

Алгосы, часть ii

Денис Осипов, Иван Ермошин, Alexander Grothendieck

15 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: 15 августа 2020 г.

Содержание

1	Параллельные алгоритмы – i (Осипов Д.)	3
1.1	Булевы схемы как модель параллельного алгоритма	3
1.2	Принцип Брента	3
1.3	Параллельное умножение булевых матриц	4
1.4	Параллельная достижимость в графе	5
2	Параллельные алгоритмы – ii (Осипов Д.)	5
2.1	Параллельное вычисление всех префиксных сумм	5
2.2	Параллельное сложение чисел	6
2.3	Параллельное умножение чисел	6
3	Параллельные алгоритмы – iii (Осипов Д., Grothendieck А.)	7
3.1	(В РАЗРАБОТКЕ) Параллельное вычисление всех расстояний до конца списка . . .	7
3.2	(В РАЗРАБОТКЕ) Параллельное вычисление всех глубин дерева	8
4	Приближенный алгоритм для задачи о рюкзаке (Осипов Д.)	8
5	Set Cover - i (Осипов Д.)	9
5.1	Сведение к задаче линейного программирования	9
5.2	Следствие для задачи вершинного покрытия (Vertex Cover)	11
5.3	Двойственная задача	11
5.4	Прямо-двойственный метод	13
6	Set Cover – ii (Осипов Д.)	14
6.1	Жадный приближенный алгоритм	14

7	(WIP)Транспортные сети. Задача о максимальном потоке. Разрез. Теорема о максимальном потоке и минимальном разрезе. Алгоритм Форда-Фалкерсона (Grothendieck A.)	15
7.1	Транспортные сети. Задача о максимальном потоке	15
7.2	Разрез. Теорема о максимальном потоке и минимальном разрезе	17
7.3	Алгоритм Форда-Фалкерсона	18
7.4	Применение к паросочетаниям	19
8	(WIP)Алгоритм Эдмондса-Карпа (Grothendieck A.)	20
9	(WIP)Алгоритм проталкивания предпотока (Grothendieck A.)	21
9.1	Интуитивные соображения	21
9.2	Операция проталкивания	22
9.3	Операция подъема	22
9.4	Начальный предпоток	23
9.5	Алгоритм. Его корректность.	23
10	Вероятностные алгоритмы с односторонней ограниченной вероятностью ошибки. Алгоритм Фрейвальдса для проверки умножения матриц. (Ермошин И.)	24
11	Слабоэкспоненциальные детерминированные алгоритмы SAT для 3-КНФ (Осипов Д.)	25
11.1	Начальные сведения	25
11.2	Метод расщепления: $O(1.92^n)$, $O(1.84^n)$	25
11.3	Метод локального поиска: $O(1.74^n)$	26
12	Алгоритм Шоннинга для 3-SAT, использующий случайное блуждание (Осипов Д.)	26

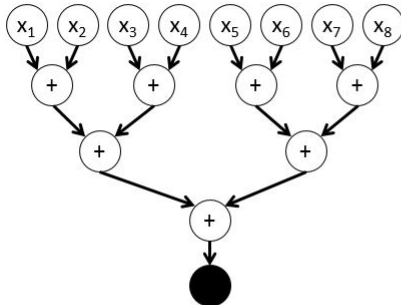
1 Параллельные алгоритмы – i (Осипов Д.)

Параллельный алгоритм – предназначенный для исполнения на нескольких процессах.

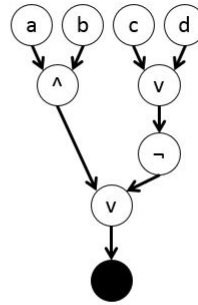
1.1 Булевы схемы как модель параллельного алгоритма

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 $(a \wedge b) \vee \neg(c \vee d)$
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. ■

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.

1.2 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 $W = \sum_{i=1}^t w_i$ и назовем эту величину общей работой алгоритма. Тогда алго-

ритм 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: булевы – 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 $C_{ij} = \sum_{k=1}^n A_{ik}B_{kj}$. Сделаем это за $\log n$ шагов (суммирование бинарным деревом). Задача решена за $1 + \log n = O(\log n)$ шагов на n^3 процессорах. ■

NB: Сложить¹ 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 .

Вместо второго решения, 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 $\frac{W(n)}{P(n)} = O(t(n)) \implies P(n) = \frac{W(n)}{t(n)} = O\left(\frac{n^3}{\log n}\right)$

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

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 Параллельные алгоритмы – ii (Осипов Д.)

2.1 Параллельное вычисление всех префиксных сумм

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

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

Заведём 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:

```
function PrefixSum ( & A[0..n-1] ):
if n > 1:
    B = [0]*(\frac{n}{2}-1)
    parallel for i = 0 to \frac{n}{2}-1:
        B[i] = A[2i] + A[2i+1]
    PrefixSum (B)

    parallel for i = 0 to n-1:
        if i > 0:
            if i%2 == 0:
                A[i] = B[\frac{i}{2}-1] + A[i]
            else:
                A[i] = B[\frac{i-1}{2}]
```

Время работы оценивается просто: из кода следует соотношение $T(n) = T(n/2) + C$, и далее можно написать $= T(n/4) + 2C = T(n/8) + 3C = \dots = C \cdot \log n = O(\log n)$. Общая работа:

$W(n) = W(n/2) + O(n)$, откуда по мастер-теореме $W(n) = O(n)$. По принципу Брента количество процессоров можно взять $P(n) = \frac{W(n)}{T(n)} = O(\frac{n}{\log n})$. ■

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 $O(n)$. 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 было бы вычислить по формуле² $c_i = (a_i + b_i + z_{i-1}) \% 2$.

Положим:

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

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

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

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

то имеем $z_k = v_{k1} \vee v_{k2} z_{-1} = v_{k1}$. Но вектора (v_{k1}, v_{k2}) суть просто префиксные «суммы» последовательности $(0, 0), (g_1, p_1), \dots, (g_n, p_n)$ 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 $O(\frac{n}{\log n})$. ■

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 $c = \sum_{i=0}^{2n} c_i 2^i$.

Решение за $O(\log n)$. Ясно, что $ab = \sum_{i=0}^n a b_i 2^i = \sum_{i: b_i=1}^n a 2^i$. Таким образом мы свели умножение двух чисел к сложению не более чем n чисел. Но на этом не все.

²Пологая $z_{-1} = 0$ по определению

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 $O(1)$ времени³ 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 $T(n) = T(\frac{n}{3/2}) + O(1)$ 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 $O(\frac{n^2}{\log n})$. ■

3 Параллельные алгоритмы – iii (Осипов Д., Grothendieck А.)

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

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

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 концевой i ($\text{next}[i] == \text{nil}$) 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 $a_{\text{next}[i]}$ в исходном списке.

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

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

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 ?????. ■

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

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

3.2 (В РАЗРАБОТКЕ) 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 $O(\log n)$ независимо от высоты дерева. Сопоставим каждой вершине i три процессора A_i, B_i, C_i . Перестроим дерево в ориентированный граф, вершины которого будут этими процессорами. Именно, проведем ребро:

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

Можно проверить⁵, что у этого графа существует эйлеров обход, начинающийся в A корня и заканчивающийся в C корня.

?????

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 .

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

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

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 $\varepsilon > 0$. Заменяем все v_i на:

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

⁵Например, 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 $in(v_2) = out(v_2) - 1$.

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

Оценим время работы. $V = \sum v_i \leq n \cdot \frac{n}{\varepsilon} = \frac{n^2}{\varepsilon}$. 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 $K^* = \sum_{i \in S} v_i$.

С точки зрения новой задачи сумма этого набора оценивается как:

$$\sum_{i \in S} \hat{v}_i = \sum_{i \in S} \left\lfloor \frac{v_i n}{\varepsilon v_{\max}} \right\rfloor \geq \sum_{i \in S} \left(v_i \cdot \frac{n}{\varepsilon v_{\max}} - 1 \right) \geq K^* \frac{n}{\varepsilon v_{\max}} - n$$

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

$$\sum_{i \in \hat{S}} \hat{v}_i \geq \sum_{i \in S} \hat{v}_i \geq K^* \frac{n}{\varepsilon v_{\max}} - n$$

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

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

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

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:

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

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 $h = h(x_1, \dots, x_n)$ при ограничениях вида $g_k \wedge b_k$, где \wedge есть один из знаков \leq, \geq , а $g_k = g_k(x_1, \dots, x_n)$ – 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, \dots, S_m , в которые входит элемент e_j . Положим $f = \max_{i=1..n} f_j$. 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 $x = (x_1, \dots, x_m)^T$ взаимно

однозначно кодирует любой набор индексов I . Тогда целевая функция – суммарный вес покрытия – выглядит как $\sum_{j=1}^m w_j x_j$. Ограничение на то, что набор I – покрытие, записывается так: каждый элемент e_i покрыт хотя бы одним элементом I , или же что условие $\sum_{j: e_i \in S_j} x_j \geq 1$ выполнено для всех $i = 1..n$. Итак, формулировка задачи:

$$\begin{aligned} \sum_{j=1}^m w_j x_j &\rightarrow \min, \\ \sum_{j: e_i \in S_j} x_j &\geq 1, \quad i = 1..n, \\ x_j &\in \{0, 1\}, \quad j = 1..m \end{aligned}$$

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

$$\begin{aligned} \sum_{j=1}^m w_j x_j &\rightarrow \min, \\ \sum_{j: e_i \in S_j} x_j &\geq 1, \quad i = 1..n, \\ x_j &\geq 0 \quad j = 1..m \end{aligned}$$

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*).

Считаем, что ослабленную ЛП-задачу мы решать умеем. Пусть $x^* = (x_1^*, \dots, x_m^*)^T$ – оптимальное решение ослабленной ЛП-задачи, т.е. $Z = \sum_{j=1}^m w_j x_j^*$. 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 $\sum_{j: e_i \in S_j} x_j^* \geq 1$. 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 $Z = \sum_{j=1}^m w_j x_j^*$). 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:

$$\begin{aligned} \sum_{j=1}^m w_j x_j &\rightarrow \min, \\ \sum_{j: e_i \in S_j} x_j &\geq 1, \quad i = 1..n, \\ x_j &\geq 0 \quad j = 1..m \end{aligned}$$

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

$$w^T x \rightarrow \min$$

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 $\sum_{j: e_i \in S_j} x_j \geq 1$ 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 $w^T x \rightarrow \min, \mathcal{E}x \geq \mathbb{I}_n, x \geq \mathbb{O}_m$.

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

$$\begin{aligned} \mathbb{I}_n^T y &\rightarrow \max, \\ \mathcal{E}^T y &\leq w, \\ y &\geq \mathbb{O}_m \end{aligned}$$

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:

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

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

$$\begin{aligned} \sum_{j=1}^n y_j &\rightarrow \max, \\ \sum_{i: e_i \in S_j} y_i &\leq w_j, \quad j = 1..m, \\ y_i &\geq 0, \quad i = 1..n \end{aligned}$$

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 $y^* = (y_1^*, \dots, y_n^*)^T$ – оптимальное решение двойственной ЛП-задачи. Сконструируем из нее решение I' исходной задачи (и задачи Set Cover) следующим образом:

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

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

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

Обозначим $\varepsilon = \min_{j: e_k \in S_j} (w_j - \sum_{i: e_i \in S_j} y_i^*) > 0$. 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 $\sum_{i: e_i \in S_j} y'_i = \varepsilon + \sum_{i: e_i \in S_j} y_i^* \stackrel{\text{def } \varepsilon}{\leq} \left(w_j - \sum_{i: e_i \in S_j} y_i^* \right) + \sum_{i: e_i \in S_j} y_i^* = w_j$
2. А для всякого $S_j \not\ni e_k$ имеем просто $\sum_{i: e_i \in S_j} y'_i = \sum_{i: e_i \in S_j} y_i^* \leq w_j$.

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^* : $\sum_{j=1}^n y'_j = \varepsilon + \sum_{j=1}^n y_j^* > \sum_{j=1}^n y_j^*$, но мы предполагали, что y^* – оптимальное решение. Противоречие.

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

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

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

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

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

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

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:

```
function PrimalDual ( $E = \{e_1, \dots, e_n\}$ ,  $S_1, \dots, S_m$ ):
 $y = [0]$  *  $n$ 
 $I = []$ 
while  $\exists e_i \notin \bigcup_{j \in I} S_j$ :
     $l =$  all indices such that  $e_i \in S_l$  and  $\varepsilon = (w_l - \sum_{k: e_k \in S_l} y_k)$  is minimal
     $y_i += \varepsilon$ 
     $I.append(l)$ 
```

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. \blacksquare

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 $H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}$, то получим:

Приближенное H_n -оптимальное решение.

Вот вполне интуитивный «жадник»:

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

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, $n = n_1 > \dots > n_{l+1} = 0$.

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

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

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

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

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

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

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

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

$$\min_{j: \hat{S}_j \neq \emptyset} \frac{w_j}{|\hat{S}_j|} \leq \min_{j \in O} \frac{w_j}{|\hat{S}_j|}$$

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

$$\min_{j=1..q} \frac{a_j}{b_j} \leq \frac{\sum_{j=1..q} a_j}{\sum_{j=1..q} b_j} \leq \max_{j=1..q} \frac{a_j}{b_j}$$

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

$$\min_{j \in O} \frac{w_j}{|\hat{S}_j|} \leq \frac{\sum_{j \in O} w_j}{\sum_{j \in O} |\hat{S}_j|}$$

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:

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

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

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

7 (WIP) Транспортные сети. Задача о максимальном потоке. Разрез. Теорема о максимальном потоке и минимальном разрезе. Алгоритм Форда-Фалкерсона (Grothendieck A.)

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 $c: V \times V \rightarrow \mathbb{N}$, 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 $f: V \times V \rightarrow \mathbb{R}$, для которой выполняются следующие свойства:

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

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

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 нулевой⁶.

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.

Чтобы правильно увеличивать членб, 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.

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 $|E_f| \leq 2|E|$.

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

⁶потому что $0 = c(u, v) \geq f(u, v) = -f(v, u) \geq -c(v, u) = 0$

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

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

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.

$$f_p(u, v) \leq c_f(p) \leq c_f(u, v) = c(u, v) - f(u, v) \leq c(u, v)$$

если $f(u, v) \geq 0$. 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 не максимального. 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 $\sum_{x \in S} \sum_{y \in S} f(x, y) = 0$.

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

□

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

Доказательство.

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

□

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 $S = \{v \in V \mid \exists p: s \rightarrow v \text{ in } G_f\}$, $T = V \setminus S$. 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 $|f| \leq c(S, T)$. 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

```
function FFA( $\langle G = \langle V, E \rangle, c, s, t \rangle$ ):
  foreach  $(u, v) \in E$ :
     $f(u, v) := 0$ 
     $f(v, u) := 0$ 
  while  $\exists p: s \rightarrow t \text{ in } G_f$ :
     $c_f(p) := \min\{c_f(u, v) \mid (u, v) \in p\}$ 
    foreach  $(u, v) \in p$ :
       $f(u, v) += c_f(p)$ 
       $f(v, u) := -f(u, v)$ 
```

На практике, понятно, он возникает в основном только с целыми числами. Проблема этого алгоритма в том, что не указано, как именно нужно искать увеличивающий путь. Если искать его неудачно⁷, то алгоритм может и зависнуть.

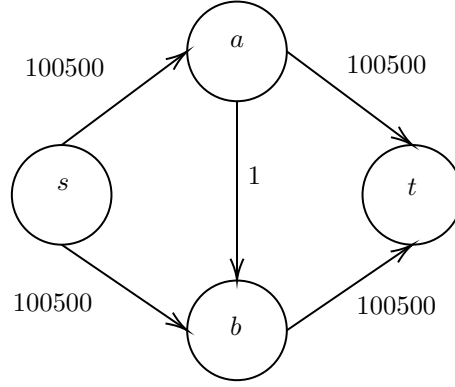
В предположении, что числа рациональные (их можно свести к целым) и при использовании поиска в глубину или поиска в ширину для нахождения увеличивающего, время его работы составляет $O(|E||f^*|)$, где f^* — максимальный поток (в случае использования поиска в ширину этот алгоритм называется **алгоритмом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

⁷а еще если значения пропускных способностей иррациональные, пример можно найти в англоязычной википедии

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 $s \rightarrow a \rightarrow b \rightarrow t$, поэтому за одну итерацию поток увеличивается всего лишь на единицу.

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

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.

Напомним, 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 , тогда 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 соответствует 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 (WIP) Алгоритм Эдмондса-Карпа (Grothendieck A.)

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 $s \rightarrow u$ 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 16), количество критических ребер будет $O(|E||V|)$.

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

9 (WIP) Алгоритм проталкивания предпотока (Grothendieck A.)

Этот алгоритм также находит максимальный поток, но отличается от предыдущих описанных алгоритмов тем, что не является вариантом алгоритма Форда-Фалкерсона, а также другой оценкой времени работы.

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

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

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

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

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

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

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

```
function push( $u, v \in V$ ):
    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 предпоток (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 $e(u) < c_f(u, v)$, то $f_{\text{new}}(v, u) = (v, u) + f_{\text{new}}(u, v) > 0 = c_f(v, u)$). 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. \square

9.3 Операция подъема

```
function relabel( $u \in V$ ):
    if  $e(u) > 0$  and  $\forall (u, v) \in E_f: h(u) \leq h(v)$ :
         $h(u) += 1 + \min_{(u, v) \in E_f} \{h(v)\}$ 
```

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 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 $h(w) \leq h(u) + 1 \leq h_{\text{new}}(u) + 1$ \square

9.4 Начальный предпоток

Начальный предпоток определяется так:

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

Начальный поток 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 .

```
function init_preflow( $G = \langle V, E \rangle, s$ ):
    foreach  $v \in V \setminus \{s\}$ :
         $h(v) := 0$ 
         $e(v) := 0$ 
    foreach  $(u, v) \in E$ :
         $f(u, v) := 0$ 
         $f(v, u) := 0$ 
     $h(s) := |V|$ 
    foreach  $(s, u) \in E$ :
         $f(s, u) := c(s, u)$ 
         $f(u, s) := -c(s, u)$ 
         $e(u) := c(s, u)$ 
         $e(s) -= c(s, u)$ 
```

9.5 Алгоритм. Его корректность.

Для начала нужно доказать лемму:

Лемма 9.3. Пусть G, s, t – транспортная сеть с предпоток 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. \square

Понятно, что они не могут быть выполнены одновременно.

Теперь мы можем написать алгоритм:

```
function PPA( $G, s, t$ ):
    init_preflow( $G, s$ )
    while  $\exists u: (\exists v \in V: \text{Pushable}(u, v)) \wedge \text{Relabelable}(u)$ :
        if  $\text{Pushable}(u, v)$ :
```

push(u, v)
else: relabel(u)

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

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

Теорема 9.1. (О корректности) Алгоритм корректен. После окончания его работы f становится максимальным потоком.

Доказательство. initialize_preflow \square

10 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 $\leq \frac{1}{2}$.

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 $\leq \frac{1}{2}$. ■

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

11.1 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. ■

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.

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

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

Рекурсивный алгоритм. Выделим один из конъюнктов

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

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 $T(n) = O(7^{n/3})$. ■

Здесь нужна картинка

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

Снова рекурсивный алгоритм. Выделим один из конъюнктов

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

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. ■

Здесь нужна картинка

11.3 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 $O(\sqrt{3}^n) = O(1.74^n)$ (локальный поиск).

Обозначим $\mathbf{0} = (0, \dots, 0)$ и $\mathbf{1} = (1, \dots, 1)$ – 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 $O(3^{n/2}) + O(3^{n/2}) = O(3^{n/2})$. ■

12 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 (Schöningg, 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**. ■

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 \simeq будем обозначать «эквивалентность с точностью до константы», т.е:

$$[f \simeq g] \iff [\exists C > 0 : f \sim Cg]$$

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

$$P_1 \simeq \frac{1}{\sqrt{n}} \left(\frac{3}{2^{5/3}}\right)^n$$

$$P_2 \simeq \frac{1}{\sqrt{n}} \left(\frac{1}{2^{1/3}}\right)^n$$

И поэтому вероятность успеха асимптотически хотя бы

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

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. ■

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.