v, u) \ge 0$

$e(u) = \sum_{v \in V} f(v, u)$ *называется* ***избыточным потоком****.*

Вершина $u \in V$ *называется* ***переполненной****, если* $e(u) > 0$*.*

Определение 9.2. *Функция* $h\colon V \to \mathbb{N}$ *называется* ***функцией высоты****, если выполняются следующие свойства:*

1. $h(s) = |V|$
2. $h(t) = 0$
3. $\forall (u, v) \in E_f\colon h(u) \le h(v) + 1$

9.1 Интуитивные соображения

Представим, что наша сеть – это система из резервуаров V, соединенных трубами E и находящихся на разной высоте h. Предпоток – это жидкость, которая течет по трубам, но где-то ее втекает больше, чем вытекает, и она остается в резервуаре (мы предполагаем, что они бесконечные). Можно "перелить"(операция проталкивания) жидкость из резервуара в соединенные трубой резервуары (увеличить значение предпотока на смежных трубах, если выполняются соответствующие интуитивные условия: высота резервуара u, из которого переливают, должна быть на единицу больше высоты резервуара v, в который переливают, и $c_f(u, v) > 0$), находящиеся на меньшей высоте или, если таких не найдется, "поднять"(операция поднятия) резервуар на высоту на единицу большую, чем самый нижний из смежных резервуаров.

Почти очевидно, что в таком случае предпоток превратится в поток. Как будет показано, он будет и максимальным.

9.2 Операция проталкивания

```
Push((u, v) ∈ E):
    if e(u) > 0 and c_f(u, v) > 0 and h(u) − h(v) = 1 :
        d := min(e(u), c_f(u, v))
        f(u, v) += d
        f(v, u) := −f(u, v)
        e(u) −= ‘d‘
        e(v) += ‘d‘
```

Условие $h(u) - h(v) = 1$ нужно, так как из отрицания пункта 3 условия на функцию высоты следует, что если высоты различаются больше чем на единицу, остаточных ребер просто нет, поэтому проталкивать что-либо бессмысленно.

Понятно, что предпоток после проталкивания остается предпотоком (сохранение свойств 1, 2 совсем очевидно, свойство 3 сохраняется, потому что мы вычитаем что-то, не превосходит $e(u)$).

Проталкивание называется ***насыщающим***, если после него $c_f(u, v) = 0$ (ребро, соответственно, становится ***насыщенным***). Понятно, что после ненасыщающего проталкивания вершина u перестает быть переполненной (мы так выбираем $d = \min(e(u), c_f(u, v))$, что зануляется либо переполненность, либо остаточная пропускная способность).

Лемма 9.1. *После проталкивания функция высоты остается функцией высоты (не нарушаются ее свойства).*

Доказательство. Так как высоты не меняются, нужно только проверить, что сохраняется условие 3. Операция может удалить ребро (u, v) из E_f (если $c_f(u, v) < e(u)$) или добавить ребро (v, u), если его не было (так как если $e(u) < c_f(u)$, то $c_{f_{\text{new}}}(v, u) = c(v, u) + f_{\text{new}}(u, v) > 0 = c_f(v, u)$). В первом случае удаление ребра делает неактуальным ограничение. Во втором случае выполняется $h(v) = h(u) + 1$, поэтому $h(v) \leq h(u) + 1$. Поэтому h остается функцией высоты. ☐

9.3 Операция подъема

```
Relabel(u ∈ V):
    if e(u) > 0 and ∀v ∈ {x|(u,x) ∈ E_f}: h(u) ≤ h(v) :
        h(u) += 1 + min_{(u,v)∈E_f}{h(v)}
```

Лемма 9.2. *После подъема функция высоты остается функцией высоты (не нарушаются ее свойства).*

Доказательство. Докажем, что эта функция назначает наибольшую возможную высоту, удовлетворяющую условиям высоты. Так как вершина u переполнена ($e(u) > 0$), то существует вершина v, для которой $f(v, u) > 0$, значит, $c_f(u, v) = c(u, v) - f(u, v) = c(u, v) + f(v, u) > 0$, а значит, $(u, v) \in E_f$. Поэтому $\min_{(u,v)\in E_f}\{h(v)\}$ определено и это наибольшее возможное значение, удовлетворяющее условию 3.

Понятно, что источник и сток выше поднять нельзя, рассмотрим другую вершину к u и входящее в него ребро (u, v). Поскольку высота строго увеличивается ($h(u) \leq h(v)$ для всех $(u, v) \in E_f$ до поднятия, а значит, $h(u) < 1 + h(v) = h_{\text{new}}(u)$ для такого смежного v, что $h(v)$ минимально), выполняется $h(w) \leq h(u) + 1 \leq h_{new}(u) + 1$ ☐

9.4 Начальный предпоток

Начальный предпоток определяется так:

$$f(u, v) = \begin{cases} c(u, v), & u = s, \\ -c(u, v), & v = s, \\ 0, & \text{otherwise} \end{cases}$$

Начальная высота определяется так:

$$h(u) = \begin{cases} |V|, & u = s, \\ 0, & \text{otherwise} \end{cases}$$

Это действительно корректно определенная функция высоты, поскольку единственные ребра, для которых не выполняется условие 3 – это ребра, выходящие из источника, но так как для них значение предпотока равно значению пропускной способности, их нет в E_f.

9.5 Алгоритм. Его корректность.

Для начала нужно доказать лемму:

Лемма 9.3. *Пусть G, s, t – транспортная сеть с предпотоком f и какой-то функцией высоты h. Тогда к любой переполненной вершине можно применить либо проталкивание, либо подъем.*

Доказательство. Для $(u, v) \in E_f$ выполняется $h(u) \leq h(v) + 1$ (по условию на высоту). Если $h(u) - h(v) = 1$ для какой-то вершины v, то выполняется операция проталкивания, а иначе $h(u) < h(v) + 1 \Rightarrow h(u) \leq h(v)\ \forall (u, v) \in E_f$, а значит, выполнима операция подъема. □

Понятно, что они не могут быть выполнены одновременно.

Теперь мы можем написать алгоритм:

```
PPA(G = ⟨V,E⟩,c,s,t):
    initialize-preflow(G,s)
    while ∃u: (∃v ∈ V: Pushable(u,v)) ∨ Relabelable(u) :
        if Pushable((u,v)) :
            Push((u,v))
        else
            Relabel(u)
```

Лемма 9.4. *$G = \langle V, E\rangle, s, t, f, h$ — транспортная сеть с источником, стоком, предпотоком и функцией высоты. Тогда нет пути из источника в сток в G_f.*

Доказательство. Предположим, что существует такой путь $v_0 = s \to v_1 \to \cdots \to v_{k-1} \to t = v_k, (v_i, v_i + 1) \in E_f$. Можно считать, что этот путь простой, а поэтому $k < |V|$. Кроме того, $h(v_i) \leq h(v_{i+1}) + 1 \forall 0 \leq i \leq k - 1$. Но, сложив все эти неравенства, мы получим, что $h(s) \leq h(t) + k \Rightarrow |V| \leq 0 + k$. Противоречие. □

Теорема 9.1. ***(О корректности)*** *После окончания работы алгоритма f становится максимальным потоком.*

Доказательство. Понятно, что f, инициализированный в initialize_preflow, является предпотоком.

В процессе выполнения алгоритма производятся операции проталкивания и поднятия, которые, как мы уже знаем, оставляют f и h предпотоком и функцией высоты.

После окончания работы все вершины, кроме s и t должны иметь избыточный поток 0 (по лемме 9.3). Поэтому это поток. По лемме 9.4 нет пути из источника в сток, а значит по теореме 7.1 этот поток — максимальный. □

9.6 Время работы

Лемма 9.5. *G, s, t, f – Транспортная сеть с предпотоком. Тогда для любой вершины u с ненулевым избыточным потоком существует простой путь $u \to s$ в G_f.*

Доказательство. Пусть $U = \{v | \exists p\colon u \to v$ — простой путь в $G_f\}$. Предположим, что $s \notin U$.

Заметим, что $\forall v \in U, w \in V \setminus U\colon f(w, v) \leq 0$, так как если бы $f(w, v) > 0$, то $f(v, w) < 0 \Rightarrow c_f(v, w) > 0$, а значит $(v, w) \in E_f$ и существует простой путь $u \to v \to w$, что противоречит выбору w.

Отсюда следует, что

$$\sum_{x\in U} e(x) = \sum_{x\in U}\sum_{v\in V} f(v,x) = \sum_{y\in V\setminus U}\sum_{x\in U} f(y,x) + \sum_{y\in U}\sum_{x\in U} f(y,x) = \sum_{y\in V\setminus U}\sum_{x\in U} f(y,x) \le 0$$

Поскольку избыточные потоки неотрицательны для любой вершины, кроме s, а $s \notin U$, то все избыточные потоки нулевые, а значит $e(u) = 0$, что противоречит условию. $\square$

Лемма 9.6. ***(Ограничение на функцию высоты)*** *Во время работы алгоритма* $\forall u \in V : h(u) \le 2|V| - 1$.

Доказательство.
$$h(s) = |V| \le 2|V| - 1$$
$$h(t) = 0 \le 2|V| - 1$$

Для $u \in V \setminus \{s,t\}$ в начале алгоритма $h(u) = 0$. После операции поднятия вершина переполнена, а значит есть простой путь $v_0 = u \to v_1 \to \dots \to s = v_k$ (по лемме 9.5), а значит, $k \le |V| - 1$. По условию на высоту $h(v_i) \le h(v_{i+1}) + 1$. Складывая неравенства, получаем $h(u) \le h(s) + k \le |V| + |V| - 1$. $\square$

Теорема 9.2. ***(Оценка времени)*** *Алгоритм выполняется за время* $O(|V|^2|E|)$.

Доказательство. Построим ограничение на каждую из операций: на поднятия, насыщающие и ненасыщающие проталкивания.

Операций поднятия меньше чем $2|V|^2$. Тут все совсем просто. Подниматься могут $|V \setminus \{s,t\}| = |V| - 2$ вершин и, так как изначально все высоты 0, а верхняя граница $2|V| - 1$, всего поднятий не больше $(|V| - 2)(2|V| - 1) < 2|V|^2$.

Операций насыщающих проталкиваний меньше чем $2|V||E|$. (напомню, что насыщающим проталкиванием вдоль ребра (u,v) называется такое, что после него $c_f(u,v) = 0$.) Будем считать проталкивания вдоль (u,v) и вдоль (v,u) вместе. Когда произошло проталкивание вдоль (u,v), выполнялось $h(u) = h(v) + 1$. чтобы оно произошло еще раз, должно произойти проталкивание вдоль (v,u), для которого требуется $h(v) = h(u) + 1$, поэтому высота u должна увеличиться (уменьшиться она не может) хотя бы на 2. Из ограничения на высоту получаем, что насыщающих увеличений вдоль (u,v) или (v,u) должно быть меньше $\frac{2|V|-1}{2} + \frac{2|V|-1}{2} < 2|V|$. Значит всего насыщающих увеличений вдоль любого ребра должно быть меньше $2|V||E|$.

Операций ненасыщающих проталкиваний меньше чем $4|V|^2(|V| + |E|)$. Рассмотрим величину $\Phi = \sum\limits_{\substack{v\in V\\ e(v)>0}} h(v)$. Понятно, что $\Phi \ge 0$. В начале и в конце выполнения алгоритма $\Phi = 0$ (в конце, потому что у потока единственная переполненная вершина – это t, но ее высота 0) и оно может измениться после любой из оперций, однако при поднятии и насыщающем проталкивании значение обязательно вырастет меньше чем на $2|V|$: первое — из ограничения на высоту, а второе – из того, что только одна вершина v (если проталкивать вдоль (u,v)) может стать переполненной, а ее высота меньше строго меньше $2|V|$. При ненасыщающем же проталкивании Φ уменьшается хотя бы на 1: пусть мы проталкиваем вдоль (u,v). После него $e(u) = 0$, поэтому Φ уменьшилась на $h(u)$, а вершина v могла стать, а могла не стать переполненной. Если она не стала, то она не влияет на Φ, а если стала, то к Φ добавилось $h(v)$, но так как $h(u) - h(v) = 1$, Φ уменьшилось на 1. Из прошлых пунктов мы знаем количество увеличений и насыщающих проталкиваний, а значит знаем верхнюю границу на Φ: $\Phi < (2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|)$. Так как Φ в конце становится нулем, ненасыщающих проталкиваний меньше чем $4|V|^2(|V| + |E|)$.

Из всего этого получается оценка на количество операций. $O(2|V|^2 + 2|V||E| + 4|V|^2(|V| + |E|)) = O(|V|^2|E|)$ (напомню, что у нас связный граф, а значит, $|V| \le |E| + 1$).

Если хранить список переполненных вершин и для каждой вершины хранить список менее высоких соседей, то операция проталкивания реализуется за $O(1)$, а операция поднятия за $O(|V|)$ (поскольку нужно вычислить минимум). Понятно, что при такой структуре выбор операции реализуется за $O(1)$, что дает оценку времени. $\square$

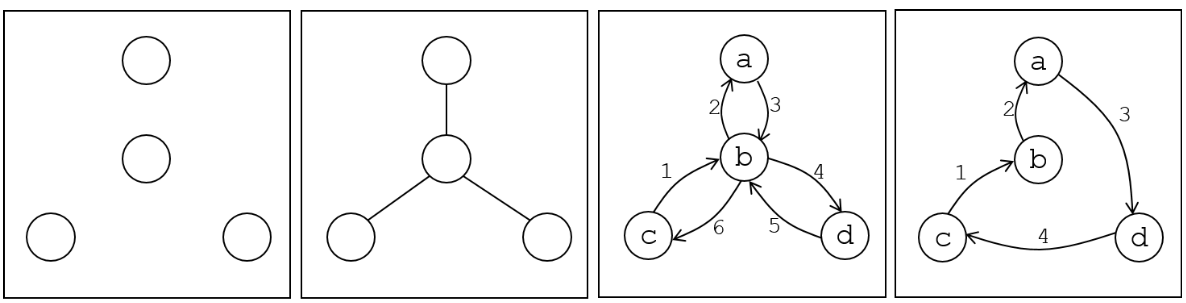
10 (10) Приближенные алгоритмы для метрической задачи коммивояжера

Задача. ***(Метрическая задача коммивояжера, metric TSP).*** *Дан* ***полный*** *неориентированный граф с неотрицательными весами l у ребер, удовлетворяющий неравенству треугольника: для любой тройки вершин u, v, w верно $l(uv)+l(vw) \geq l(uw)$. Найти в нем цикл минимальной длины, проходящий по всем вершинам (минимальный гамильтонов цикл).*

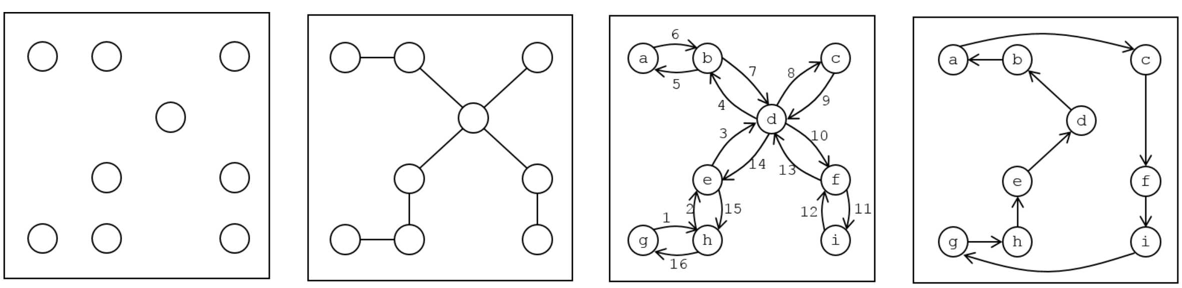
10.1 2-оптимальное решение

Решение (2-оптимальное). Найдем T – минимальное остовное дерево в G. Удвоим в T каждое ребро, получится эйлеров граф D. Пусть C – порядок вершин в эйлеровом цикле в D. Построим по нему гамильтонов цикл A следующим образом: для всякой вершины v удалим все ее вхождения в список C, кроме первого. Описание алгоритма закончено.

Два примера работы алгоритма на полных графах, заданных набором точек на плоскости:



T   **C:** cbabdba   **A:** cbadc



T   **C:** ghedbabdcdfifdehg   **A:** ghedbacfig

Обозначим TSP – вес оптимального гамильтонова цикла, MST – вес найденного минимального остовного дерева T.

Теорема. *Вес $A \leq 2 \cdot TSP$*

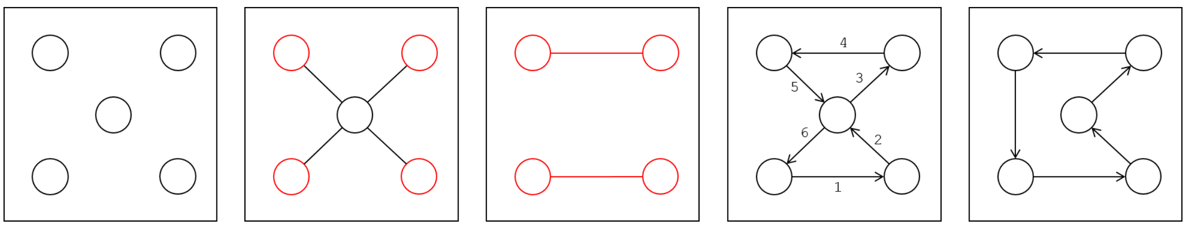
Доказательство. Заметим, что $MST \leq TSP$. Действительно, удалением любого ребра из гамильтонова цикла мы получаем какое-то остовное дерево, вес которого не превосходит TSP.

Вес эйлерова цикла C по определению равен $2 \cdot MST \leq 2 \cdot TSP$. Вес A не превосходит C, потому что при каждом удалении вершины суммарное расстояние неувеличивается по неравенству треугольника. Стало быть, вес $A \leq 2 \cdot TSP$. □

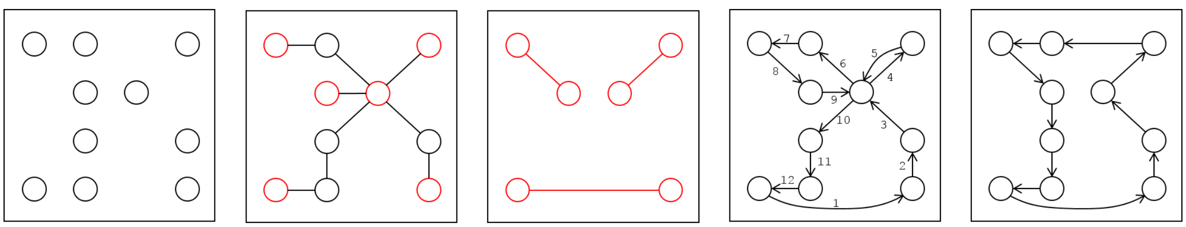
10.2 1.5-оптимальное решение

Решение (1.5-оптимальное). Найдем T – минимальное остовное дерево в G. Выделим все вершины нечетной степени в T, их четное число. Обозначим индуцированный (из G) граф на этих вершинах O. Этот граф все еще полный, значит в нем есть совершенное паросочетание. Пусть M – ребра минимального совершенного паросочетания в O. Добавим их в T (если какое-то ребро M уже есть в T, то добавим еще раз) – получим граф T', у которого степени всех вершин четные. Дальнейшие действия те же: пусть C – порядок вершин в эйлеровом цикле в T', гамильтонов цикл A строится по C выкидыванием всех повторных вхождений всякой вершины. Описание алгоритма закончено.

Снова два примера работы алгоритма:



T,O **M** **T′** **A**



T,O **M** **T′** **A**

Снова обозначим TSP – вес оптимального гамильтонова цикла, MST – вес остовного дерева T.

Теорема. *Вес* $A \leq \frac{3}{2} \cdot TSP$

Доказательство. По тем же рассуждениям (неравенство треугольника) вес найденного гамильтонова цикла A не превосходит вес эйлерова графа T', который равен MST + вес M, и по тем же рассуждениям $MST \leq TSP$. Достаточно показать, что вес $M \leq TSP/2$, а для этого в свою очередь достаточно доказать, что существует какое-то совершенное паросочетание на вершинах O веса $\leq TSP/2$.

Мы построим это паросочетание так. Упорядочим вершины M в том порядке, в котором они идут в **оптимальном** гамильтоновом обходе в G. Пусть H – гамильтонов цикл на вершинах M в указанном порядке. Так как H получен из оптимального гамильтонова цикла удалением вершин из него, то вес $H \leq TSP$. Далее, удалением из H ребер через одно мы можем получить два различных совершенных паросочетания на вершинах M, причем их объединение есть в точности H. Сумма весов этих двух паросочетаний есть вес H, таким образом, хотя бы у одного из двух паросочетаний вес не превосходит $TSP/2$, что и требовалось. □

11 (11) Алгоритмы Прима и Крускала для задачи о минимальном остовном дереве (Нечаев Е.)

Определение 11.1. *Для связного графа $G = \langle V, E\rangle$* ***остовным деревом*** *называется подграф $G' = \langle V, E'\rangle, E' \subseteq E$, который является деревом.*

Задача 11.1. *В связном взвешенном неориентированном графе $G = \langle V, E\rangle$ с весовой функцией $w\colon E \to \mathbb{R}$ найти остовное дерево минимального веса.*

Вспомним, что подмножества ребер графа, в которых нет циклов, являются независимыми в *цикловом матроиде*. Наша задача превращается в поиск базы минимального веса, для которого можно использовать *жадный алгоритм*[10]: начиная с пустого множества последовательно добавляем ребра минимального веса, пока можем.

Алгоритмы Прима и Крускала являются реализациями этого подхода. Остается только научиться быстро находить ребро минимального веса, который можно добавить, чтобы множество осталось независимым.

Определение 11.2. ***Разрезом*** *графа $G = \langle V, E\rangle$ называется пара $(S, T), S, T \subseteq V$, что $V = S \sqcup T$. Ребро называется* ***пересекающим разрез****, если концы ребра находятся в разных множествах разреза. Разрез называется* ***согласованным*** *со множеством A, если никакое ребро из A не пересекает разрез. Ребро называется* ***легким****, если оно пересекает разрез и имеет минимальный вес среди всех таких ребер, пересекающих разрез.*

Теорема 11.1. *В графе $G = \langle V, E\rangle$ с весовой функцией w $A \subseteq E$ – независимое подмножество, согласованое с разрезом (S, T), ребро (u, v) – легкое. Тогда $A \cup \{(u, v)\}$ – подмножество базы минимального веса, содержащего A.*

Доказательство. Понятно, что множество будет независимым: если это нет так, то еще какое-то ребро пересекает разрез.

Пусть M – база минимального веса, содержащая $A \cup \{(u, v)\}$, M' – другая база минимального веса, содержащая A. Если она не содержит (u, v), то она содержит какое-то другое ребро (x, y), пересекающее разрез, но тогда $w(M') = w(M) - w((u, v)) + w((x, y)) \geq w(M)$. Но также $w(M') \leq w(M) \Rightarrow w(M') = w(M)$. □

Следствие 11.1. *$G = \langle V, E\rangle, w$ — неориентированный взвешенный связный граф. $A \subseteq E$ – независимое множество в его цикловом матроиде. $C = \langle V_C, E_C\rangle$ – компонента связности леса $G_A = \langle V, A\rangle$. Если (u, v) – легкое ребро, которое соединяет C с другой компонентой связности, то $A \cup \{u, v\}$ — подмножество базы минимального веса, содержащего A.*

Доказательство. Разрез $(V_C, V \setminus V_C)$ согласован с A и (u, v) его пересекает. □

11.1 Алгоритм Крускала

Этот алгоритм использует структуру данных, которая называется

11.1.1 Система непересекающихся множеств

11.2 Алгоритм Прима

Это тоже вариант жадного алгоритма. Он похож на алгоритм Дийкстры, в частности, использует *очередь с приоритетами* (она была у Охотина в лекции 4, а описание реализации с помощью кучи в лекции 5).

[10] в курсе комбинаторики был поиск множества максимального веса, но это, по большому счету, одно и то же, потому что можно инвертировать все веса

Этот алгоритм находит остовное дерево, сторя его из корня $r \in V$.

```
def Prim-MST(G = ⟨V, E⟩, w, r ∈ V):
```

12 (12) Вероятностные алгоритмы с односторонней ограниченной вероятностью ошибки. Алгоритм Фрейвальдса для проверки умножения матриц. (Ермошин И.)

Вероятностные алгоритмы с односторонней ограниченной вероятностью ошибки. *Нам надо что-то проверить. Придумываем алгоритм, который это проверяет, но может ошибиться. Более точно: при истинном успехе алгоритм всегда сообщает об успехе, но при истинной неудаче алгоритм может ошибиться – сообщить об успехе с вероятностью p. Повторив его 10 раз, получим вероятность ошибки p^{10}, что, вероятно(ha ha), гораздо меньше.*

Алгоритм Фрейвальдса для проверки умножения матриц. *Есть три матрицы: $A_{m,n}$, $B_{n,k}$ и $C_{m,k}$, хотим узнать $A \times B = C$ или нет.*

Решение за $O\left(\frac{mnk}{\min(m,n,k)}\right)$ с вероятностью ошибки $\le \frac{1}{2}$.

NB: *Если все матрицы A, B, C квадратные, то мы проверим за $O(n^2)$, а если бы проверяли умножением матриц - это $O(n^{2.8})$ (алгоритм Штрассена)*

Сгенерируем случайный столбец r длины k из нулей и единиц (все равновероятно). Давайте проверять равенство $AB \times r = C \times r$; если умножать так: $A \times (B \times r)$, получится время $O(nk + mn + mk) = O\left(\frac{mnk}{\min(m,n,k)}\right)$, где каждое слагаемое есть время перемножения матриц $B \times r$, $A \times Br$ и $C \times r$.

Убедимся в том, что у алгоритма все хорошо

Очевидно, если $A \times B = C$, алгоритм так и сообщит.

Посчитаем вероятность ошибки, т.е. когда при $A \times B \neq C$ для случайно выбранного вектора r окажется $AB \times r = C \times r$.

Перепишем $ABr = Cr$ как $Xr = 0$, $X = AB - C$. Посмотрим на какой-нибудь ненулевой элемент x_{kl}. Имеем:

$$\sum_{i=1, i \neq l}^{n} x_{ki} r_i + x_{kl} r_l = 0$$

Из этого выражения однозначно определяется r_l. Это означает, что уже векторов r, удовлетворяющих $Xr = 0$, не более 2^{k-1}, а шанс выбрать такой вектор не превосходит $\frac{2^{k-1}}{2^k} = \frac{1}{2}$. Таким образом, вероятность ошибки $\le \frac{1}{2}$. □

13 (13) (В РАЗРАБОТКЕ) Вероятностный алгоритм для сравнения строк на расстоянии и алгоритм Рабина-Карпа. (Ермошин И.)

Вероятностный алгоритм для сравнения строк.

Под «на расстоянии» имеется в виду, что мы хотим потратить как можно меньше памяти на сравнение этих строк.

Есть две строчки a и b длины n над $\{0;1\}$, хотим их сравнить. Выберем случайное простое число p от 3 до τ, сравним остатки (a и b отождествим с числами, которые они задают при переводе в 10-ю систему счисления) от деления на p. Посмотрим на вероятность того что $a \neq b$, но $a \mod p = b \mod p$. Коли сравнивать мы будем числа по модулю p, нам потребуется только $O(\log p)$ памяти на хранение двух остатков. Плохие числа - это $\{p \in \mathbb{P} | (a-b) \vdots p\}$.
Лемма *У числа $k \leq 2^n$ меньше n различных простых делителей(очевидно)*
Пусть $\tau = n^2 \log^2(n)$; очевидно, $a - b \leq 2^n$, тогда $P_{\text{ошибки}} \leq \frac{n}{\frac{\tau}{\log \tau}} = O(\frac{1}{n})$. Здесь мы пользуемся ослабленной версией PNT: $\lim_{n\to\infty} \frac{\pi(n)}{\frac{n}{\log n}} = 1$.

Алгоритм Рабина-Карпа Есть две строчки, $|a| = m \leq |b| = n$, хотим посмотреть, является ли a подстрокой b.
Обзовем $b(i) = b_i b_{i+1} \ldots b_{i+m-1}$, если отождествить их с числами, получим $b(i) = \frac{b(i-1)-b_{i-1}}{2} + 2^{m-1} b_{i+m-1}$, посчитаем $\{b(i)\}_{0 \leq i \leq n-m}$. Теперь посравниваем a с получившимися b-шками по модулю случайного p, как в предыдущем алгоритме, но возьмем $\tau = (n^2 m) \log(n^2 m)$.$P_{\text{ошибки}} \leq \frac{n}{\frac{\tau}{\log \tau}} \leq \frac{2}{n^2} = O(\frac{1}{n^2})$, если $a = b(i)$, проверим руками равенство данной подстроки и a, теперь вероятность ошибки равна нулю. Посмотрим за сколько наш алгоритм работает. Если выключить ручную перепроверку, очевидно, будет $O(n)$, так как мы за $O(m)$ посчитаем $\{b(i)\}$ и $n - m + 1$ раз проведем сравнение $b(i)$ и a. Есть включить, то <это я напишу когда-нибудь, когда пойму, что написано в конспекте Гирша(никогда)>.

14 (14) Рандомизированный QuickSort (Осипов Д.)

В этой главе мы строго докажем, что среднее время (матожидание времени) работы рандомизированного Quicksort есть $O(n \log n)$.

Приведем его реализацию (Cormen). Сортировка всего массива вызывается `Quicksort(A, 1, len(A)`.

Algorithm 1: Нижний текст

```
1  Partition(A, p, r):
2  i = random ∈ [p, r]
3  A[r] ↔ A[i]
4  x = A[r]
5  i = p − 1
6  for j = p...r − 1 do
7      if A[j] ≤ x then
8          i++
9          A[i] ↔ A[j]
10 A[i + 1] ↔ A[r]
11 return i + 1
12
13 Quicksort(A, p, r):
14 if p < r then
15     q = Partition(A, p, r)
16     Quicksort(A, p, q − 1)
17     Quicksort(A, q + 1, r)
```

Напомним, как работает процедура `Partition`. Она обрабатывает отрезок $[p..r]$ массива A следующим образом. В строках 2-4 случайно выбирается *опорный элемент*, который перемещается в конец отрезка и запоминается в x. В строках 5-11 элементы $[p \ldots r-1]$ меняются таким образом, что все элементы $\leq x$ расположены слева, а все элементы $> x$ справа. Строка 12 располагает x

между этими частями (немного меняя правую часть). Таким образом, отрезок $A[p..r]$ разбивается на три части: сначала идут элементы $\leq x$, потом сам x, потом $> x$. Строка 13 возвращает позицию x.

Итак, считаем среднее время. За X обозначим случайную величину – общее число сравнений, произведенных на строке 7, за все время выполнения `Quicksort(A, 1, len(A))`. Из строк 18-19 видно, что каждый вызов `Quicksort` «убирает» из работы один элемент массива, поэтому всего вызовов `Partition` не более n. Каждый вызов `Partition` совершает $O(1)$ действий плюс какое-то количество сравнений, так что суммарное время работы `Quicksort(A, 1, len(A))` есть $O(n+X)$.

Нам нужно вычислить матожидание общего числа сравнений $\mathbb{E}X$. Переименуем элементы A как $z_1 \leq \ldots \leq z_n$.

Заметим, что любая пара элементов сравнивается не более одного раза. Действительно, при любом сравнении, как видно из строки 7, один из двух сравниваемых элементов – опорный, и после окончания цикла (строка 6) этот опорный элемент «выпадает» из работы и более ни с чем не сравнивается. Так что введем случайную величину $X_{ij} = [z_i, z_j \text{ когда-то сравнивались}]$. Ясно, что тогда:

$$\mathbb{E}X = \mathbb{E}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij} = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathbb{E}X_{ij} = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathbb{P}[z_i, z_j \text{ когда-то сравнивались}]$$

.

Осталось подсчитать $\mathbb{P}[z_i, z_j \text{ когда-то сравнивались}]$. Нужно выяснить, в каком случае z_i и z_j сравнятся, а в каких нет.

Для примера рассмотрим массив, содержащий в каком-то порядке числа $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Пусть первым опорным элементом стала 7. Во-первых, это означает, что на данном этапе 7 сравнится со всеми остальными числами, а далее «выпадет» и ни с чем сравниваться не будет. Во-вторых, множество разбивается на две части $\{1, 2, 3, 4, 5, 6\}$ и $\{8, 9, 10\}$ в том смысле, что все дальнейшие сравнения будут происходить **только** внутри этих частей. Например, 2 и 9 точно не сравнятся, а 2 и 4 могут сравниться (если не попадут в разные части на какой-нибудь из следующих итераций).

В общем случае всё обстоит так: если какой-то элемент x ($z_i \leq x \leq z_j$) был опорным до того, как опорными стали z_i и z_j, то z_i и z_j не сравнятся. И наоборот – если ни один из элементов $z_i, z_{i+1}, \ldots, z_j$ не стал опорным до того, как опорным стал z_i или z_j, то z_i и z_j сравнятся.

Итак, можно видеть, что z_i и z_j сравнятся тогда и только тогда, когда среди элементов $z_i, z_{i+1}, \ldots, z_j$ раньше всех опорным элементом станет либо z_i, либо z_j. Вероятность этого равна $\frac{2}{j-i+1}$.

Имеем:

$$\mathbb{E}X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Замена переменных во внутренней сумме $k = j - i$ дает:

$$= \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n} O(\log n) = O(n \log n)$$

Таким образом, матожидание времени работы `Quicksort` есть $O(n + n\log n) = O(n \log n)$. $\square$

15 (15) Проверка равенства полиномов. Лемма Шварца-Циппеля. (Ермошин И.)

Лемма Шварца-Зиппеля. $0 \neq p \in \mathbb{Z}[x_1, \dots, x_m]$, *все* $\deg x_i \leq d, A \subset \mathbb{Z}, |A| < \infty$. *Тогда*

$$|\{(i_1, \dots, i_m) \in A^m : p(i_1, \dots, i_m) = 0\}| \leq md|A|^{m-1}$$

Докажем индукцией по количеству переменных: при $m = 1$, очевидно, корней $\leq d$.

Переход: Очевидно, можно написать $p(x_1, \dots, x_m) = x_m^d \cdot y_d + \dots + x_m^0 \cdot y_0$, где $\{y_i\} \subset \mathbb{Z}[x_1, \dots, x_{m-1}]$. Оценим число корней $x = (x_1, \dots, x_m)$, рассмотрев два случая:

1) x обнуляет y_d. Таких корней $\leq (m-1)d|A|^{m-2} \cdot |A|$ (выберем $x_1, \dots, x_{m-1}$ по предположению индукции $\leq (m-1)d|A|^{m-2}$ способами и возьмем в качестве x_m любое число из A).

2) x не обнуляет y_d. Тогда при подстановке $x_1, \dots, x_{m-1}$ получится ненулевой многочлен от x_m. У него корней не больше, чем d. Оценим грубо: всего наборов $(x_1, \dots, x_{m-1})$ значений $|A|^{m-1}$ штук, значит корней $\leq d|A|^{m-1}$.

Итого $\leq (m-1)d|A|^{m-1} + d|A|^{m-1} = md|A|^{m-1}$. □

Задача. *Даны два многочлена p_1 и p_2. Выяснить, равны ли они.*

Считается, что многочлены достаточно большие и даны в таком виде, что приведение к каноническому виду $p = a_d x^d + \dots + a_0$ очень затруднено, но вычисление значений многочленов в точках возможно. Например, если многочлен задан разложением на множители $p = (x - x_1) \dots (x - x_d)$, для раскрытия скобок и приведения подобных нужно сделать $O(2^d)$ действий, но вычислить значение в точке можно за $O(d)$.

Алгоритм таков. За полином от длины вычислим m – количество переменных, $d = \max \deg x_i$. Зафиксируем некоторое конечное $A \subseteq \mathbb{Z}$. Выберем случайно $x \in A^m$ и вычислим значения $p_1(x), p_2(x)$. Если $p_1(x) = p_2(x)$, то ответим «многочлены совпадают», иначе – «многочлены не совпадают». Описание алгоритма закончено.

Ясно, что если $p_1 = p_2$, то алгоритм об этом и сообщит. Посчитаем вероятность ошибки: когда $p_1 \neq p_2$, но $p_1(x) = p_2(x)$. По лемме знаем, что у $p_1 - p_2$ не более $md|A|^{m-1}$ корней, таким образом, вероятность попасть в корень не превосходит:

$$\leq \frac{md|A|^{m-1}}{|A|^m} = \frac{md}{|A|}$$

Ясно, что это и есть вероятность $p_1(x) = p_2(x)$ при $p_1 \neq p_2$. □

NB: Размер выбранного конечного A влияет на вероятность ошибки, но не влияет на время работы алгоритма. Таким образом, требуемая малость вероятности ошибки может достигаться не только повторением алгоритма, но и размером A.

16 (17) Хеш-таблицы. Универсальные семейства хеш-функций. (Осипов Д.)

Задача. *Реализовать структуру – множество, поддерживающее операции вставки (Insert), удаления (Delete), поиска (Search).*

16.1 Прямая адресация

Считаем, что элементы нашего множества – целые числа в диапазоне $[0, m-1]$.

Решение (очевидное – прямая адресация): *всегда на всё O(1) времени, O(m) памяти.* Заводим булевый массив A из m нулей.

- Вставить k – пометить A[k] $= 1$,
- Удалить k – пометить A[k] $= 0$,
- Найти k – посмотреть A[k].

□

16.2 Хеш-таблица с чеинингом

Если так случилось, что элементами множества могут быть не все числа $\{0, \ldots, m-1\}$, а лишь элементы некоторого $K \subseteq \{0, \ldots, m-1\}$, $|K| = n$, то при большом m и маленьком n будет бесполезно потрачено много памяти.

Пусть теперь K – произвольное множество натуральных чисел, m – натуральный параметр.

Решение (хеш-таблица с чеинингом).

Предположим, что выбрана некоторая *хеш-функция*

$$h : K \to \{0, \ldots, m-1\},$$

вычислимая за $O(1)$. Наше множество будем хранить в массиве двусторонних списков $T[0..m-1]$. Операции реализуем так:

- Вставить k – положить k в начало списка $T[h(k)]$.
- Найти k – просмотреть весь список $T[h(k)]$.
- Удалить k – найти k в списке $T[h(k)]$ и удалить его, если нашелся.

Описание алгоритма закончено.

16.3 Гипотеза простого равномерного хеширования: оценки (кажется, не входит в экз?????)

Для работоспособности алгоритма функция h может быть совершенно любой, но желательно, чтобы хеш-коды $\{h(k)\}_{k \in K}$ распределялись равномерно. Для этого предположим, что:

1. для каждого $k \in K$ значение $h(k)$ является случайной величиной,
2. $\mathbb{P}\{h(k) = r\} = \frac{1}{m}$ для всех $r \in \{0, \ldots, m-1\}$,
3. для всех $k_1 \neq k_2$ величины $h(k_1)$ и $h(k_2)$ независимы.

Эти предположения составляют *гипотезу простого равномерного хеширования.*

Сейчас мы покажем, что в этой модели операция поиска выполняется за $O\left(\frac{n}{m} + 1\right)$.[11]

Теорема. *При гипотезе простого равномерного хеширования средняя длина списка есть* $\frac{n}{m}$.

Доказательство. Занумеруем $K = \{k_1, ..., k_n\}$. Фиксируем $j \in \{0, \ldots, m-1\}$. Для $i = 1..n$ определим случайную величину

$$X_{ij} = [h(k_i) = j] \text{ – попал ли элемент } k_i \text{ в ячейку } j.$$

Из пункта 2) ясно, что $\mathbb{E}X_{ij} = \frac{1}{m}$. Также ясно, что длина списка $T[j]$ есть $X_{1j} + ... + X_{nj}$. Матожидание этой величины есть

$$\mathbb{E}\sum_{i=1}^{n} X_{ij} = \sum_{i=1}^{n} \mathbb{E}X_{ij} = \frac{n}{m}.$$

[11] Стоит понимать как O(1) в случае $\frac{n}{m} \leq 1$ и $O\left(\frac{n}{m}\right)$ иначе.

□

Далее нужно рассмотреть два случая: искомый элемент k_i есть в списке $T[h(k_i)]$ *(успешный поиск)*, либо же его нет *(неудачный поиск)*.

Теорема. *Среднее время работы неудачного поиска есть* $O\left(\frac{n}{m}+1\right)$

Доказательство. В случае, если элемента k в списке $T[h(k)]$ нет, то алгоритм просматривает весь список длины $\frac{n}{m}$. □

Теорема. *Среднее время работы успешного поиска есть* $O\left(\frac{n}{m}+1\right)$.

Доказательство. Тут придется повозиться. Предположим, что $k_1, ..., k_n$ занумерованы в порядке добавления их в множество. Фиксируем k_i и считаем среднее время поиска его в списке $T[h(k_i)]$. Для $j = 1..m$ определим случайную величину

$$X_{ij} = [h(k_i) = h(k_j)] \text{ – «} k_i \text{ в одном списке с } k_j \text{».}$$

Ясно, что $\mathbb{E}X_{ii} = 1$ и $\mathbb{E}X_{ij} = \frac{1}{m}$ при $j \neq i$.

Алгоритм ищет k_i в списке $T[h(k_i)]$. Сколько элементов он пройдет, прежде чем наткнется на k_i? Он пройдет те элементы k_j, которые лежат в $T[h(k_i)]$ (т.е. $h(k_i) = h(k_j)$) и которые находятся в этом списке раньше k_i (т.е. $j \geq i$ – ведь каждый новый элемент добавляется в начало списка). Поэтому количество пройденных элементов равно

$$X_{in} + ... + X_{ii}.$$

Матожидание времени поиска фиксированного k_i равно

$$\mathbb{E}\sum_{j=i}^{n} X_{ij} = \mathbb{E}X_{ii} + \sum_{j=i+1}^{n} \mathbb{E}X_{ij} = 1 + \frac{n-i}{m}$$

А если взять среднее по $i = 1..n$, получаем:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right) = 1 + \frac{n-1}{2m} = O\left(\frac{n}{m}+1\right)$$

□

16.4 Универсальное семейство хеш-функций: оценки

Определение. *Универсальное семейство хеш-функций* $\mathcal{H}$ *– такое множество хеш-функций, что для любой пары различных ключей* k_1, k_2 *количество функций* $h \in \mathcal{H}$ *таких, что* $h(k_1) = h(k_2)$, *не превосходит* $\frac{|\mathcal{H}|}{m}$.

NB: Другими словами, при случайном равновероятном выборе функции из $\mathcal{H}$ вероятность того, что для фиксированной пары различных ключей k_1, k_2 случится коллизия $h(k_1) = h(k_2)$, не превосходит $\frac{1}{m}$.

Если у нас есть такое семейство, то перед началом работы мы выбираем из него случайно и равномерно одну функцию, по ней строим хеш-таблицу с чеинингом.

Оказывается, что в этой модели среднее время поиска остается тем же самым.

Теорема. *В модели универсального хеширования среднее время работы как неудачного, так и успешного поиска есть* $O\left(\frac{n}{m}+1\right)$.

Доказательство. Для элементов k и l обозначим:

$$X_{kl} = [h(k) = h(l)]$$

Длина цепочки $T[h(l)]$, в которой, возможно, лежит l, есть $X_{k_1\,l} + ... + X_{k_n\;l}$.

Если l не лежит в этой цепочке, то матожидание этой суммы не превосходит $\frac{n}{m}$, так как матожидание каждого слагаемого не превосходит $\frac{1}{m}$ (см. ***NB*** выше).

Если же l лежит в этой цепочке, то матожидание суммы не превосходит $1 + \frac{n-1}{m}$: матожидание слагаемого X_{ll} равно 1, матожидание любого другого слагаемого $\leq \frac{1}{m}$.

Получается, что и в том, и в другом случае число шагов оценивается сверху как $O\left(\frac{n}{m} + 1\right)$. □

16.5 Универсальное семейство хеш-функций: построение

Осталось построить какое-нибудь универсальное семейство хеш-функций. Это несложно. Опишем для числового множества.

Выберем простое $p > m$. Для всяких целых $a \in \{1, \dots, p-1\}$ и $b \in \{0, \dots, p-1\}$ положим

$$h_{ab}(k) = ((ak + b) \mod p) \mod m$$

Теорема. *$\mathcal{H} = \{h_{ab}\}$ универсально.*

Доказательство. Доказательство счётом. Убедитесь самостоятельно в следующем:

1. Фиксируем различные $k_1, k_2 \in \{0, \dots, m-1\}$. Обозначим $t_i = (ak_i + b) \mod p$ для $i = 1, 2$. Докажите, что $t_1 \neq t_2$. Напоминание: $a \neq 0$ и $k_1, k_2 < m < p$.

2. Фиксируем различные $k_1, k_2 \in \{0, \dots, m-1\}$ и различные $t_1, t_2 \in \{0, \dots, p-1\}$. Докажите, что существуют и единственные $a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\}$, что $t_i = (ak_i + b) \mod p$ для $i = 1, 2$ (вычислите эти t_1, t_2).

Следовательно, каждая пара (t_1, t_2), где $t_1 \neq t_2$, при фиксированных $k_1 \neq k_2$ реализуема ровно одним способом. Так что если выбирать хеш-функцию h_{ab} случайно и равновероятно из $\mathcal{H}$, то для нее пара (t_1, t_2) окажется так же равновероятна среди всех пар $\{(t_1, t_2) \colon t_1 \neq t_2\}$.

Теперь вероятность того, что $h_{ab}(k_1) = h_{ab}(k_2)$, равна вероятности, с которой различные равновероятно выбранные $t_1, t_2 \in \{0, \dots, p-1\}$ совпадут по модулю m.

3. Докажите, что для фиксированного $t_1 \in \{0, \dots, p-1\}$ количество таких $t_2 \in \{0, \dots, p-1\}$, что $t_1 \neq t_2$ и $t_1 \equiv t_2 (\mod m)$, не превосходит $\frac{p-1}{m}$.

Тогда при фиксированном t_1 вероятность выбрать $t_2 \neq t_1$ т.ч. $t_1 \equiv t_2 (\mod m)$ не превосходит

$$\frac{1}{p-1} \cdot \frac{p-1}{m} = \frac{1}{m}.$$

Нетрудно видеть, что эта оценка сверху справедлива и для успешного равновероятного выбора пары (t_1, t_2). □

17 (20) Слабоэкспоненциальные детерминированные алгоритмы SAT для 3-КНФ (Осипов Д.)

17.1 Начальные сведения

Задача (SAT) *Для данной пропозициональной формулы от n переменных определить, выполнима ли она.*

Решение за $O(2^n)$**.** Переберем все 2^n возможных наборов значений переменных. $\square$

Факт. *Задача SAT NP-трудна: любую задачу из NP можно свести к SAT. Научимся решать SAT за полином* $\implies$ *научимся решать любую NP-задачу за полином и получим* $P = NP$*. Докажем, что SAT не решается за полином* $\implies$ *автоматически* $P \neq NP$*.*

Факт. *SAT сводится к 3-SAT.*

Задача (3-SAT) *Пусть дана пропозициональная формула от n переменных в 3-КНФ (каждый конъюнкт содержит не более трех слагаемых). Определить, выполнима ли она.*

17.2 Метод расщепления: $O(1.92^n)$, $O(1.84^n)$

Решение за $O\left(\sqrt[3]{7}^n\right) = O(1.92^n)$ ***(метод расщепления-1).***

Рекурсивный алгоритм. Выделим один из конъюнктов

$$\ldots \wedge (x_1^{\sigma_1} \vee x_2^{\sigma_2} \vee x_3^{\sigma_3}) \wedge \ldots$$

Из всех восьми возможных наборов значений x_1, x_2, x_3 конкретно под этот конъюнкт подходят только семь – все, кроме $(x_1, x_2, x_3) = (\bar{\sigma_1}, \bar{\sigma_2}, \bar{\sigma_3})$.

Для каждого из семи наборов значений делаем следующее: подставляем его в формулу и запускаем алгоритм рекурсивно на получившейся формуле от $n-3$ переменных.

Время работы определяется соотношением $T(n) = 7T(n-3) + O(1)$, откуда немедленно $T(n) = O(7^{n/3})$ $\square$.

Здесь нужна картинка

Решение за $\sim O(1.84^n)$ ***(метод расщепления-2)*** .

Снова рекурсивный алгоритм. Выделим один из конъюнктов

$$\ldots \wedge (x_1^{\sigma_1} \vee x_2^{\sigma_2} \vee x_3^{\sigma_3}) \wedge \ldots$$

Рекурсивно рассмотрим три случая, когда этот конъюнкт может быть истинен:

1. либо $x_1 = \sigma_1$,
2. либо $x_1 = \neg\sigma_1$ и $x_2 = \sigma_2$,
3. либо $x_1 = \neg\sigma_1$, $x_2 = \neg\sigma_2$ и $x_3 = \sigma_3$

Для каждого из этих случаев сделаем подстановку и рекурсивно решим подзадачи: для формул от $n-1$, $n-2$ и $n-3$ переменных соответственно.

Время работы описывается соотношением $T(n) = T(n-1) + T(n-2) + T(n-3) + O(1)$. $T(n) = O(1.84^n)$ – его приближенное решение. $\square$

Здесь нужна картинка

17.3 Метод локального поиска: $O(1.74^n)$

Следующее решение основано на методе «локального поиска». Зададим на множестве векторов $\{0,1\}^n$ метрику $d(x,y)$ = количество позиций, в которых x и y различны. Для данного вектора x и натурального r определим шар $H(x,r)$ – множество векторов, отличающихся от x не более, чем в r позициях.

Нам понадобится следующая *вспомогательная задача*.

Вспомогательная задача. *Дан вектор* $x \in \{0,1\}^n$ *и натуральный радиус* r*. Проверить, есть ли в шаре* $H(x,r)$ *выполняющий набор для данной 3-КНФ формулы.*

Решение вспомогательной задачи за $O(3^r)$***.*** Рекурсивный алгоритм. Сначала проверим формулу на наборе x. Если в нем формула не выполнена, выделим в ней любой ложный конъюнкт $(x_a^{\sigma_a} \vee x_b^{\sigma_b} \vee x_c^{\sigma_c})$. Если в $H(x, r)$ присутствует выполняющий набор x^*, то x^* **не** совпадает с x хотя бы в одной из позиций a, b, c. Рассмотрим три набора $x^{(a)}, x^{(b)}, x^{(c)}$, каждый из которых получается из x инвертированием a-той, b-той и c-той переменной соответственно. Хотя бы один из наборов $x^{(a)}, x^{(b)}, x^{(c)}$ будет на единицу ближе к x^* (ведь изменилась всего одна переменная). Запустим от каждого из них этот алгоритм рекурсивно. Тогда на глубине рекурсии, не превосходящей r, набор x^* найдется, если он есть в $H(x, r)$. Очевидно, решение работает за $O(3^r)$. □

Теперь мы готовы решать нашу задачу $3 - SAT$.

Решение за $O\left(\sqrt{3}^n\right) = O(1.74^n)$ ***(локальный поиск).***

Обозначим $\mathbf{0} = (0, \ldots, 0)$ и $\mathbf{1} = (1, \ldots, 1)$ – вектора в $\{0,1\}^n$. Заметим, что всё пространство $\{0,1\}^n$ покрывается двумя шарами $H(\mathbf{0}, n/2)$ и $H(\mathbf{1}, n/2)$. Действительно, каждый вектор длины n имеет либо хотя бы $n/2$ единиц, либо хотя бы $n/2$ нулей, откуда следует требуемое. Значит, достаточно за $O(3^{n/2})$ поискать выполняющий набор в каждом из двух шаров. Итоговое время работы $O(3^{n/2}) + O(3^{n/2}) = O(3^{n/2})$. □

18 (21) Алгоритм Шоннинга для 3-SAT, использующий случайное блуждание (Осипов Д.)

Условие задачи все еще `в том билете`.

Мы предъявим вероятностное решение *с односторонней ограниченной вероятностью ошибки* (такое было `здесь`).

Вероятностное решение (Schöning, 1999), время $O(n^2(4/3)^n)$***, шанс ошибки*** $\leq 1/2$

Алгоритм описывается даже проще, чем предыдущие. В начале мы берем случайный $x \in \{0,1\}^n$. Повторим не более n раз следующее: если x не выполняет формулу, то возьмем в ней случайный ложный конъюнкт, случайно выберем **одну** переменную в нем и изменим ее значение. Описание алгоритма закончено.

Оценим снизу вероятность того, что этот алгоритм найдет выполняющий набор x^*. Не уменьшая общности, предположим, что выполняющий набор x^* существует и единственный. Заметим тогда (по `рассуждению из билета 20`), что при каждой итерации цикла x становится ближе к x^* с вероятностью $\geq 1/3$ и дальше от x^* с вероятностью $\leq 2/3$. Поэтому, не уменьшая общности, еще предположим, что вероятности приближения и отдаления – **ровно** $1/3$ и $2/3$ соответственно.

Итак, вероятность того, что x совпадет с x^*, моделируется следующей задачей на случайное блуждание по отрезку $[0, N]$. x начинает свой путь в некоторой точке этого отрезка, делает шаг влево с вероятностью $1/3$, вправо – с $2/3$ (и все время остается в отрезке $[0, N]$), и необходимо оценить вероятность того, что в течение n шагов он когда-нибудь посетит 0.

Не уменьшая общности, для того, чтобы x посетил 0 в течение n шагов, **достаточно** (конечно, не необходимо) два условия:

1. Случайно выбранный в начале алгоритма x оказался x^* на расстоянии $n/3$ от x^*
2. За n шагов из точки $n/3$ он придет в 0, совершив $2n/3$ шагов влево и $n/3$ шагов вправо, не выходя при этом за границу отрезка 0.

Сейчас мы посчитаем вероятности этих двух событий, их произведение и будет оценкой снизу на вероятность того, что алгоритм найдет выполняющий набор.

Вероятность первого события равна $P_1 = \frac{\binom{n}{n/3}}{2^n}$ так как из 2^n равновероятных наборов $\in \{0,1\}^n$ мы должны выбрать тот, у которого ровно $n/3$ позиций, в которых он и x^* различаются.

Для подсчета вероятности второго события воспользуемся следующей задачей.

Теорема (задача о пьянице и канаве). *Сколько существует путей из точки $S = P - Q > 0$ в точку 0, состоящих ровно из P шагов влево, Q шагов вправо и не выходящих за точку 0? Ответ: $\frac{P-Q}{P+Q}\binom{P+Q}{P}$.*

Доказательство задачи можно найти `здесь`. $\square$

В нашем случае количество шагов влево $P = 2n/3$, вправо $Q = n/3$, так что всего таких путей $\frac{1}{3}\binom{n}{n/3}$. Для фиксированного пути с P шагами влево и Q шагами вправо вероятность, что x пройдет именно его, равна $(1/3)^P(2/3)^Q = (1/3)^{2n/3}(2/3)^{n/3}$, так что:

$$P_2 = \frac{1}{3}\binom{n}{n/3}(1/3)^{2n/3}(2/3)^{n/3}$$

Символом $\overset{c}{\sim}$ будем обозначать «эквивалентность с точностью до константы», т.е:

$$[f \overset{c}{\sim} g] \iff [\exists C > 0 : \ f \sim Cg]$$

С помощью формулы Стирлинга $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \overset{c}{\sim} \sqrt{n}\left(\frac{n}{e}\right)^n$ можно убедиться, что:

$$P_1 \overset{c}{\sim} \frac{1}{\sqrt{n}}\left(\frac{3}{2^{5/3}}\right)^n$$
$$P_2 \overset{c}{\sim} \frac{1}{\sqrt{n}}\left(\frac{1}{2^{1/3}}\right)^n$$

И поэтому вероятность успеха асимптотически хотя бы

$$P \geq P_1 \cdot P_2 \overset{c}{\sim} \frac{1}{\sqrt{n}}\left(\frac{3}{2^{5/3}}\right)^n \cdot \frac{1}{\sqrt{n}}\left(\frac{1}{2^{1/3}}\right)^n = \frac{1}{n}\left(\frac{3}{4}\right)^n$$

Однако этот алгоритм работает за $O(n)$ времени! Его можно повторить много раз, увеличивая шансы на успех. В частности, если повторить его $n\left(\frac{4}{3}\right)^n = L$ раз, то имеем вероятность неудачи:

$$\left(1 - \frac{1}{L}\right)^L \leq \frac{1}{e} \leq \frac{1}{2}$$

Что и приводит нас к требуемому результату. $\square$

NB: *А если повторить в q раз больше, то есть $qn\left(\frac{4}{3}\right)^n$ раз, то вероятность неудачи $\leq \left(\frac{1}{2}\right)^q$ можно выбрать сколь нужно малой.*