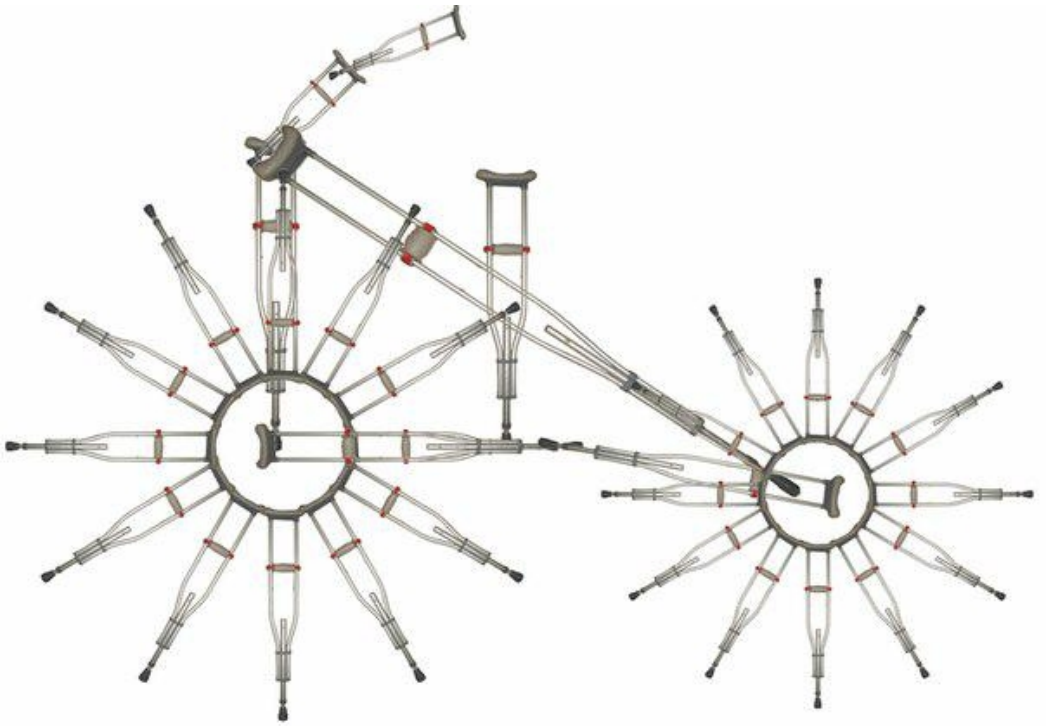


Избранные главы алгоритмов и структур данных

Конспекты лекций

Лектор: Г.О. Евстропов

Под редакцией Данилы Кутенина

НИУ ВШЭ, 2015-2016

Предисловие

Так уж вышло, что предисловие к этой книжке становится в первую очередь послесловием групп 151 и 153 к нашему лектору. Послесловие — это всегда грустный жанр, но есть одно замечательное свойство: послесловия всегда дают надежду на продолжение разговора, даже если уже много чего было сказано.

Многие из нас провели с тобой больше года, но все мы надеемся, что всё, что происходило было не зря в первую очередь для нас. Да, было много трудностей, которые и мы, и ты преодолевали все 5 модулей. Для тебя мы — первые студенты, которых ты учил. Для нас ты первый и, пожалуй, единственный лектор по алгоритмам.

Эта маленькая книжечка не отражает всё, что мы проходили, но в этом модуле были рассмотрены одновременно практически и теоретически важные алгоритмы.

Мы называли тебя по-сердитому «Глебас», но ты не обижайся, ладно? Отнесись с юмором :)

Люди, которые прошли все 5 модулей рядом с тобой:

Группа 151:

Бирюков Валентин
Воробьев Петр
Калинов Алексей
Когтенков Алексей
Корозевцев Павел
Кутенин Данила
Лазарев Владислав
Лукьянов Илья
Мельников Артем
Мусаткина Дарья
Проскуряков Александр
Смалюк Арсений
Смирнов Александр
Старченко ~~Владаимир~~ Владимир
Тульчинский Эдуард

Группа 153:

Абдумуталов Рустам
Андреев Александр
Баранов Юрий
Бесчетнов Павел
Богомолов Павел
Гурциева Тамара
Зойкин Александр
Зубанов Виктор
Капранов Иван
Кнышов Александр
Латышев Павел
Остяков Павел
Сидоров Евгений
Харламов Алексей

Глеб, читай курс всегда с таким же диким интересом и на одной волне с аудиторией, это просто топ. Чтобы через лет 10 можно было завалиться к тебе на лекцию, и слышать, что определения ещё набрасываются на вентилятор, а аутисты в аудитории никогда не спят. Спасибо тебе за бодрые пять модулей. Курс просто огонь.

Какой-то аутист

Было очень классно тебя слушать. Надеюсь, что в будущем ты защитишь кандидатскую, а каждый из нас найдёт область, которая будет востребована, чтобы она вызывала бурную реакцию для того, чтобы что-то делать и создавать. Никогда не забуду, как мне снизили за «Кутенин Д.»

Автор конспектов

Глеб! Твой курс был очень полезным для меня. Я вынес много нового и надеюсь мне это пригодится в жизни. У тебя отличная подача материала и замечательное чувство юмора. Спасибо за все пять модулей, которые ты нас терпел. Надеюсь, мы ещё встретимся :)

Андреев А.

Этот курс был чертовски полезен и крут. Пусть я и не всё понимал, но на лекциях было всегда интересно, к тому же у тебя отличное чувство юмора. Эта книжечка тебе, чтобы не терять листочки в сумке.

Иван Капранов

Содержание

Программа. Организационные моменты

Формула такая же, как и в прошлом году:

$0.3 \cdot O_{\text{контесты}} + 0.25 \cdot O_{\text{семинарские листки}} + 0.15 \cdot O_{\text{кр}} + 0.3 \cdot O_{\text{экзамен}} + \text{Б}.$

Округление вверх.

Лекция 1 от 02.09.2016. Матроиды

Пока поговорим немного о темах, не связанных с матроидами.

У нас есть конечное множество A, которое в будущем мы будем называть *носителем*. Пусть $F \subset 2^A$, и F мы будем называть *допустимыми* множествами.

Также у нас есть весовая функция $c(w)\ \forall w \in A$. Для каждого $B \in F$ мы определим *стоимость* множетсва, как $\sum\limits_{w \in B} c(w)$. Наша задача заключается в том, чтобы найти максимальный вес из всех допустимых множеств.

Пример 1 (Задача о рюкзаке)**.** *У каждого предмета есть вес и стоимость. Мы хотим унести как можно больше вещей максимальной стоимости с весом не более k.*

Вес не более k нам задает ограничение, то есть множество F. А максимизация унесенной суммы нам и задаёт задачу.

Матроид

Множество F теперь будет всегда обозначаться как I. Также условимся, что элементы отождествляются с одноэлементными множествами.

Матроидом называется множество подмножеств множества A таких, что выполняются следующие 3 свойства (аксиомы):

1. $\varnothing \in I$
2. $B \in I : \ \forall\ D \subset B \implies D \in I$
3. Если $B, D \in I$ и $|B| < |D| \implies \exists\ w \in D \setminus B$ такой, что $B \cup w \in I$

Дальнейшее обозначение матроидов — $\langle A, I \rangle$.

Определение 1. *Базой матроида называют множество всех таких элементов $B \in I$, что* **не** *существует B', что $B \subset B', |B| < |B'|$ и $B' \in I$. Обозначение $\mathfrak{B}$.*

Свойство 1. *Все элементы из базы имеют одну и ту же мощность. И все элементы из I, имеющие эту мощность, будут в базе.*

Доказательство очевидно из определения.

Пример 2 (Универсальный матроид)**.** *Это все подмножества B множества A такие, что $|B| \leqslant k$ при $k \geqslant 0$. Все свойства проверяются непосредственно.*

База такого матроида — все множества размера k.

Пример 3 (Цветной матроид)**.** *У элементов множества A имеются цвета. Тогда $B \in I$, если все элементы множества B имеют разные цвета. Свойства проверяются непосредственно, в 3 свойстве надо воспользоваться принципом Дирихле.*

База такого матроида — множества, где присутствуют все цвета.

Пример 4 (Графовый матроид на n вершинах)**.** *$\langle E, I\rangle$. Множество ребер $T \in I$, если T не содержит циклов.*

Докажем 3 свойство:

Доказательство. Пусть у нас есть T_1 и T_2 такие, что $|T_1| < |T_2|$. Разобьём граф, построенный на T_1 на компоненты связности. Так как ребер ровно $|T_1|$ на n вершинах, то компонент связности будет $n - |T_1|$. В другом случае компонент связности будет $n - |T_2| < n - |T_1|$. То есть во 2-ом графе будет меньше компонент связности, а значит по принципу Дирихле найдётся ребро, которое соединяет две компоненты связности в первом графе.

Эти рассуждения чем-то отдаленно напоминают алгоритм Краскала. □

Базой в таком матроиде являются все остовные леса.

Пример 5 (Матричный матроид)**.** *Носителем здесь будут столбцы любой фиксированной матрицы. I — множество всех подмножеств из линейно независимых столбцов. Все свойства выводятся из линейной алгебры (третье из метода Гаусса, если быть точным).*

Пример 6 (Трансверсальный матроид)**.** *$G = \langle X, Y, E\rangle$ — двудольный граф с долями X, Y. Матроид будет $\langle X, I\rangle$ такой, что $B \in I$, если существует паросочетание такое, что множество левых концов этого паросочетания совпадает с B.*

Докажем 3 свойство:

Доказательство. Пусть есть 2 паросочетания на $|B_1|$ и $|B_2|$ ($|B_1| < |B_2|$) вершин левой доли. Тогда рассмотрим симметрическую разность этих паросочетаний. Так как во 2-ом паросочетании ребер больше, то существует чередующаяся цепь, а значит при замене ребер на этой чередующейся цепи с новой добавленной вершиной (а она найдётся по принципу Дирихле) получим паросочетание с ещё 1 добавленной вершиной. □

Базой в таком матроиде будут вершины левой доли максимального паросочетания.

Приводимость одной базы к другой

Лемма 1. *Пусть $B, D \in \mathfrak{B}$. Тогда существует последовательность $B = B_0, B_1, \ldots, B_k = D, B_i \in \mathfrak{B}$ такие, что $|B_i \triangle B_{i+1}| = 2$, где $\triangle$ обозначает симметрическую разность множеств.*

Доказательство. Будем действовать по шагам. Если текущее $B_i \neq D$, тогда возьмём произвольный элемент w из $B_i \setminus D$. Тогда по 2-ому пункту определения матроида следует, что $B_i \setminus w \in I$. Так как $|B_i \setminus w| < |D|$, то существует $u \in D \setminus (B_i \setminus w)$ такой, что $(B_i \setminus w) \cup u \in I$. И теперь $B_{i+1} \leftarrow (B_i \setminus w) \cup u$. Мы сократили количество несовпадающих элементов с D на 1, симметрическая разность B_i и B_{i+1} состоит из 2 элементов — w и u. □

Наконец, мы подошли к основной теореме лекции — жадный алгоритм или теорема Радо-Эдмондса.

Жадный алгоритм на матроиде

Доказательство будет в несколько этапов.

Для начала определимся с обозначениями. $M = \langle A, I\rangle, n = |A|, w_i$ — элементы множества A. Решаем обычную задачу на максимизацию необходимого множества.

Теорема 1 (Жадный алгоритм. Теорема Радо-Эдмондса)**.** *Если отсортировать все элементы A по невозрастанию стоимостей весовой функции: $c_1 \geqslant c_2 \geqslant \ldots \geqslant c_n$, то такой алгоритм решает исходную задачу о нахождении самого дорогого подмножества:*

Algorithm 1 Жадный алгоритм на матроиде.

```
B ← ∅
for c_i do
    if B ∪ w_i ∈ I then
        B ← B ∪ w_i
```

Доказательство. Пусть B_i — множество, которое мы получим после i шагов цикла нашего алгоритма. Теперь поймём, что наш алгоритм в итоге получит какой-то элемент из базы. Действительно, предположим, что это не так. Тогда существует множество из базы, которое его накрывает: $\exists\ D \in \mathfrak{B} : B_n \subset D$ и $|B_n| < |D|$, так как можно взять любой элемент из базы и добавлять в B_n по 1 элементу из пункта 3 определения матроида.

Получаем, что у нас существует элемент $w_i \in D \setminus B_n$, который мы не взяли нашим алгоритмом, но $B_{i-1} \cup w_i \in I$, так как $B_{i-1} \cup w_i \subset B_n \cup w_i \subset D$, то есть это лежит в I по пункту 2 определения матроида. Значит мы должны были взять w_i, противоречие.

Рассмотрим последовательность d_i из нулей и единиц длины n такую, что $d_i = 1$ только в том случае, если мы взяли алгоритмом i-ый элемент. А оптимальное решение задачи пусть будет e_i — тоже последовательность из 0 и 1. Последовательности будут обозначаться d_i и e_i соответственно.

Если на каком-то префиксе последовательности d_i единиц стало меньше, чем в e_i, то возьмём все элементы, которые помечены в последовательности e_i единицами. Пусть это множество будет E. Аналогично на этом префиксе последовательности d_i определим множество D. $|D| < |E|, D \in I, E \in I$, поэтому мы можем дополнить D каким-то элементом из E, которого не было в D. То есть на этом префиксе у d_i стоит 0 (пусть это будет место i), но заметим, что на i-ом шаге мы обязаны были брать этот элемент, из-за рассуждений аналогичным рассуждению про базу (2 абзаца выше).

Получаем, что на каждом префиксе d_i единиц не меньше, чем на этом же префиксе последовательности e_i. Значит первая единица в d_i встретится не позже, чем в e_i, вторая единица в d_i не позже, чем вторая в e_i и т.д. по рассуждениям по индукции.

□

На лекции была теория про ранги. В доказательстве можно обойтись без неё, просто приложу то, что сказал Глеб. Может быть понадобится в задачах.

Определение 2. ***Рангом*** *множества $B \subseteq A$ (обозн. $r(B)$) называют максимальное число k такое, что $\exists\, C \subseteq B$ такое, что $|C| = k, C \in I$.*

Эта функция обладает таким свойством: для любого элемента $w \in A$ следует, что $r(B \cup w) \leqslant r(B) + r(w)$. Давайте поймём, почему так:

Если $r(B \cup w) = r(B)$, то всё хорошо, так как $r(w) \geqslant 0$. Если $r(B \cup w) = r(B) + 1$ (других вариантов не бывает из определения), то тогда $w \in I$, так как $C \subset (B \cup w)$, что $|C| =$

$r(B \cup w), w \in C$ (иначе C годилось бы для B и $r(B \cup w) = r(B)$), значит $r(w) = 1$, так как $C \in I$, а $w \in C$.

Лекция 2 от 06.09.2016. Быстрое преобразование Фурье

Чтобы быть успешным программистом, надо знать 3 вещи:

- Сортировки;
- Хэширование;
- Преобразование Фурье.

Глеб

В этой лекции будет разобрано дискретное преобразование Фурье (Discrete Fourier Transform).

Применение преобразования Фурье

Допустим, что мы хотим решить такую задачу:

Пример 1. *Даны 2 бинарные строки A и B длины n и m соответственно. Мы хотим найти, какая подстрока в A наиболее похожа на B. Наивная реализация решает эту задачу в худшем случае за $\mathcal{O}(n^2)$. Преобразование Фурье поможет решить эту задачу за $\mathcal{O}(n \log n)$, а именно научимся решать другую задачу:*

Цель. *Хотим научиться перемножать многочлены одной степени $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$ и $B(x) = b_0 + b_1 x + \ldots + b_{n-1}x^{n-1}$ так, что $C(x) = A(x)B(x)$, то есть считать свёртку (найти все коэффициенты, если по-другому) $\sum\limits_{i=0}^{2n-2} \sum\limits_{\substack{j+k=i,\\ j,k\in[n-1]}} a_j b_k x^i$ за $\mathcal{O}(n \log n)$.*

Вернёмся к нашему примеру. Поймём как с помощью нашей **цели** *решать задачу про бинарные строки.*

Пусть $A = a_0 \ldots a_{n-1}, B = b_0 \ldots b_{n-1}$. Их можно считать одной длины (просто добавим нулей в конец b при надобности). Теперь задача переформулировывается как нахождение максимального скалярного произведение B и некоторых циклических сдвигов A (до $n-m+1$).

Инвертируем массив B и припишем в конец n нулей, а к массиву A припишем самого себя. Посмотрим на все коэффициенты перемножения:

$$c_k = \sum_{i+j=k} a_i b_j$$

Но $b_i = 0$ при $i \geqslant n$, поэтому при $k \geqslant n$:

$$c_k = \sum_{i=0}^{n-1} b_i a_{k-i}$$

Выбрав нужные коэффициенты, мы решили эту задачу.

Алгоритм быстрого преобразования Фурье

Основная идея алгоритма заключается в том, чтобы представить каждый многочлен через набор n точек и значений многочлена в этих точках, быстро (за $\mathcal{O}(n \log n)$) вычислить значения в каких-то n точках для обоих многочленов, потом перемножить за $\mathcal{O}(n)$ сами значения. Потом применить обратное преобразование Фурье и получить коэффициенты $C(x) = A(x)B(x)$.

Итак, для начала будем считать, что $n = 2^k$ (просто добавим нулей до степени двойки).

Рассмотрим циклическую группу корней из 1 — $W_n = \{e^{i\frac{2\pi k}{n}} \mid k \in [n-1]\}$. Обозначим за $w_n = e^{i\frac{2\pi}{n}}$. Одно из самых главных свойств, что $w_n^p \cdot w_n^q = w_n^{p+q}$, которым мы будем пользоваться в дальнейшем.

Воспользуемся идеей метода «разделяй и властвуй».

Пусть $A(x) = a_0 + \ldots + a_{n-1}x^{n-1}$.

Представим $A(x) = A_l(x^2) + xA_r(x^2)$ так, что

$$A_l(x^2) = a_0 + a_2x^2 + \ldots + a_{n-2}x^{n-2}$$
$$A_r(x^2) = a_1 + a_3x^2 + \ldots + a_{n-1}x^{n-2}$$

Определение 1. *Назовём* ***Фурье-образом*** *многочлена* $P(x) = p_0 + \ldots + p_{m-1}x^{m-1}$ *вектор из* m *элементов* — $\langle P(1), P(w_m), P(w_m^2), \ldots, P(w_m^{m-1})\rangle$.

Теперь рекурсивно запускаемся от многочленов меньшей степени. Так как для любого целого неотрицательного k следует, что $2k$ четное число, то $w_n^{2k} = w_{n/2}^k \in W_{n/2}$, то есть мы можем уже использовать значения Фурье-образа для вычисления $A(x)$.

Если мы сможем за линейное время вычислить сумму $A_l(x^2) + xA_r(x^2)$, то суммарное время работы будет $\mathcal{O}(n \log n)$, так как $A_l(x), A_r(x)$ имеют степень в 2 раза меньше, чем $A(x)$.

Действительно это легко следует из псевдокода, который приведен ниже:

Algorithm 2 FFT

function FFT(A) ▷ A — массив из комплексных чисел, функция возвращает Фурье-образ
  $n \leftarrow \text{length}(A)$
  if $n = 1$ **then**
    return A
  $A_l \leftarrow \langle a_0, a_2, \ldots, a_{n-2}\rangle$
  $A_r \leftarrow \langle a_1, a_3, \ldots, a_{n-1}\rangle$
  $\hat{A}_l \leftarrow \text{FFT}(A_l)$
  $\hat{A}_r \leftarrow \text{FFT}(A_r)$
  for $k \leftarrow 0$ to $\frac{n}{2} - 1$ **do**
    $\hat{A}[k] \leftarrow \hat{A}_l[k] + e^{i\frac{2\pi k}{n}}\hat{A}_r[k]$
    $\hat{A}[k + \frac{n}{2}] \leftarrow \hat{A}_l[k] - e^{i\frac{2\pi k}{n}}\hat{A}_r[k]$ ▷ Здесь минус перед комплексным числом из-за того, что мы должны найти другой угол, удвоенный которого на окружности будет $\frac{2\pi(k+n/2)}{n}$
  return $\hat{A}$

Теперь поговорим про обратное FFT.

Фактически, мы вычислили такую вещь за $\mathcal{O}(n \log n)$:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

где y_i — Фурье-образ многочлена $A(x)$.

Фактически нам надо найти обратное преобразование. Магическим образом обратная матрица к квадратной матрице выглядит почти также:

$$\frac{1}{n}\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}$$

$$\frac{1}{n}\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Откуда получаем: $a_k = \frac{1}{n}\sum\limits_{j=0}^{n-1} y_j w_n^{-kj}$.

Теперь напишем псевдокод обратного алгоритма:

Algorithm 3 FFT_inverted

function FFT_INVERTED(A) ▷ A — Фурье-образ, возвращает коэффициенты многочлена
    $n \leftarrow \text{length}(A)$
    if $n = 1$ **then**
        return A
    $A_l \leftarrow \langle a_0, a_2, \ldots, a_{n-2}\rangle$
    $A_r \leftarrow \langle a_1, a_3, \ldots, a_{n-1}\rangle$
    $\hat{A}_l \leftarrow$ FFT_inverted(A_l)
    $\hat{A}_r \leftarrow$ FFT_inverted(A_r)
    for $k \leftarrow 0$ to $\frac{n}{2} - 1$ **do**
        $\hat{A}[k] \leftarrow \hat{A}_l[k] + \mathrm{e}^{i\frac{-2\pi k}{n}}\hat{A}_r[k]$ ▷ Здесь угол идёт с минусом
        $\hat{A}[k + \frac{n}{2}] \leftarrow \hat{A}_l[k] - \mathrm{e}^{i\frac{-2\pi k}{n}}\hat{A}_r[k]$
        $\hat{A}[k] \leftarrow \hat{A}[k]/2$ ▷ Поделим на 2 $\log n$ раз, а значит поделим на n в итоге
        $\hat{A}[k + \frac{n}{2}] \leftarrow \hat{A}[k + \frac{n}{2}]/2$ ▷ Аналогично строчке выше
    return $\hat{A}$

Лекция 3 от 16.09.2016. Алгоритм Карацубы, алгоритм Штрассена

«Алгоритм Карацубы — это верх математики. Просто посидел, придумал, вот и кандидатская готова.»

Глеб

Перемножение двух длинных чисел с помощью FFT

Пусть $x = \overline{x_1 x_2 \dots x_n}$ и $y = \overline{y_1 y_2 \dots y_n}$. Распишем их умножение в столбик:

$$\begin{array}{rr} & \times\, x_1\, x_2 \dots x_n \\ & \underline{y_1\, y_2 \dots y_n} \\ & z_{11}\, z_{12} \dots z_{1n} \\ & z_{21}\, z_{22} \dots z_{2n} \\ + & \dots\dots\dots\dots \\ & \underline{z_{n1}\, z_{n2} \dots z_{nn}} \\ & z'_1\, z'_2 \dots\dots\dots\dots\dots\dots z'_{2n} \end{array}$$

Понятно, что наивное умножение двух длинных чисел будет иметь сложность $\mathcal{O}(n^2)$.

Давайте научимся перемножать два числа быстрым преобразованием Фурье за $\mathcal{O}(n \log n)$.

Пусть $a = \overline{a_{n-1} \dots a_0}, b = \overline{b_{n-1} \dots b_0}$.

Тогда введём многочлены $f(x) = \sum\limits_{i=0}^{n-1} a_i x^i, g(x) = \sum\limits_{i=0}^{n-1} b_i x^i$.

За $\mathcal{O}(n \log n)$ мы можем найти $h(x) = f(x) \cdot g(x) = \sum\limits_{i=0}^{2n-2} c_i x^i$.

После этого надо аккуратно провести переносы разрядов таким образом и после этого развернуть полученное число, отбросив ненужные нули в начале:

Algorithm 4 Умножение 2 длинных чисел.

```
1: function Умножение 2 длинных чисел(h(x))  ▷ h(x) — перемножение 2 многочленов f(x) и g(x).
2:     carry ← 0
3:     for i ← 0 to 2n − 1 do
4:         h_i ← h_i + carry
5:         carry ← ⌊h_i / 10⌋
6:         h_i ← h_i mod 10
```

Но этот метод плохо применим на практике из-за того, что быстрое преобразование Фурье имеет очень большую константу в асимптотике времени работы.

Алгоритм Карацубы

Какое-то время человечество не знало алгоритмов перемножения быстрее, чем за $\mathcal{O}(n^2)$. А.Н. Колмогоров считал, что это вообще невозможно. В один момент собрались математики на мехмате МГУ и решили доказать, что это невозможно. Но один из аспирантов (Анатолий Алексеевич Карацуба) Колмогорова пришёл и сказал, что у него получилось сделать это быстрее. Давайте посмотрим, как:

Будем считать, что $n = 2^k$ (если это не так, дополним нулями, сложность вырастет лишь в константу раз).

Для начала просто попробуем воспользоваться стратегией «Разделяй и властвуй». Разобьём числа в разрядной записи пополам. Тогда

$$\times \begin{cases} x = 10^{n/2}a + b \\ y = 10^{n/2}c + d \end{cases}$$
$$\Downarrow$$
$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd$$

Как видно, получается 4 умножения чисел размера $\frac{n}{2}$ или $\frac{n}{2} + 1$ (оставляем читателю на подумать, как справиться с последней ситуацией быстро). Так как сложение имеет сложность $\Theta(n)$, то

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$$

Чему равно $T(n)$? Если посмотреть на дерево исходов или воспользоваться индукцией, то получим, что $T(n) = \mathcal{O}(n^2)$, что, конечно, неэффективно.

Анатолий Алексеевич проявил недюжие способности и предложил следующее:

$$(a + b)(c + d) = ac + (ad + bc) + bd \implies ad + bc = (a + b)(c + d) - ac - bd$$

Подставим это в начальное выражение для xy:

$$xy = 10^n ac + 10^{n/2}((a + b)(c + d) - ac - bd) + bd$$

Отсюда видно, что на предыдущем уровне надо вычислить 3 умножения чисел размера $\frac{n}{2}$ или $\frac{n}{2} + 1$ (оставляем читателю на подумать, как справиться с последней ситуацией быстро): $(a + b)(c + d), ac$ и bd. Тогда:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$$

Докажем, что $T(n) = \mathcal{O}(n^{\log_2 3})$.

Рассмотрим дерево исходов: в каждой вершине дерева мы выполняем не более Cm действий, где C —какая-то фиксированная константа, а m — размер числа на данном шаге, поэтому

$$T(n) \leqslant Cn\left(1 + \frac{3}{2} + \ldots + \frac{3^{\log_2 n}}{2^{\log_2 n}}\right),$$

так как на каждом шаге мы запускаемся три раза от задачи в два раза меньше.

Откуда $T(n) \leqslant Cn \cdot \frac{\frac{3}{2}^{\log_2 n} - 1}{1/2} = 2Cn^{\log_2 3} = \mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.5849})$

Полученный алгоритм называется алгоритмом Карацубы.

Перемножение матриц. Алгоритм Штрассена

После идеи А.А. Карацубы, появились многие алгоритмы, использующие ту же идею. Одним из этих алгоритмов является алгоритм Штрассена. Будем считать, что $n = 2^k$ снова (оставляем читалелю самим подумать, как дополнить матрицы $m \times t, t \times u$, чтобы потом лекго восстановить ответ)

Пусть у нас есть квадратные матрицы

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \text{ и } B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}$$

Сколько операций нужно для умножения матриц? Умножим их по определению. Матрицу $C = AB$ заполним следующим образом:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Всего в матрице n^2 элементов. На получение каждого элемента уходит $\mathcal{O}(n)$ операций (умножение за константное время и сложение n элементов). Тогда умножение требует $n^2\mathcal{O}(n) = \mathcal{O}(n^3)$ операций.

Попробуем применить аналогичную стратегию «Разделяй и властвуй». Представим матрицы A и B в виде:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ и } B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

где каждая матрица имеет размер $\frac{n}{2}$. Тогда матрица C будет иметь вид:

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Как видно, получаем 8 перемножений матриц порядка $\frac{n}{2}$. Тогда

$$T(n) = 8T\left(\frac{n}{2}\right) + \mathcal{O}(n^2)$$

По индукции получаем, что $T(n) = \mathcal{O}\left(n^{\log_2 8}\right) = \mathcal{O}(n^3)$.

Можно ли уменьшить число умножений до 7? Алгоритм Штрассена утверждает, что можно. Он предлагает ввести следующие матрицы (даже не спрашивайте, как до них дошли):

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}); \\ M_2 = (A_{21} + A_{22})B_{11}; \\ M_3 = A_{11}(B_{12} - B_{22}); \\ M_4 = A_{22}(B_{21} + B_{11}); \\ M_5 = (A_{11} + A_{12})B_{22}; \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}); \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}); \end{cases}$$

Тогда

$$\begin{cases} C_1 &= M_1 + M_4 - M_5 + M_7; \\ C_2 &= M_3 + M_5; \\ C_3 &= M_2 + M_4; \\ C_4 &= M_1 - M_2 + M_3 + M_6; \end{cases}$$

Можно проверить что всё верно (оставим это как ~~наказание~~ упражнение читателю). Сложность алгоритма:

$$T(n) = 7T\left(\frac{n}{2}\right) + \mathcal{O}(n^2) \implies T(n) = \mathcal{O}\left(n^{\log_2 7}\right) \approx \mathcal{O}(n^{2.8073})$$

Доказательство времени работы такое же, как и в алгоритме Карацубы.

Также существует модификация алгоритма Штрассена, где используется лишь пятнадцать сложений матриц на каждом шаге, вместо восемнадцати предъявленных выше.

Эквивалентность асимптотик некоторых алгоритмов

Этот раздел не войдёт в экзамен.

Здесь мы поговорим об обращении и перемножении 2 матриц. Докажем, что асимптотики этих алгоритмов эквивалентны.

Теорема 1 (Умножение не сложнее обращения)**.** *Если можно обратить матрицу размеров $n \times n$ за время $T(n)$, где $T(n) = \Omega(n^2)$, и $T(3n) = \mathcal{O}(T(n))$ (условие регулярности), то две матрицы размером $n \times n$ можно перемножить за время $\mathcal{O}(T(n))$*

Доказательство. Пусть A и B матрицы одного порядка размера $n \times n$. Пусть

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

Тогда легко понять, что

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

Матрицу D мы можем построить за $\Theta(n^2)$, которое является $\mathcal{O}(T(n))$, поэтому с условием регулярности получаем, что $M(n) = \mathcal{O}(T(n))$, где $M(n)$ — асимптотика перемножения 2 матриц. □

С обратной теоремой предлагаем ознакомиться в книге Кормена, Лейзерсона, Ривеста и Штайна или в книге Ахо, Хопкрофта и Ульмана.

Лекция 4 от 20.09.2016. Простейшие теоретико-числовые алгоритмы

Числовые алгоритмы играют огромную роль в криптографии, фактически вся криптография держится на том, что не придуман до сих пор алгоритм, который умеет факторизовать числа за полиномиальное время от размера числа.

Алгоритм Евклида

Начнём, пожалуй, с одного из самых известных алгоритмов нахождения наибольшего общего делителя, а именно — алгоритм Евклида и его расширенную версию.

Algorithm 5 Алгоритм Евклида.

```
1: function gcd(int a, int b)
2:     if b = 0 then
3:         return a;
4:     else
5:         return gcd(b, a mod b);
```

Вспомним курс дискретной математики, и сошлёмся на то, что все умеют доказывать корректность этого алгоритма.

Асимптотика такого алгоритма $\mathcal{O}(\log n)$ (где $n = \max(a, b)$) — легко проверить, что каждое число уменьшается хотя бы в два раза за два шага алгоритма.

Расширенный алгоритм Евклида

Пусть даны числа a, b, c, мы хотим найти хотя бы одну пару решений x, y таких, что $ax+by = c$. Понятно, что $\gcd(a, b) \mid c$, поэтому если это условие не выполняется, то найти решение мы не сможем. Пусть $c = k \gcd(a, b)$. Сейчас мы предъявим хотя бы одну пару чисел x, y, что $ax + by = \gcd(a, b)$ — после этого мы просто домножим x, y на k и получим, что сможем представить c в таком виде.

Algorithm 6 Расширенный алгоритм Евклида.

```
1: function EXTENDED_gcd(int a, int b)        ▷ — возвращаем тройку чисел (x, y, gcd(a, b)).
2:     if b = 0 then
3:         return (1, x', a);                 ▷ x' можно вернуть любым числом, так как b = 0
4:     (x', y', d) ← EXTENDED_gcd(b, a mod b)
5:     return (y', x' − ⌊a/b⌋ y', d)
```

Лемма 1. *Для произвольных неотрицательных чисел a и b ($a \geqslant b$) расширенный алгоритм Евклида возвращает целые числа x, y, d, для которых* $\gcd(a, b) = d = ax + by$.

Доказательство. Если не рассматривать x, y в алгоритме, то такой алгоритм полностью повторяет обычный алгоритм Евклида. Поэтому алгоритм третьим параметром действительно вычислит $\gcd(a, b)$.

Для обоснования корректности возвращаемых значений x, y будем вести индукцию по b. Если $b = 0$, тогда мы действительно вернём верное значение. Шаг индукции: заметим, что алгоритм находит $\gcd(a, b)$, произведя рекурсивный вызов для $(b, a \bmod b)$. Поскольку $(a \bmod b) < b$, мы можем воспользоваться предположением индукции и заключить, что для возвращаемых рекурсивным вызовом чисел x', y' выполняется равенство:

$$\gcd(b, a \bmod b) = bx' + (a \bmod b)y'$$

Понятно, что $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$, поэтому

$$d = \gcd(a, b) = \gcd(b, a \bmod b) = bx' + (a \bmod b)y' = \\ = bx' + \left(a - \left\lfloor \frac{a}{b} \right\rfloor b\right) y' = ay' + b\left(x' - \left\lfloor \frac{a}{b} \right\rfloor y'\right)$$

□

Пример 1. *Мы умеем с помощью расширенного алгоритма Евклида вычислять обратные остатки по простому модулю (в поле $\mathbb{F}_p$). Действительно, если $(a, p) = 1$ то существуют x, y, что $ax + py = 1$, а значит в поле $\mathbb{F}_p$ — $ax = 1$, откуда $x = a^{-1}$.*

Алгоритм быстрого возведения в степень по модулю

Хотим вычислить $a^b \bmod p$. Основная идея в том, чтобы разложить b в двоичную систему и вычислять только $a^{2^i} \bmod p$. Здесь будем предполагать, что операции с числами выполняются достаточно быстро. Приведём ниже псевдокод такого алгоритма:

Algorithm 7 Алгоритм быстрого возведения в степень.

```
1: function FAST_POW(int a, int b, int p)        ▷ — возвращаем a^b mod p.
2:     if b = 0 then
3:         return 1;
4:     if b mod 2 = 1 then
5:         return FAST_POW(a, b − 1, p) · a mod p
6:     else
7:         c ← FAST_POW(a, b/2, p)
8:         return c^2 mod p
```

Корректность этого алгоритма следует из того, что $a^b = a^{b-1} \cdot a$ для нечетных b и

$a^b = a^{b/2} \cdot a^{b/2}$ для четных b. Также мы здесь неявно пользуемся индукцией по b, в которой корректно возвращается база при $b = 0$.

От каждого числа b, если оно четно, мы запускаем наш алгоритм от $b/2$, а если оно нечетно, то от $b-1$, откуда получаем, что количество действий, совершенным нашим алгоритмом будет не более, чем $2 \log b = \mathcal{O}(\log b)$.

Замечание 1. *На самом деле быстрое возведение в степень работает на всех ассоциативных операциях. Например, если вы хотите вычислить A^n, где A — квадратная матрица $(m \times m)$, то это можно сделать тем же самым алгоритмом за $\mathcal{O}(T(m) \log n)$, где $T(m)$ асимптотика перемножения матриц $m \times m$.*

Китайская теорема об остатках и вычисление решений линейных систем

Китайская теорема об остатках звучит так: пусть даны попарно взаимно простые модули и числа $r_1, \ldots, r_n$. Тогда существует единственное с точностью по модулю $a_1 \ldots a_n$ решение такой системы:

$$\begin{cases} x \equiv r_1 \pmod{a_1} \\ x \equiv r_2 \pmod{a_2} \\ \quad \vdots \\ x \equiv r_n \pmod{a_n} \end{cases}$$

Доказательство. Докажем теорему и предъявим сразу алгоритм вычисления решения за $\mathcal{O}(n \log(a_1 \cdot \ldots \cdot a_n))$.

Пусть $x = \sum\limits_{i=1}^{n} r_i M_i M_i^{-1}$, где $M_i = \dfrac{a_1 \cdot \ldots \cdot a_n}{a_i}, M_i^{-1}$ это обратное к M_i по модулю a_i (такое всегда найдётся из попарной взаимной простоты). Прошу заметить, что такое число мы можем вычислить за $\mathcal{O}(n \log(a_1 \cdot \ldots \cdot a_n))$(см. пример в расширенном алгоритме Евклида).

Докажем, что это число подходит по любому модулю a_i.

$$x \equiv \sum_{j=1}^{n} r_j M_j M_j^{-1} \equiv r_i M_i M_i^{-1} \equiv r_i \pmod{a_i}$$

Второе равенство следует из того, что $a_i \mid M_j$ при $j \neq i$ (из построения).

Докажем единственность решения по модулю. Пусть x, x' — различные решения данной системы, тогда $0 < |x - x'| < a_1 \cdot \ldots \cdot a_n$ и $|x - x'|$ делится на $a_1 \cdot \ldots \cdot a_n$, что невозможно, так как ни одно положительное число до $a_1 \cdot \ldots \cdot a_n$ не делится на $a_1 \cdot \ldots \cdot a_n$. □

Решето Эратосфена

Решето Эратосфена — это один из первых алгоритмов в истории человечества. Он позволяет найти все простые числа на отрезке от $[1; n]$ за $\mathcal{O}(n \log \log n)$, а разложить все числа на простые множители за $\mathcal{O}(n \log n)$

В первом случае у нас задача состоит в том, чтобы вернуть 1, если число простое и 0, если непростое.

Предъявим ниже псевдокод такого алгоритма.

Лемма 2. *Алгоритм* Sieve_of_Eratosthenes *корректно оставит все простые числа.*

Доказательство. Докажем индукцией по n. База $n = 2$ очевидна.

Переход $n \to n + 1$. Заметим, что наш алгоритм и корректно обрабатывает первые n чисел, потому что мы только расширяем область рассматриваемых чисел.

Если $n + 1$ составное, тогда $n + 1 = p \cdot m$ для какого-то простого $p < n + 1$. По предположению индукции, мы рассмотрим простое число p, то есть удалим из массива все числа, которые кратны p, а значит и $n + 1$ мы пометим как составное.

Если $n + 1$ простое, то мы не могли пометить его как составное, так как оно не делилось ни на одно простое, рассмотренное ранее. □

Algorithm 8 Решето Эратосфена.

```
1: function SIEVE_OF_ERATOSTHENES(int n)        ▷ найти — массив prime_i, определяющий
   характеристическую функцию простых чисел от 1 до n.
2:     for i ← 1 to n do
3:         prime_i ← true
4:     prime_1 ← false
5:     for i ← 2 to n do
6:         if prime_i = true then
7:             j ← 2i
8:             while j ⩽ n do
9:                 prime_j ← false
10:                j ← j + i
```

Заметим, что алгоритм будет выполняться за время

$$\sum_{\substack{p\leqslant n,\\ p\in\mathbb{P}}} \frac{n}{p},$$

так как для каждого простого числа мы рассматриваем в таблице все числа, кратные p.

Можно оценить очень грубо и получим, что

$$\sum_{\substack{p\leqslant n,\\ p\in\mathbb{P}}} \frac{n}{p} \leqslant \sum_{i=1}^{n} \frac{n}{i} \approx n\ln n + o(n) = \mathcal{O}(n\log n)$$

Но используя свойства ряда $\sum\limits_{\substack{p\leqslant n,\\ p\in\mathbb{P}}} \frac{n}{p} \approx n\ln\ln n + o(n)$, получим, что алгоритм работает за $\mathcal{O}(n\log\log n)$, но факт про асимптотику этого ряда мы оставим без доказательства.

Если теперь первый раз, приходя в составное число в алгоритме, будем сохранять его наименьший простой делитель, то рекурсивно мы можем разложить число на простые множители. Общее количество простых делителей у числа не может превышать $\mathcal{O}(\log n)$ (так как наименьший простой делитель это двойка), поэтому разложение на множители будет выполняться за $\mathcal{O}(n\log n)$.

Решето Эйлера

Составим двусвязный список из чисел от 2 до n, а также ещё массив (длины n) указателей на элементы списка.

Будем идти итеративно: первый непросмотренный номер в списке берётся как простое число, и определяются все произведения с последующими элементами в списке (само на себя тоже умножим). Это продолжается до тех пор, пока мы не выйдем в произведении за пределы n. После этого удаляются все числа, которые мы вычислили (смотрим в массив укзателей и удаляем по указателю за $\mathcal{O}(1)$). Далее заново повторяем процедуру.

Лемма 3. *После k шагов алгоритма останутся первые k простых чисел в начале, и в списке будут только числа взаимно простые с первыми k.*

Доказательство. База при $k = 1$ очевидна. Просто убираем все четные числа.

Переход $k \to k + 1$.

Докажем, что следующим нерассмотренным элементом списка мы возьмём p_{k+1}. Действительно, простые числа мы не выкидываем, а значит следующим шагом после p_k мы возьмём число, не большее p_{k+1}, но по предположению индукции все числа от (p_k, p_{k+1}) были убраны, так как они составные и содержат в разложении только простые меньшие p_{k+1}.

Предположим, что после ещё одного шага алгоритма у нас осталось число, кратное p_{k+1} (и большее p_{k+1}) (все числа, делящиеся на предыдущие простые до этого были убраны).

Тогда пусть это будет $m = p_{k+1} \cdot a, a > 1$. Если a содержит в разложении на простые хотя бы одно число, меньшее p_{k+1}, то получим противоречие, так как все числа не взаимно простые с $p_1, \ldots, p_k$ по предположению индукции были убраны.

Значит a содержит в разложении на простые числа, не меньшие p_{k+1}, а значит $a \geqslant p_{k+1}$ и это число ещё было в списке, значит мы это число уберем, противоречие. □

Если мы вдруг на шаге алгоритма получили в умножении число, которое мы уже убрали, то значит у этого числа есть меньший простой делитель, чем p_k, но по доказанной лемме у нас все такие числа к k-ому шагу были убраны. Значит каждое составное число мы рассмотрим ровно один раз и ровно один раз уберем за $\mathcal{O}(1)$.

По лемме получаем, что в списке останутся только простые числа.

Простые числа мы тоже рассматриваем по одному разу в нашем алгоритме, значит общая сложность решета Эйлера будет $\mathcal{O}(n)$.

Наивная факторизация числа за $\mathcal{O}(\sqrt{n}\,)$

На данный момент не существует алгоритма факторизации числа за полином от размера числа, а не от значения. Здесь мы рассмотрим наивный алгоритм факторизации числа. На следующей лекции рассмотрим ρ-метод Полларда (**UPD** так и не рассмотрели), который работает за $\mathcal{O}(\sqrt[4]{n}\,)$.

Пусть $n' = n$. Будем перебирать от 2 до $\lceil \sqrt{n'} \rceil$ числа и пока текущее n делится на данное число, делим n на это число.

Легко показать, что делим мы только на простые числа (иначе мы поделили бы на меньшее простое несколькими шагами раньше).

В конце n будет либо 1 (тогда факторизация удалась), либо простым. Составным оно не может быть, иначе $n = ab, a, b > 1$ и $a, b > \lceil \sqrt{n'} \rceil$, так как на все числа, меньшие корня, мы поделили.

Осталось оценить, сколько операций раз мы обращаемся к циклу *while*. В нём мы делаем суммарно не более, чем $\mathcal{O}(\log n + \sqrt{n}\,)$ действий, так как сумма степеней в разложении числа не более, чем $\mathcal{O}(\log n)$ (см. выше). Ну а также обращаемся по одному разу каждый шаг внешнего цикла.

Лекция 5 от 27.09.2016. RSA, продолжение некоторых теоретико-числовых алгоритмов, некоторые комбинаторные оптимизации

> «На самом деле, если этот алгоритм (разложение на простые) придумают за полиномиальное время, можно спокойно идти и покупать попкорн и смотреть, как рушится этот мир.
> Только не забудьте перед этим снять все деньги с банковских карточек.»
>
> Глеб

Прим. Я в этой лекции поменял местами темы, чтобы было легче воспринимать материал.

Предисловие

Теория чисел с появлением алгоритмов, а особенно криптографии, приобрела новую «жизнь». Теперь простые числа, разложение на простые множители являются важными алгоритмами даже в нашей повседневной жизни. Сначала поговорим о самих простых числах, потом о криптографии.

Тест Рабина-Миллера на проверку простоты числа

В той же самой криптографии есть необходимость в генерировании длинных простых чисел. Благо простые числа встречаются не так редко. Пусть функция распределения простых чисел будет $\pi(n)$ — количество чисел, не бóльших n, которые являются простыми. Есть теорема, о которой мы уже упоминали

$$\lim_{n\to+\infty} \frac{\pi(n)}{n/\ln(n)} = 1$$

То есть если взять случайно $\ln(n)$ чисел до n, то математическое ожидание количества простых чисел равно единице.

Поэтому, чтобы сгенерировать большое простое число, надо уметь проверять за полином от числа (было бы вообще прекрасно), является ли число простым. Был найден алгоритм, который проверяет это свойство за полином от размера числа.

Но мы рассмотрим достаточно эффективный вероятностный алгоритм проверки числа на простоту. Вспомним малую теорему Ферма, которая нам гласит:

$$a^{p-1} \equiv 1 \bmod p \text{ для всех простых чисел } p.$$

Определение 1. *Назовём число n псевдопростым по основанию a, если $(a, n) = 1$ и $a^{n-1} \equiv 1 \pmod{n}$.*

Поэтому первый наш алгоритм будет такой (будем проверять нечётные n и $a = 2$):

Algorithm 9 Проверка на псевдопростоту по основанию 2.

```
1: function PSEUDOPRIME(int n)
2:     if FAST_POW(2, n − 1, n) ≠ 1 then
3:         return Составное                          ▷ 100% составное
4:     return Простое          ▷ молимся и надеемся, что тут действительно простое
```

Но, к сожалению, существуют и составные числа, которые удовлетворяют этому алгоритму и нам выведется, что число простое, а, на самом деле, нет. Наименьшие из них это $341, 561 \ldots$ и так далее. Поменять основание тоже не вариант, так как существуют числа Кармайкла, которые псевдопростые по любому основанию. Но было доказано, что с ростом n вероятность, что непростое число является псевдопростым стремится к 0.

К ~~счастью~~ сожалению, мы не будем доказывать вероятность ошибки этого алгоритма (она составляет не более 0.5), поэтому если запустить этот алгоритм k раз, то вероятность ошибки будет равна 2^{-k}. Собственно, алгоритм:

Algorithm 10 Тест Миллера-Рабина, считаем, что $n - 1 = 2^t u$, где u нечетно и $t \geqslant 1$

```
1: function MILLER_RABIN(int n)
2:     a ← random_integer(1, n − 1)
3:     x_0 ← FAST_POW(a, u, n)
4:     for i ← 1 to t do
5:         x_i ← x_{i−1}^2 mod n
6:         if x_i = 1 и x_{i−1} ≠ 1 и x_{i−1} ≠ n − 1 then
7:             return Составное
8:     if x_t ≠ 1 then
9:         return Составное
10:    return Простое
```

Ясно, что этот алгоритм работает за полином от числа (при условии, что мы умеем вычислять по модулю быстро, но это задача в листке, да и это легко придумать без метода Ньютона, оставляю читалю это сделать).

Лемма 1. *Если алгоритм как-то вышел на строчки семь или девять, то число составное.*

Доказательство. Заметим, что $x_i \equiv a^{2^i u} \bmod n$, так как $x_0 \equiv a^u \bmod n$ (база индукции) и $x_i = x_{i-1}^2 \bmod n$ (переход индукции), а значит $x_i \equiv a^{2^{i-1}u} \cdot a^{2^{i-1}u} \equiv a^{2^i u} \bmod n$.

Поэтому если $x_i = 1$ и $x_{i-1} \neq 1$ и $x_{i-1} \neq n - 1$, то $n \mid x_{i-1}^2 - 1$, то есть $(x_{i-1} - 1)(x_{i-1} + 1) = 0$ в $\mathbb{Z}_n$. То есть у нас нетривиальные делители нуля, а из курса алгебры известно, что в поле (а при простых n — $\mathbb{Z}_n$ поле) их нет, поэтому перед нами составное число.

Если $x_t \neq 1$, то просто-напросто не выполняется малая теорема Ферма и тогда n точно составное. □

Лемма 2. *Количество таких a, на которых алгоритм выдаст «Составное» не меньше $\frac{n-1}{2}$ при составном нечетном n.*

Именно этот факт мы оставим без доказательства (*Прим. на самом деле, он не очень сложный, видимо, Глебу было лень*). И именно он нам даёт ошибку не более 0.5.

RSA, криптография

Криптографическую систему с открытым ключом можно использовать для шифровки сообщений, которыми обмениваются два партнера (Алиса и Боб), чтобы посторонние люди (Ева в дальнейшем), даже перехватившие сообщения, не могли его расшифровать. Также некоторая система позволяет подписывать свои подписи. Кто угодно без труда может её проверить, но подделать никак.

Давайте уже перейдём к обсуждению различных систем.

Но для начала несколько определений. У Алисы есть ключи P_A, R_A, у Боба P_B, R_B — публичные и приватные соответственно (на самом деле, это функции, которые что-то вычисляют). Алиса и Боб хранят приватные ключи у себя, а с открытыми могут делать что угодно. Будем считать, что Алиса и Боб передают двоичные последовательности. Также будем считать, что $M = P_A(R_A(M)) = R_A(P_A(M))$ и $M = P_B(R_B(M)) = R_B(P_B(M))$. Также, чтобы шифрование имело смысл, надо, чтобы секретные ключи владельцы умели вычислять быстро, и чтобы по открытому ключу было очень сложно вычислить обратное преобразование. На этом и держится весь алгоритм. Рассмотрим пример:

Боб хочет отправить сообщение M Алисе, зашифрованное так, чтобы для Евы оно выглядело как ужасный набор символов:

- Боб получает открытый ключ Алисы P_A любым способом.
- Боб шифрует сообщение, которое знает только он, как $C = P_A(M)$.
- Алиса, когда получает сообщение C, расшифровывает это сообщение с помощью секретного ключа.

Функции обратные, поэтому вычисления будут корректными. Но, к сожалению, такая система плоха тем, что, перехватив сообщения, Ева может их подменивать. Поэтому часто используют ещё и цифровые подписи:

Пусть Алиса хочет отправить сообщение M' Бобу:

- Алиса вычисляет свою подпись с помощью своего секретного ключа. $\sigma = R_A(M')$.
- Алиса отправляет пару Бобу (M', σ).
- Боб может легко убедиться, что это действительно Алиса, с помощью открытого ключа, вычислив $P_A(\sigma)$ и сравнив с M'.

В данном случае никакая Ева не страшна в подмене сообщения, так как она не может вычислить $R_A(M')$ ни для какого M'.

Такие подписи позволяют проверять целостность сообщений. Но всё равно есть проблема — Ева знает содержания сообщений. Можно взять ещё один ключ, который шифрует по первой схеме сообщения, которые мы передаём по второй схеме. И тогда Ева, даже получив перехваченное сообщение, во-первых, не сможет понять, какой парой оно было зашифровано, то есть дешифровка невозможна за разумное время, да и подмена тоже, так как к сообщению применяется сложный ключ.

Проблема остаётся одна — что Алиса и Боб должны обменяться ключами, чтобы Ева не могла подменить открытые ключи. Но, к сожалению, невозможно спрятаться от Евы, если быть совсем параноиком. Она всегда может подменять вам ключи, где бы вы ни находились. Поэтому факт личной встречи должен быть. И самая большая «insecurity» состоит именно в том, что Ева внедряется работать к Алисе, чтобы разузнать, а то и подменять ключи для Боба.

Давайте уже, наконец-то поговорим о способах шифрования:

- Самая старая система — это шифр Цезаря. Она просто переставляет по циклу символы в алфавите, что, конечно же, ломается за $\mathcal{O}(k)$, где k — размер алфавита.
- Взять рандомную перестановку алфавита. Да, задача уже сложнее, но если это какой-нибудь язык, то можно из статистических параметров восстанавливать символы, что значительно сократит перебор. Не годится.
- Например, выравнивать двоичные сообщения и брать рандомную перестановку, применяя её ко всем сообщениям. Тогда нетрудно убедиться, что за $\mathcal{O}(\log n)$ действий мы сможем понять, какой символ, где стоит. Просто смотреть, куда переходят единицы и нули. На непонятных рандомных сообщениях математическое ожидание того, чтобы разобрать, где что стоит, будет $\mathcal{O}(\log n)$.

Все примеры сверху так или иначе зависят от человеческого фактора или для них легко подобрать обратную функцию. Рассмотрим криптографическую систему RSA (Rivest–Shamir–Adleman public-key cryptosystem).

1. Случайным образом выбираются два больших простых числа $p \neq q$. Мы уже обсудили выше, что это сделать легко достаточно.
2. Вычисляется $n = pq$ (что можно сделать тоже не очень сложно алгоритмом Карацубы или преобразованием Фурье).
3. Выбирается маленькое нечетное число e, взаимно простое с $\varphi(n) = (p-1)(q-1)$ (разложение $\varphi(n)$ такое из-за мультипликативности).
4. Вычисляем число $d = e^{-1} \bmod \varphi(n)$. Это можно сделать расширенным алгоритмом Евклида.
5. Пара $P = (e, n)$ будет открытым ключом.
6. Пара $S = (d, n)$ закрытым.

Теперь в качестве сообщения мы передаём сообщение $P(M) = M^e \bmod n$.

Обратное шифрование будет равно $S(C) = C^d \bmod n$.

Аналогично, ясно, что это шифрование работает за полином от длины числа, так как все операции мы умеем делать за полином от длины числа.

Лемма 3 (Корректность RSA)**.** *Докажем, что $P(M)$ и $S(C)$ взаимно обратные функции.*

Доказательство. Видно, что $P(S(M)) = S(P(M)) = M^{ed} \bmod n$.

Так как e, d взаимно обратные по модулю $\varphi(n)$, то $ed = 1 + k(p-1)(q-1)$

$$M^{ed} \equiv M^{1+k(p-1)(q-1)} \equiv M \cdot M^{k(q-1)\varphi(p)} \bmod p$$

$$M^{ed} \equiv M^{1+k(p-1)(q-1)} \equiv M \cdot M^{k(p-1)\varphi(q)} \bmod q$$

Малая теорема Ферма имеет очень простое следствие, что для любых чисел $M^p \equiv M \bmod p$ (предлагается это доказать самостоятельно). Поэтому в обоих равенствах в арифметике это просто эквивалентно M, то есть

$$M^{ed} \equiv M \bmod p \text{ и } M^{ed} \equiv M \bmod q$$

А значит по легкому следствию из основной теоремы арифметики $M^{ed} \equiv M \bmod n$. □

Основная сложность в том, что зная $n, e, M^e \bmod n$ практически невозможно найти M. Перебрать все M может занять экспоненциальную сложность, а разложение на множители n и вычисление d оказалось очень сложной задачей, которая пока не решается за полиномиальное время.

Байка от Глебаса: На самом деле вся теория по шифрованию в интернете появилась лет 5-6 назад. Раньше кто угодно мог перехватывать сообщения вашей почты, платить в интернете было опасно (если вообще можно было). Я только однажды покупал что-то не через безопасное соединение в интернете. Я очень хотел ту пиццу, мне было без разницы тогда на безопасность.

Комбинаторные оптимизации

Метод имитации отжига

Этот метод пришёл из физики, основная его идея заключается в том, чтобы вероятностно менять решения с «понижающейся» температурой T. Давайте чуть подробнее:

- Сравниваем текущее значение S_* с наилучшим найденным (S). Если нашли лучше, меняем глобально наилучшее.
- Если наше состояние хуже, на самом деле, нам есть смысл менять на это состояние, только уже с какой-то вероятностью, которая зависит от температуры и, собственно, от значений. Обычно берут вероятность такой:

  $$p = \exp\left\{\frac{S_* - S}{T}\right\} \text{ подойдёт для задачи максимизации.}$$

- После этого, если мы приняли новое состояние, понижаем температуру и идём в начало алгоритма, иначе генерируем новое S_* и делаем так, пока не примется новое состояние.
- Когда температура становится близкой к 0, нужно выйти из программы.

В итоге, данный алгоритм способен решать многие сложные задачи комбинаторных оптимизаций, но при этом работать он будет довольно-таки быстро, в силу превосходства своей эффективности над полиномиальными аналогами.

Генетический алгоритм

Генетический алгоритм позволяет решать некоторые трудные задачи методом ошибок за разумное время.

Есть несколько фаз алгоритма:

- **Создание популяции.** Обычно это какие-то рандомные решение нашей задачи, которые могут иметь много ошибок.
- **Размножение.** Тут всё как у людей. Мы скрещиваем некоторые особи (обычно сильные особи) вместе, чтобы получить лучшее поколение.
- **Мутация.** Тут природа говорит, что мутации иногда хорошо влияют на организмы. Мы просто берем некоторые особи и мутируем их с помощью какого-то заранее определенного алгоритма. Да, могут получиться плохие особи, но есть вероятность, что получатся хорошие.
- **Отбор.** Мы отбираем самых лучших, те, кто пойдут дальше повторять этот процесс.

Иногда такой алгоритм приносит правильные решения.

Далее материала не было на лекции.

Приведём пример работы генетического алгоритма в задаче о правильной расстановке ферзей.

Берем какую-нибудь перестановку, что в строках и столбцах ровно по 1 ферзю. Генерируем, например, 100 таких перестановок.

Потом считаем количество ошибок, то есть количество пар, которые бьют друг друга (это можно сделать за $\mathcal{O}(n)$, пройдясь по диагоналям).

После этого выбираем хороших особей — примерно половину, у которых ошибок меньше всего. Берем любые 2, смотрим, какие элементы у них совпадают, оставляем их в предположении, что они являются «сильными» генами, а остальное всё перемешиваем между собой. Так делаем с каким-то количеством пар, потом выбираем, например, одну особь, мутируем её — меняем 2 любых элемента местами.

И это должно работать достаточно быстро. Для $n = 200$, скажем, такой алгоритм может приносить успех.

Опять же считаем количество ошибок и выбираем лучших 100. Повторяем алгоритм, пока не найдём нужное решение.

Генетический алгоритм ничего не доказывает, он лишь может работать как природа. Мы можем находить какие-то хорошие решение с его помощью за более разумное время.

Байка от Глебаса: В конце 40-ых годов появилась компания RAND, которая одна из первых вообще придумала работать со случайными числами (*Прим. и даже сгенерировала огромный список случайных цифр*). Вообще, это была первая компания, которая моделировала процессы с помощью случайных чисел.

Лекция 6 от 30.09.2016. Разбор некоторых интересных задач по матроидам

Здесь мы разберем три важных задачи, две из которых, скорее всего Глеб включит куда-нибудь (экзамен или что-то такое).

Я везде отождествляю элементы как одноэлементные множества.

Матроид паросочетаний

Лемма 1. *Пара $\langle V, I\rangle$, где V — множество вершин некоторого графа $G = (V, E)$, и $B \in I$, если существует паросочетание, покрывающее множество B, является матроидом. Также его называют матроидом паросочетаний.*

Доказательство. Первые два свойства матроида и правда ясны без объяснения (проделайте сами).

Будем доказывать третье свойство.

Давайте возьмём два множества вершин $B_1, B_2 \in I$ такие, что $|B_1| < |B_2|$.

Пусть X_1, X_2 — паросочетания, насыщающие B_1 и B_2 соответственно.

Есть два случая:

1. Если существует элемент $x \in B_2 \setminus B_1$, насыщенный в X_1, то всё отлично, так как X_1 насыщает $B_1 \cup x$

2. Теперь мы предполагаем, что все $x \in B_2 \setminus B_1$ не задействованы в вершинах в X_1. Рассмотрим подграф $G' = X_1 \triangle X_2$ — симметрическая разность ребер. Теперь степень каждой вершины не более двух, поэтому наш граф разбивается на циклы и цепочки, притом в циклах идёт чередование ребер, в цепочках тоже.

Ясно, что в каждой цепи концевые вершины будут в одном паросочении не насыщены, так как иначе будет четное число ребер, содержащую данную вершину в G', значит это не конец пути.

Так как ни одна вершина из $B_2 \setminus B_1$ (а их там хотя бы 1) не насыщена первым паросочетанием, то эти вершины могут быть только концами путей.

Докажем, что $|B_2 \setminus B_1| > |B_1 \setminus B_2|$. Пусть $|B_1 \cap B_2| = \alpha$, тогда $|B_2 \setminus B_1| = |B_2| - \alpha, |B_1 \setminus B_2| = |B_1| - \alpha$, так как из каждого множества мы убираем только элементы из пересечения, откуда $|B_2 \setminus B_1| > |B_1 \setminus B_2|$.

Так как $|B_2 \setminus B_1| > |B_1 \setminus B_2|$, то существует путь P (не цикл), начинающаяся в $B_2 \setminus B_1$ и заканчивающаяся **не** в $B_1 \setminus B_2$. Заканчиваться путь в $B_1 \cap B_2$ не может, так как в этом множестве все вершины имеют четную степень в G', значит этот путь заканчивается вне B_1.

Докажем, что $X_1 \triangle P$ будет паросочетанием, насыщающим $B_1 \cup v_{start}$, где $v_{start} \in B_2 \setminus B_1$ и начинает путь P с одного из концов. Фактически мы написали, что мы поменяем цвета этих рёбер, то есть те рёбра, которые были в $X_1 \cap P$, больше не ребра паросочетания, а ребра из X_2 на этом пути будут теперь рёбрами $X_1 \triangle P$. Все вершины из B_1, лежащие внутри пути (не на концах) останутся быть вершинами паросочетания $X_1 \triangle P$. Единственная проблема может возникнуть с концами — v_{start}: теперь вершина паросочетания $X_1 \triangle P$, а другой конец не входил в B_1, значит даже если там не берется ребро, то ничего страшного, эта вершина нам и не нужна была. □

Worst-out Greedy

Лемма 2 (Worst-out Greedy)**.** *Пусть дан некоторый матроид $M = \langle M \rangle$, элементам присвоены некоторые стоимости $c(w)$, причём элементы $w_1, \ldots, w_n$ упорядочены так, что $0 \leqslant c(w_1) \leqslant \ldots \leqslant c(w_n)$. Рассмотрим следующий жадный алгоритм:*

Algorithm 11 Worst-out greedy

```
1: F ← X
2: for i ← 1 to n do
3:     if F \ w_i содержит базу then
4:         F ← F \ w_i
```

Тогда такой алгоритм строит базу максимального веса.

Введем понятие *двойственного матроида.*

Двойственный матроид к $M = \langle X, I \rangle$ это матроид $M^* = \langle X, I^* \rangle$, где $I^* = \{A \mid \exists\ B \in \mathfrak{B} : A \cap B = \varnothing\}$.

Докажем, что это матроид:

Доказательство. Проверим все 3 свойства.

1. Пустое множество лежит в этом множестве.

2. Пусть $A_1 \subseteq A_2$ и $A_2 \in I^*$, тогда $A_2 \cap B = \varnothing$, тогда $A_1 \cap B = \varnothing$, так как подмножество пустого множества будет пустым.

3. Возьмём произвольные A_1, A_2 такие, что $|A_1| < |A_2|$. Из определения следует, что $X \setminus A_2$ сожержит какую-то базу — пусть это будет база B. Мы знаем, что $B \setminus A_1 \subseteq X \setminus A_1$, так как $B \subseteq X$.

Также пусть $B_1' \subseteq X \setminus A_1$ — та база, которая содержится в дополнении A_1.

Рассмотрим множества $B \setminus A_1$ и B_1'. Будем дополнять по третьей аксиоме матроида M первое множество, пока оно не станет базой. Пусть мы в итоге получили $B' = B \setminus A_1 \cup \{x_1, \ldots, x_{|A_1|}\}$. Мы получили множество B', которое является базой, содержит $B \setminus A_1$ и содержится в $X \setminus A_1$, так как мы добавляли элементы только из B_1', а $B_1' \subseteq X \setminus A_1$.

Отлично, мы нашли базу B', такую что $B \setminus A_1 \subseteq B' \subseteq X \setminus A_1$.

Предположим, что $A_2 \setminus A_1 \subseteq B'$.

Также нам понадобится факт $B \cap A_1 \subseteq A_1 \setminus A_2$, который объясняется тем, что B не имеет общих элементов с A_2 по определению.

Теперь выпишем цепочку неравенств и равенств и докажем каждое из них поочереди.

$|B| = |B \cap A_1| + |B \setminus A_1|$ — простое свойство из теории множеств (это просто все элементы лежащие в B и в первом случае и в A_1, а во втором не в A_1).

$|B \cap A_1| + |B \setminus A_1| \leqslant |A_1 \setminus A_2| + |B \setminus A_1|$ — см. свойство выше.

$|A_1 \setminus A_2| + |B \setminus A_1| < |A_2 \setminus A_1| + |B \setminus A_1|$ — так как $|A_1 \setminus A_2| < |A_2 \setminus A_1|$, так как $|A_2| > |A_1|$ (см. факт из первой леммы).

$|A_2 \setminus A_1| + |B \setminus A_1| \leqslant |B'|$ — это действительно верно, так как два множества слева под модулями не пересекаются (так как B и A_2 не пересекаются). И каждое из этих множеств является подмножеством B' (первое по предположению, второе доказано выше).

Откуда $|B| < |B'|$, что неверно, так как все базы равномощны между собой.

Значит, $A_2 \setminus A_1 \not\subseteq B'$, откуда существует элемент $z \in A_2 \setminus A_1$ такой, что $z \notin B'$, откуда $(A_1 \cup z) \cap B' = \varnothing$, что нам и требуется.

□

Теперь перейдём к доказательству леммы.

Доказательство. **Свойство двойственности баз.**

Понятно, что база матроида M^* будет дополнением всех элементов из базы $\mathfrak{B}$, потому что для каждого дополнения существует база, с которой он пересекается по пустому множеству. И если существует множество из I^*, которое по мощности больше, чем мощность дополнения любого элемента из $\mathfrak{B}$, то такое множество по принципу Дирихле пересекается со всеми элементами $\mathfrak{B}$, а значит это множество не лежит в I^*. И если есть элемент базы M^*, который является недополнением какого-то элемента из базы $\mathfrak{B}$, то такое множество тоже пересекается со всеми базами, так как базы имеют одинаковую мощность в обоих случаях.

Теперь чуточку перепишем алгоритм, данный в лемме.

Algorithm 12 Модификация алгоритма на матроиде M^*(именно на двойственном)

```
1: F* ← ∅
2: for i ← 1 to n do
3:     if F* ∪ w_i ∈ I* then
4:         F* ← F* ∪ w_i
```

Заметим, что этот алгоритм возьмёт все те и только те элементы, которые алгоритм из леммы выкинет. Докажем это по индукции:

Утверждение. На каждом шаге i от 0 до n выполняется $F^* \cup F = X, F \cap F^* = \varnothing$.

База $i = 0$. На нулевом шаге у нас $F^* \cup F = X, F^* \cap F = \varnothing$.

Переход $i - 1 \to i$:

Если F выкидывает w_i, то существует база B_i такая, что $B_i \subseteq F \setminus w_i$. А значит, в двойственном матроиде $F^* \cup w_i \subseteq X \setminus B_i$ (здесь мы пользуемся предположением индукции), то есть $F^* \cup w_i$ является подмножеством какой-то базы матроида M^*, а значит $F^* \cup w_i \in I^*$. Это всё следует из свойств, которые мы доказывали выше. В обратную сторону аналогично: если w_i добавляется к F^*, то и F выкинет w_i. Переход доказан.

В первой лекции было, что алгоритм из решения приносит максимальную сумму, если веса расположены в невозрастающем порядке. **Дословно** переносятся все утверждения, когда порядок неубывающий, только в итоге мы получим минимальную сумму множества.

Поэтому мы получили множество F^*, которое набрало минимальную сумму. Следовательно F наберет максимальную, так как $F \cup F^* = X, F \cap F^* = \varnothing$, откуда $c(F) + c(F^*) = c(X)$, а значит $c(F)$ набирает максимум. Так как F^* будет элементом базы (из следствия на первой лекции) в матроиде M^*, то и F будет элементом базы в M из-за свойства двойственности баз (см. выше). □

Лемма об обмене

Лемма 3 (Лемма об обмене)**.** *Пусть имеются 2 базы B_1, B_2, тогда $\forall\, x \in B_1 \setminus B_2\ \exists\, y \in B_2 \setminus B_1$, такие, что $(B_1 \setminus x) \cup y \in I$ и $(B_2 \setminus y) \cup x \in I$.*

Здесь у нас будет матроид $M = \langle S, I \rangle$

Введем понятие *цикла. Цикл* — это наименьшее по включению зависимое множество, то есть все собственные подмножества цикла принадлежат I.

Все ссылки на леммы, которые я делаю в этой теореме, это ссылки на леммы «задачи».

Докажем такие леммы:

Лемма задачи 1. $r(A) + r(B) \geqslant r(A \cup B) + r(A \cap B)$.

Доказательство. Пусть X максимальное независимое подмножество $A \cap B$. Возьмём Y' — максимальное независимое множество $A \cup B$. Заметим, что $|X| \leqslant |Y'|$, поэтому будем по третьей аксиоме матроидов добавлять к X элементы из $Y' \setminus X$. Получим максимальное независимое подмножество $A \cup B$, которое содержит X. Пусть это подмножество будет $Y \Rightarrow X \subseteq Y$.

Разделим Y на 3 категории множеств — $Y = X \cup V \cup W$ так, что $V \subseteq A \setminus B, W \subseteq B \setminus A$. Так и будет, потому что из пересечения мы не могли добавить к X больше элементов, иначе X был бы не максимальным по включению.

Получаем, что $X \cup V$ — независимо в A (аксиома два), аналогично $X \cup W$ независимо в B, откуда $r(A) \geqslant |X \cup V|, r(B) \geqslant |X \cup W|$.

Откуда $r(A) + r(B) \geqslant |X \cup V| + |X \cup W| = |X| + |X| + |W| + |V| = r(A \cup B) + r(A \cap B)$. □

Лемма задачи 2. *Пусть C_1, C_2 — различные циклы одного матроида и $x \in C_1 \cap C_2$. Тогда существует цикл $C_3 \subseteq (C_1 \cup C_2) \setminus x$.*

Доказательство. Покажем, что $r(C') < |C'|$, где $C' = (C_1 \cup C_2) \setminus x$.

Так как C_1, C_2 — различные циклы, то $C_1 \cap C_2$ является собственным подмножеством C_1, то есть $r(C_1 \cap C_2) = |C_1 \cap C_2|$.

Также мы знаем, что $r(C_1) = |C_1| - 1, r(C_2) = |C_2| - 1$, так как это циклы.

По лемме 1 получаем

$$r(C_1 \cup C_2) \leqslant r(C_1) + r(C_2) - r(C_1 \cap C_2) = |C_1| + |C_2| - |C_1 \cap C_2| - 2 = |C_1 \cup C_2| - 2 < |C'|$$

Также мы знаем, что $r(C') \leqslant r(C_1 \cup C_2)$, так как $C' \subseteq C_1 \cup C_2$.

Откуда $r(C') < |C'|$. Значит существует цикл в таком множестве. □

Лемма задачи 3. *Если A независимое множество, а $x \in S$, тогда $A \cup x$ содержит не более одного цикла.*

Доказательство. Пусть есть 2 различных цикла $C_1, C_2 \subseteq (A \cup x)$. Они оба содержат x, иначе они независимы.

Рассмотрим множество $(C_1 \cup C_2) \setminus x$. По лемме 2 в этом множестве есть цикл, а значит $(C_1 \cup C_2) \setminus x$ зависимо. Но $(C_1 \cup C_2) \setminus x \subseteq A$, что противоречит независимости A. □

Также введём ещё понятие для любого подмножества $A \subseteq S$ — $span(A) = \{s \in S : r(A \cup s) = r(S)\}$. Тривиально, что $A \subseteq span(A)$.

Лемма задачи 4. a) Если $A \subseteq B$, то $span(A) \subseteq span(B)$

b) Если $e \in span(A)$, то $span(A \cup e) = span(A)$.

Доказательство. a) Пусть $e \in span(A)$. Если $e \in B$, то отсюда сразу следует, что $e \in span(B)$ (см. тривиальное свойство). По лемме 1 следует, что $r(A \cup e) + r(B) \geqslant r((A \cup e) \cap B) + r((A \cup e) \cup B)$

Откуда сразу следует, что $r(A \cup e) + r(B) \geqslant r(A) + r(B \cup e)$, так как $e \notin B$, поэтому $(A \cup e) \cap B = A, (A \cup e) \cup B = B \cup e$. Из определения следует, что $r(A \cup e) = r(A)$, значит

$r(B) \geqslant r(B \cup e)$, но мы знаем, что все независимые подмножества B являются независимыми множествами $B \cup e$, значит в другую сторону неравенство выполняется очевидно.

Поэтому $r(B) = r(B \cup e)$, откуда $e \in span(B)$.

b) Из пункта a) следует, что $span(A) \subseteq span(A \cup e)$. Поэтому нам надо доказать для каждого $f \in span(A \cup e)$, что оно лежит в $span(A)$. Опять воспользуемся леммой 1:

$r(A \cup e) + r(A \cup f) \geqslant r((A \cup e) \cap (A \cup f)) + r(A \cup e \cup A \cup f)$

Случай $e = f$ очевиден, пусть $e \neq f$.

Тогда

$r(A \cup e) + r(A \cup f) \geqslant r(A) + r(A \cup e \cup f)$

$r(A \cup e) = r(A)$ по определению. Значит

$r(A \cup f) \geqslant r(A \cup e \cup f)$ откуда аналогично следует, что $r(A \cup f) = r(A \cup e \cup f)$.

Откуда $e \in span(A \cup f)$, но $e \in span(A)$ и $f \in span(A \cup e)$, значит $r(A) = r(A \cup e) = r(A \cup e \cup f) = r(A \cup f)$ — последнее равенство следует из выше доказанного. Поэтому $f \in span(A)$, что и требовалось. □

Доказательство леммы.

Доказательство. Пусть $x \in B_1 \setminus B_2$, тогда $B_2 \cup x$ содержит ровно один цикл по лемме 3 (так как B_2 независимо, а $B_2 \cup x$ зависимо). Пусть этот цикл будет C. Мы знаем, что $x \in span(C \setminus x)$, так как добавление не меняет ранг. Поэтому $x \in span((B_1 \cup C) \setminus x)$ (см. лемма 4a). По лемме 4b следует, что $span((B_1 \cup C) \setminus x) = span(B_1 \cup C) = S$, так как B_1 является базой. Получается, что какой-бы элемент к максимально независимому множеству в $(B_1 \cup C) \setminus x$ ни добавляй, получим, что ранг меняться не будет. Это возможно только если $r((B_1 \cup C) \setminus x) = |B_1|$, иначе мы могли бы получить противоречие с аксиомой 3 матроидов.

Пусть B' — база, содержащаяся в $(B_1 \cup C) \setminus x$.

По аксиоме 3 (для $B_1 \setminus x$ и B') в $B' \setminus (B_1 \setminus x)$ существует y, что $(B_1 \setminus x) \cup y$ — база.

$B' \setminus (B_1 \setminus x) \subseteq ((B_1 \cup C) \setminus x) \setminus (B_1 \setminus x)$ (см. 2 абзаца выше).

$((B_1 \cup C) \setminus x) \setminus (B_1 \setminus x) \subseteq C \setminus x$ — легко проверяется кругами Эйлера. То есть $y \in C \setminus x$. Но $C \subseteq (B_2 \cup x)$, поэтому $C \setminus x \subseteq B_2$. Значит $y \in B_2$, значит $y \in B_2 \setminus B_1$ (см. выше, почему $y \notin B_1$). Также важно отметить, что $x \neq y$, так как $B' \subseteq (B_1 \cup C) \setminus x$.

Докажем, что $(B_2 \setminus y) \cup x$ — тоже база. Допустим, что это не так. Тогда существует цикл $C' \subseteq (B_2 \setminus y) \cup x \subseteq B_2 \cup x$. Причем $C' \neq C$, так как $y \in C$ (см. выше), но $y \notin C'$. Значит у $B_2 \cup x$ существовало 2 различных цикла, что невозможно по лемме 3. Что и завершает наше доказательство. □

Лекция 7 от 04.10.2016. Венгерский алгоритм решения задачи о назначениях

Формальная постановка задачи

Оригинальная постановка задачи о назначениях выгдядит так: нам дана матрица $n \times n$

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix},$$

где строки отвечают за работников, которые требуют определенную сумму за то или иное задание (столбцы).

Требуется найти такую перестановку $p(i)$, что $\sum\limits_{i=1}^{n} a_{ip(i)} \to \min$. Обозначим эту задачу за A'.

Немного линейного программирования

Выпишем задачу линейного программирования (назовём её B').

$$\begin{aligned} \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_{ij} &\to \min \\ \forall i \in \overline{1\dots n} \implies \sum_{j=1}^{n} x_{ij} &= 1, \\ \forall j \in \overline{1\dots n} \implies \sum_{i=1}^{n} x_{ij} &= 1, \\ x_{ij} &\geqslant 0. \end{aligned}$$

Ясно, что если есть решение задачи A', то оно является каким-то решением задачи B', так как мы в каждой строке и столбце у нас ровно одно $x_{ij} = 1$, поэтому все такие решения подойдут под B'. Значит решение задачи B' не хуже, чем у A'. Выпишем этой задаче двойственную (что, конечно же, было проделано много раз на курсе дискретной математики). Двойственные переменные u_i отвечают за строки, v_i за столбцы. Напомним пару двойственных задач:

$$\begin{cases} cx \to \max \\ Ax \leqslant b. \end{cases} \quad \text{двойственна} \quad \begin{cases} yb \to \min \\ yA = c, \\ y \geqslant 0. \end{cases}$$

Поэтому двойственная к B' будет

$$\begin{aligned} \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j &\to \max \\ \forall i,j \in \overline{1\dots n} \implies u_i + v_j &\leqslant a_{ij}. \end{aligned}$$

Это действительно так, потому что в строках матрицы A задачи B' будет лишь 2 элемента — отвечающее за строку и столбец, в котором переменная x_{ij} находится.

Обозначим эту задачу за C'.

Как мы знаем из курса дискретной математики, двойственная и прямая задачи имеют одно и то же оптимальное значение, поэтому оптимальные значения B' и C' совпадают. Также легко убедиться, что каждая задача имеет хоть какое-то решение.

Заметим, что любое решение задачи A' является решением задачи C'. Действительно,

$$\sum_{i=1}^{n} a_{ip(i)} \geqslant \sum_{i=1}^{n} u_i + v_{p(i)} = \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j.$$

То есть любое допустимая перестановка не меньше, чем любое значение целевой функции задачи C', поэтому если мы предъявим хоть какое-то решение, что значение целевой функции задачи C' совпадает с каким-то из задачи A', то мы найдём оптимальное решение (достигается равенство выше). И тогда все оптимальные решения задач A', B', C' совпадают. Предъявим конструктивно алгоритм построения решения C'.

Венгерский алгоритм

Нарисуем двудольный граф, где левая доля отвечает переменным u_i, правая за v_j.

Определение 1. *Назовём ребро* (i, j) ***жёстким****, если* $u_i + v_j = a_{ij}$.

Заметим, что если мы найдём совершенное паросочетание на жёстких ребрах, то мы найдём какую-то перестановку $p(i)$, тогда мы найдём какое-то решение, где $u_i + v_{p(i)}$ будет входить в сумму целевой функции, как $a_{ip(i)}$, а значит мы найдём какое-то решение задачи A'. Поэтому надо всего лишь «подвигать значения переменных», чтобы получить совершенное паросочетание на жёстких ребрах.

Мы будем пытаться добавлять вершины левой доли по одной на каждом шаге, в предположении, что на предыдущих шагах мы смогли всё добавить. Пусть мы добавляем вершину i, а на всех $i - 1$ предыдущих максимальное паросочетание можно построить. Запустим алгоритм Куна из i-ой вершины. Если мы нашли паросочетание и на i-ой вершине, то отлично, мы увеличили паросочетание. Теперь допустим, что не смогли увеличить.

Обозначим за R_+ — все вершины, которые мы достигли алгоритмом Куна в правой доле из вершины i, L_+ в левой. Аналогично определяются множества вершин R_- и L_-. В R_+ вершин меньше, так как для любой вершины из R_+ алгоритм Куна посетит вместе с ней какую-то вершину левой доли из паросочетания.

Заметим, что нет жёстких ребёр из L_+ в R_-, так как иначе мы могли бы найти удлиняющий путь (просто пройдя как-то по этой вершине).

Теперь введём обозначение $d = \min(a_{ij} - u_i - v_j), i \in L_+, j \in R_-$. Как мы только что показали, $d > 0$, так как ни одно ребро не жёсткое.

Теперь применим данные действия к алгоритму:

- Увеличим все u_i на d для $i \in L_+$,
- Уменьшим все v_j на d для $j \in R_+$.

Заметим, что целевая функция увеличилась хотя бы на d, так как в L_+ больше вершин, чем в R_+. Осталось проверить, что все неравенства линейной задачи C' сохранятся:

- На ребрах из $L_+ \to R_+$ ничего не изменится, так как мы добавили d и вычли d, в частности жёсткие ребра останутся жёсткими.
- На ребрах из $L_- \to R_+$ мы уменьшим $u_i + v_j$ на d, но уменьшать нам не запрещено, так как неравенство сохранится,
- На ребрах из $L_+ \to R_-$ мы по определению не получим противоречия в неравенствах $u_i + d + v_j \leqslant a_{ij}$, так как $d \leqslant a_{ij} - u_i - v_j$,
- Ребра из $L_- \to R_-$ мы никак не модифицировали, поэтому всё корректно будет и тут.

Поэтому из L_+ после пересчёта переменных будет ещё одно жёсткое ребро, которое ведёт в R_-, а значит алгоритмом Куна мы строго увеличим количество посещенных вершин.

Бесконечно этот процесс не может происходить, так как мы не можем строго увеличивать бесконечно количество посещённых вершин. Максимум через n действий мы получим, что R_- состоит из одной вершины, а все вершины левой доли мы посетили, поэтому мы сможем найти удлиняющую цепочку.

Понижение асимптотики алгоритма до $\mathcal{O}(n^3)$

Сейчас наш алгоритм работает за $\mathcal{O}(n^4)$. Действительно, мы должны добавить n вершин, каждый раз запуская алгоритм Куна от вершины не более n раз, пересчет переменных происходит за $\mathcal{O}(n^2)$ и за $\mathcal{O}(n^2)$ будет работать одна фаза алгоритма Куна, поэтому асимптотика $\mathcal{O}(n^4)$, но, можно и быстрее, давайте поймём как проапгрейдить наш алгоритм.

Ключевая идея: теперь мы будем добавлять в рассмотрение строки матрицы одну за одной, а не рассматривать их все сразу. Таким образом, описанный выше алгоритм примет вид:

- Добавляем в рассмотрение очередную строку матрицы a.
- Пока нет увеличивающей цепи, начинающейся в этой строке, пересчитываем переменные.
- Как только появляется увеличивающая цепь, чередуем паросочетание вдоль неё (включая тем самым последнюю строку в паросочетание), и переходим к началу (к рассмотрению следующей строки).

Чтобы достичь требуемой асимптотики, надо реализовать шаги 2-3, выполняющиеся для каждой строки матрицы, за время $\mathcal{O}(n^2)$.

Заметим, что если вершина была достижима из i-ой, то после пересчётов переменных она останется достижимой, так как жёсткие ребра остаются жесткими, и пересчет переменных выполняется $\mathcal{O}(n)$ раз.

Также заметим, что для проверки наличия увеличивающей цепочки нет необходимости запускать алгоритм Куна заново после каждого пересчёта переменной. Вместо этого можно оформить light версию обхода Куна: после каждого пересчёта переменной мы просматриваем добавившиеся жёсткие рёбра и, если их левые концы были достижимыми, помечаем их правые концы также как достижимые и продолжаем обход из них.

Введём также величину c_j для всех j:

$$c_j = \min_{i \in L_+}(a_{ij} - u_i - v_j)$$

Тогда наша $d = \min\limits_{j \in R_-} c_j$. Это и есть в точности те вершины правой доли, которые мы ещё не посетили, соединенных с вершинами левой доли L_+. Это делается за $\mathcal{O}(n)$.

Теперь легко изменять c_j при расстановке потенциалов — надо просто вычесть d из всех $c_j, j \in R_-$. Если мы добавляем в левой доли вершину k, надо лишь обновить все $c_j = \min(c_j, a_{kj} - u_k - v_j)$. Это делается за $\mathcal{O}(n)$.

А инициализировать c_j надо при добавлении i-ой вершины, как $c_j = a_{ij} - u_i - v_j$, так как пока только 1 вершина посещена — эта вершина на i-ой фазе алгоритма.

Теперь поймём, почему теперь алгоритм работает за $\mathcal{O}(n^3)$. Есть, как и в прошлой версии, внешняя фаза — добавление вершин левой доли за $\mathcal{O}(n)$. Теперь обновление переменных выполняется за $\mathcal{O}(n)$, обновление массива c_j происходит при каждом обновлении за $\mathcal{O}(n)$, что даёт $\mathcal{O}(n^2)$ на каждом шаге. Плюс ещё мы должны во внешней фазе запустить алгоритм Куна, чтобы найти паросочетание (мы уже уверены, что оно есть), что тоже $\mathcal{O}(n^2)$. В итоге $\mathcal{O}(n^3)$. Приведем псевдокод:

Algorithm 13 Венгерский алгоритм

```
1:  M ← ∅                                   ▷ Поддерживающееся паросочетание на жёстких рёбрах
2:  for i ← 1 to n do
3:      c_j ← a_ij − u_i − v_j                                          ▷ По всем j
4:      R_+ ← ∅
5:      L_+ ← {i}
6:      flag ← true
7:      while flag do
8:          d ← min c_j                          ▷ По всем j ∈ R_− — надо найти это самое d
9:          u_k ← u_k + d                                  ▷ По всем k ∈ L_+ — см. выше
10:         v_k ← v_k − d                                  ▷ По всем k ∈ R_+ — см. выше
11:         c_k ← c_k − d                                  ▷ По всем k ∈ R_− — см. выше
12:         for k ∈ R_− do
13:             if c_k = 0 then                    ▷ Значит вершина стала достижимой
14:                 if k ∈ Right(M) then           ▷ Вдруг, наша вершина оказалась уже в
    паросочетании, то мы ещё не можем увеличить
15:                     L_+ ← L_+ + Left_neighbour(k)   ▷ Добавляем посещенную в L_+, которая
    насыщена паросочетанием M
16:                     R_+ ← R_+ + k              ▷ Суммарно добавлений будет не более O(n)
17:                     q ← Left_neighbour(k)
18:                     c_p ← min(c_p, a_qp − u_q − v_p)   ▷ надо обновить после добавления по всем
    p ∈ R_−
19:                 else
20:                     flag ← false    ▷ уже точно знаем, что можем увеличить паросочетание
21:                     break                ▷ надо выйти вообще, чтобы запустить алгоритм Куна
22:     M ← kunh_algo(i)
```

Действительно, в L_+ и R_+ мы добавляем не больше, чем $\mathcal{O}(n)$ раз, значит и восемнадцатая строка выполнится не более $\mathcal{O}(n)$ раз, значит в цикле с седьмой строки мы в целом будем

выполнять не более $\mathcal{O}(n^2)$ действий. Также алгоритм Куна будет выполняться не более $\mathcal{O}(n^2)$ действий на каждой итерации. А значит общая асимптотика равна $\mathcal{O}(n^3)$.

Можно немного сэкономить на практике и не писать алгоритм Куна, а запоминать, из какой вершины левой доли мы пошли в правую, а вместо алгоритма Куна делать всего $\mathcal{O}(n)$ действий, идя обратно по пути.

Байка от Глебаса: Была одна команда на ACM, вроде из Саратова. На финал можно было принести с тобой несколько листов заготовленного кода. Одна из задач была на венгерский алгоритм, и у команды была распечатка по этому алгоритму. В итоге она у них не заходила, потому что когда они тестили у себя в контесте, видимо, тесты были слабые. Мораль: ставьте asserts везде, где можно, чтобы убедиться, что этот алгоритм работает корректно.

Венгерский алгоритм на прямоугольной матрице

Будем считать, что $n \leqslant m$, иначе просто транспонируем матрицу.

Здесь всё тоже самое, только давайте теперь оценим время работы, если у нас матрица размера $n \times m$. Заметим, что в алгоритме мы будем добавлять не больше, чем $\min(n, m) = n$ вершин, так как мы либо ничего не делаем, либо добавляем по одному элементу к L_+ и R_+ одновременно. Поэтому тут $\mathcal{O}(n)$, но восемнадцатую строку мы будем обновлять за $\mathcal{O}(m)$, поэтому внешний цикл с седьмой строки будет выполняться не более $\mathcal{O}(nm)$ действий, алгоритм Куна тоже будет выполняться за $\mathcal{O}(nm)$ (максимальное количество рёбер столько). Поэтому асимптотика будет равна $\mathcal{O}(n^2m)$, что может быть применимо для больших m и n поменьше.

Также для совсем маленьких n, можно оставить лучших n в каждой строке (ведь если есть решение, что в данной строке кто-то не из n лучших, то остальные работники забрали не более $n - 1$ столбец в матрице на свои лучшие ответы, а значит по принципу Дирихле мы можем взять получше ответ для этой строки). Поэтому матрица осталась максимум $n \times n^2$, значит такой алгоритм будет работать за $\mathcal{O}(n^4 + nm)$.

Лекция 8 от 11.10.2016. Сегментация и кластеризация изображений с помощью потоковых алгоритмов

Постановка задачи

Рассмотрим любую картинку (Айрат, привет):

Рис. 1: Произвольная картинка

И мы хотим отделить фон от человека. То есть присвоить каждому пикселю матрицы $n \times m$ какой-то label, к какому классу относится — фон или человек, например.

Фактически это единственный алгоритм машинного обучения, где используются алгоритмы на потоках.

Прим. Те, кто не помнят, что такое поток, могут закрывать эту лекцию.

Минимизация парно-сепарабельной энергии от бинарных переменных

Пусть у нас задан неориентированный граф $G(V, E)$. Для каждого $i \in V$ пусть x_i могут принимать значения только из $\{0, 1\}$.

Определение 1. *Назовём* ***энергией*** *(обозначение I) функцию из* $\{0,1\}^{|V|} \to \mathbb{R}$*:*

$$I(X) = \sum_{i \in V} \theta_i(x_i) + \sum_{(i,j) \in E} \theta_{ij}(x_i, x_j) + \theta_0,$$

где θ_i, θ_{ij} — какие-то потенциалы, а θ_0 — константа.

И наша задача заключается в том, что минимизировать $I(X)$. Известно, что если не вводить никаких дополнительных ограничений, то задача минимизации энергии является **NP**-трудной.

Давайте поймём, как это относится к сегментации. На выборке из огромного числа изображений мы можем с уверенностью говорить, о том, какие пиксели находятся рядом, какие далеко по цвету, поэтому можем поставить какие-то веса на рёбрах. После этого сегментировать изображение, чтобы были в одной и другой части как можно более тёплые цвета. Рассмотрим частный случай потенциалов, в котором задача становится полиномиальной:

- $\forall\ i \in V, \theta_i(0) \geqslant 0, \theta_i(1) \geqslant 0$;
- $\forall\ (i,j) \in E \Rightarrow \theta_{ij}(0,0) = \theta_{ij}(1,1) = 0, \theta_{ij}(0,1) \geqslant 0, \theta_{ij}(1,0) \geqslant 0$.

Тогда энергию можно задать так (легко проверить все случаи):

$$I(X) = \sum_{i \in V} (x_i \theta_i(1) + (1 - x_i)\theta_i(0)) + \sum_{(i,j) \in E} (x_i(1 - x_j)\theta_{ij}(1,0) + x_j(1 - x_i)\theta_{ij}(0,1)) + \theta_0,$$

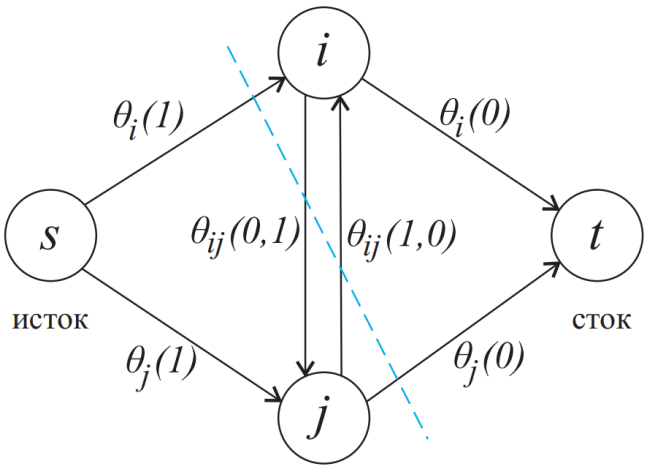


Рис. 2: Граф, построенный для минимизации энергии от двух переменных x_i, x_j. Разрез, отображенной пунктирной линией соответствует присваиванию $x_i = 1, x_j = 0$. Величина разреза составляет $\theta_i(1) + \theta_j(0) + \theta_{ij}(1,0)$

.

Теперь построим ориентированный граф $\overline{G} = (\overline{V}, \overline{E})$ по следующим правилам:

- В $\overline{V} = V \cup \{s, t\}$;
- Неориентированные рёбра делаем ориентированными в обе стороны, а для каждой вершины i проведем ещё ребра $(s, i), (i, t)$;
- $c(s, i) = \theta_i(1), c(i, t) = \theta_i(0)$, где $i \in V$;
- $\forall\ (i, j) \in E$ таким, что $i < j$ положим $c(i, j) = \theta_{ij}(0, 1), c(j, i) = \theta_{ij}(1, 0)$;
- Все вершины из V, которые попали в минимальный разрез S положим $x_i = 0$, остальным $x_i = 1$.

Тогда видно, что минимизация разреза эквивалентна этой задаче, что эквивалентно задаче максимального потока. Существует, конечно, много алгоритмов максимального потока, многие из них мы изучали, но в компьютерном зрении часто возникают алгоритмы Бойкова-Колмогорова и IBFS. С ними вы можете ознакомиться при желании самостоятельно.

Пример графа, построенного для энергии от 2-х переменных, и его разреза приведен на рис 2.

Репараметризация

Здесь мы рассмотрим, какие ещё энергии можно минимизировать при помощи разрезов графов. Назовём преобразования потенциалов, не меняющее энергию *репараметризацией*. Рассмотрим несколько видов репараметризаций:

- Вычитание константы — $\theta_i(0) \mathrel{-}= \delta, \theta_i(1) \mathrel{-}= \delta, \theta_0 \mathrel{+}= \delta$;
- Изменение потенциалов на ребрах. $\theta_{ij}(p, 0) \mathrel{-}= \delta, \theta_{ij}(p, 1) \mathrel{-}= \delta, \theta_i(p) \mathrel{+}= \delta$. Аналогично, если p на 2-ой координате.

Легко видеть из определения, что эти преобразования не меняют энергию.

Рассмотрим, что можно делать при помощи репараметризации потенциалов на ребрах. Для $(i, j) \in E$ пусть $\theta_{ij}(0, 0) = a$, $\theta_{ij}(1, 1) = b$, $\theta_{ij}(0, 1) = c$, $\theta_{ij}(1, 0) = d$.

После этого давайте 3 раза применим 2-ой пункт видов репараметризации:

- $\theta_{ij}(0, 0) \mathrel{-}= a, \theta_{ij}(0, 1) \mathrel{-}= a, \theta_i(0) \mathrel{+}= a$;
- $\theta_{ij}(0, 1) \mathrel{-}= (c - a), \theta_{ij}(1, 1) \mathrel{-}= (c - a), \theta_j(1) \mathrel{+}= c - a$;
- $\theta_{ij}(1, 1) \mathrel{-}= (b - c + a), \theta_{ij}(1, 0) \mathrel{-}= (b - c + a), \theta_i(1) \mathrel{+}= b - c + a$

Потом сделаем все потенциалы вершины неотрицательными по 1-ому пункту репараметризации. В итоге у нас ненулевым останется только $\theta_{ij}(1, 0) = d+c-a-b$. И если оно положительно, то мы можем применить наш алгоритм, то есть должно выполняться условие *субмодулярности*:

$$\theta_{ij}(0, 0) + \theta_{ij}(1, 1) \leqslant \theta_{ij}(0, 1) + \theta_{ij}(1, 0)$$

Данное условие вызвано тем, что для полиномиальной разрешимости задач о максимальном потоке и минимальном разрезе пропускные способности дуг графа должны быть неотрицательны.

α-расширение

Мы умели решать задачу только с одним объектом, теперь давайте попробуем приближенно решить задачу со многими объектами. Тот же граф, только теперь поставим в соответствие каждой вершине i — $y_i \in \{1, \dots, K\}$ — классы разбиения. Рассмотрим следующую энергию:

$$I_M(X) = \sum_{i \in V} \psi_i(x_i) + \sum_{(i,j) \in E} \psi_{ij}(x_i, x_j) + \psi_0,$$

Буква M, скорее всего, идёт от английского слова Many — много.

Алгоритм α-расширение минимизирует энергию при помощи выполнения шагов между разметками y, каждый из которых гарантированно не увеличивает значение энергии. Каждый шаг представляет собой задачу минимизации энергии бинарных переменных вида. Неформально каждый шаг позволяет каждой переменной из y либо присвоить выбранное значение α, либо оставить текущее значение (расширение метки α).

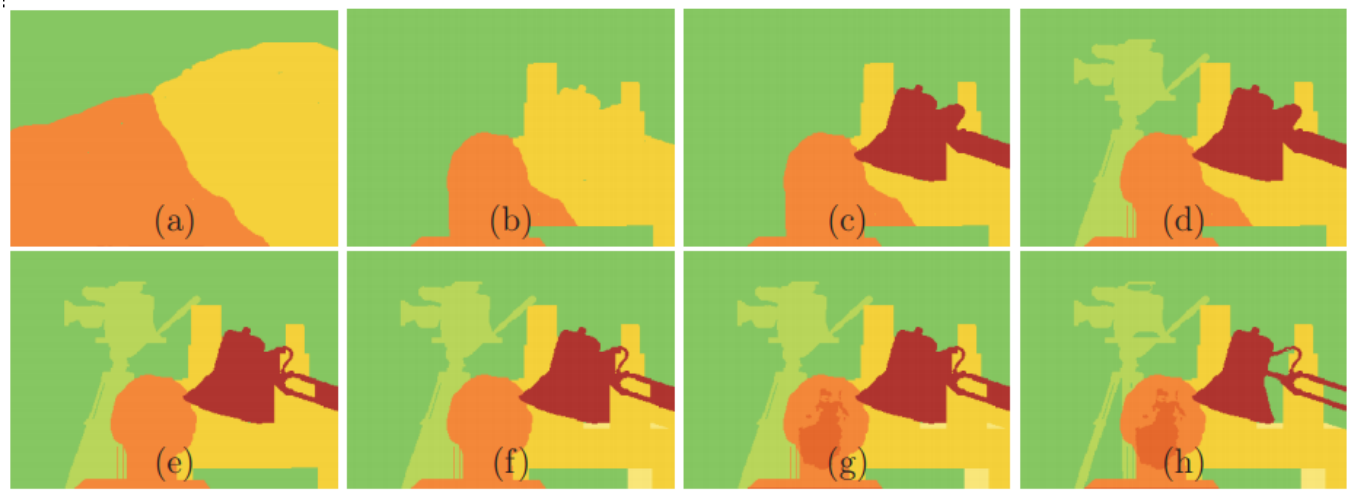


Рис. 3: Пример работы алгоритма α-расширения для задачи выровненного стерео. (a) – начальная разметка, далее последовательные расширения различных меток.

На каждом шаге алгоритма у нас есть текущее приближение y^0 и выбрана *расширяемая* метка $\alpha \in \{1, \dots, K\}$.

- Граф, потенциалы сначала одинаковы;
- Применяем алгоритм о минимальном разрезе, теперь, если $x_i = 0$, то оставляем y_i^0, а если $x_i = 1$, то меняем переменную $y_i^0 = \alpha$;
- Меняем все потенциалы вершин: $\theta_i(0) = \psi_i(y_i^0), \theta_i(1) = \psi_i(\alpha)$;
- Меняем потенциалы на ребрах: $\theta_{ij}(0,0) = \psi_{ij}(y_i^0, y_j^0), \theta_{ij}(1,1) = \psi_{ij}(\alpha, \alpha), \theta_{ij}(0,1) = \psi_{ij}(y_i^0, \alpha)$, $\theta_{ij}(1,0) = \psi_{ij}(\alpha, y_j^0)$;
- Повторяем процедуру, сколько нам надо для реальной задачи.

Лекция 9 от 14.10.2016. P vs. NP

Класс P и сведение

Сейчас мы переходим в теоретический материал, который является основой классификации алгоритмов. Нас совершенно не интересует эффективность, кроме как разделение на «алгоритмы, которые работают за полиномиальное время» и все остальные.

Договоримся, что мы определяем «алгоритм», как детерминированную машину Тьюринга. И будем использовать тезис Чёрча-Тьюринга о том, что любая вычислимая функция является вычислимой на машине Тьюринга. Также условимся, что мы рассматриваем только задачи принятия решения (принадлежит ли слово w языку L).

Определение 1. *Задача (формально говоря язык) $A \in \mathbf{P}$, если существует машина Тьюринга, проверяющая принадлежность слова языку, время работы $t_A(x)$ которой ограничено полиномом от $P(|x|)$, который является фиксированным для всех входов.*

Также, до определения класса **NP**, определим, что значит, что язык A сводится к языку B за полиномиальное время.

Определение 2. *Язык A **сводится** к языку B за полиномиальное время, если существует функция f, вычислимая за полиномиальное время, такая, что $w \in A \iff f(w) \in B$. Обозначение: $A \leqslant_p B$.*

Докажем основное утверждение о сводимости:

Утверждение 1. *Пусть $A \leqslant_p B$ и $B \in \mathbf{P}$ (существует полиномиальный алгоритм решения задачи). Тогда $A \in \mathbf{P}$.*

Доказательство. Достаточно построить алгоритм, показывающий, что для языка A тоже существует полиномиальное решение. Пусть у нас имеются МТ M_A, допускающая язык A, и M_B, допускающая язык B. Поскольку $A \leqslant_p B$, то $\exists$ полиномиальная функция f, удовлетворяющая определению выше. Построим алгоритм вычисления M_A:

Require: слово w
  вычислить $f(w)$
  return $M_B\big(f(w)\big)$

Поскольку сама функция f является вычислимой за полиномиальное время, то вычисление $f(w)$ потратит полиномиальное время. Кроме того, поскольку $B \in \mathbf{P}$, то вычисление $M_B\big(f(w)\big)$ также займёт полиномиальное время (от размера $f(w)$, но $f(w)$ также вычисляется за полином)! Отсюда $M_A(w)$ также допускает слово w за полиномальное время, что по определению означает, что $A \in \mathbf{P}$. □

Класс NP

Определение 3. *Язык L лежит в классе* **NP**, *если существует функция от двух аргументов $A(x, y)$ — **алгоритм верификации** — с полиномиальной сложностью от x такая, что x лежит в L тогда и только тогда, когда для него существует y (его принято называть **сертификатом**) такой, что $A(x, y) = 1$. При этом сертификат должен быть полиномиально зависим от размера x.*

Ясно, что $\mathbf{P} \subseteq \mathbf{NP}$, так как за алгоритм верификации можно взять просто полиномиальный алгоритм принадлежности слова языку.

Также введём понятие **coNP** класса:

Определение 4.

$$\mathbf{coNP} = \{\overline{L} = \Sigma^* \setminus L \mid L \in \mathbf{NP}\}$$

Другими словами, **coNP** — множество всех языков, которые могут верифицировать **не**принадлежность слово языку. Например, такая задача — «Нет ли гамильтонового цикла в графе?». Неизвестно, лежит ли она в **NP**, но она точно лежит в **coNP** (предъявление верификатора оставляем, как упражнение читателю). Также ясно, что $\mathbf{P} \subseteq \mathbf{coNP}$.

Определение 5. *Язык* $A \in \mathbf{PSPACE}$*, если существует МТ принимающая данный язык с полиномиальным количеством дополнительной памяти.*

Определение 6. *Язык* $A \in \mathbf{EXPTIME}$*, если существует МТ принимающая данный язык с экспоненциальным временем работы (а именно* $\mathcal{O}(2^{n^k})$*).*

Оставим тоже следующую лемму, как упражнение читателю:

Лемма 1. $\mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$

NPC класс и теорема Кука-Левина

Определение 7. *Язык* A*—* **NP***-трудный, если* $\forall\ B \in \mathbf{NP} : B \leqslant_p A$.

Определение 8. *Язык* $A \in \mathbf{NPC}$ *(***NP***-complete или Nondeterministic Polynomial Complete), если* $A \in \mathbf{NP}$ *и* $\forall\ B \in \mathbf{NP} \implies B \leqslant_p A$.

Другими словами, **NPC** — это в точности те языки, которые сами лежат в **NP** и являются **NP**-трудными одновременно.

Докажем следующие простые утверждения:

Лемма 2. *Пусть* $L \in \mathbf{NPC}$*. Тогда, если* $L \in \mathbf{P}$*, то* $\mathbf{P} = \mathbf{NP}$.

Доказательство. Пусть $L' \in \mathbf{NP}$. Так как $L \in \mathbf{NPC}$, то $L' \leqslant_p L$. Но, поскольку $L \in \mathbf{P}$, то $L' \in \mathbf{P}$, откуда $\mathbf{P} = \mathbf{NP}$. □

Лемма 3. *Если* $B \in \mathbf{NPC}$ *и* $B \leqslant_p C \in \mathbf{NP}$*, то* $C \in \mathbf{NPC}$.

Доказательство. Проверим оба условия, входящих в определение **NP**-полного языка.

1. $C \in \mathbf{NP}$ — по условию.
2. Пусть $L' \in \mathbf{NP}$. Так как $B \in \mathbf{NPC}$, то $L' \leqslant_p B$, что по определению означает, что существует некая функция $f_{L'B}$ (которая сводит одну задачу к другой), вычислимая за полиномиальное время. Кроме того, $B \leqslant_p C$, что по определению означает, что существует некая функция f_{BC}, вычислимая за полиномиальное время.

   Но это означает, что $L' \leqslant_p C$, так как существует функция $f_{L'C}$, такая, что $f_{L'C}(w) = f_{BC}\big(f_{L'B}(w)\big)$ – она также является вычислимой за полиномиальное время.

   Следовательно, $\forall\ L' \in \mathbf{NP} : L' \leqslant_p C$, откуда C – **NP**-трудный по определению.

Оба условия выполняются, значит $C \in \mathbf{NPC}$. □

Но всё равно должны же остаться вопросы по тому, зачем всё это надо и как вообще доказывать, что задача трудна, то есть лежит в классе **NP**. Ведь должна же быть какая-то «первая» задача из **NPC**. Если вы решаете задачу и вдруг понимаете, что она эквивалентна какой-то задаче из **NPC**, то возможно стоит либо перечитать задачу (и понять, каким условием надо точно пользоваться, если вдруг вы его отбросили), либо решить и получить \$1 000 000.

Одна из первых **NPC** задач была задача SAT (от англ. Satisfiability). Можем считать, что нам дана булева формула в конъюнктивно нормальной форме. Сейчас мы покажем, что любая **NP** задача сводится к этой. Вообще, так как если мы хотим сводить любую **NP** задачу к SAT, то какие-то обычные рассуждения вряд ли пройдут. Действительно, мы будем стараться моделировать работу МТ через КНФ.

Теорема 1 (Теорема Кука-Левина)**.** $\mathsf{SAT} \in \mathbf{NPC}$.

Доказательство. Сначала покажем, что $\mathsf{SAT} \in \mathbf{NP}$. Действительно, по входу булевой формулы и верификатору (значение переменных $x_1, \ldots, x_n$) мы с лёгкостью проверим выполнимость формулы. Поэтому есть линейный алгоритм верификации.

Пусть $\mathcal{P} \in \mathbf{NP}$. Надо понять, как полиномиально свести эту задачу к SAT. По определению класса **NP** существует полином p и алгоритм $\mathcal{P}'$, который верифицирует задачу $\mathcal{P}$, причём верификатор размера не больше, чем $p(|x|)$.

Пусть

$$\Phi : \{0, \ldots, N\} \times \overline{A} \to \{-1, \ldots, N\} \times \overline{A} \times \{-1, 0, 1\}$$

МТ, работающая за полиномиальное время, отвечающая за задачу $\mathcal{P}'$ с алфавитом A. Добавим пустой символ к A, чтобы легче было работать. Пусть работа этой МТ ограничена полиномом q, то есть $\mathrm{time}(\Phi, x\#c) \leqslant q(|x\#c|)$. Сейчас мы соорудим набор дизъюнктов или «клауз» (от англ. Clause) $Z(x)$, который ограничен полиномом $Q \leftarrow q(|x\#c|)$, такой, что $Z(x)$ выполняется тогда и только тогда, когда слово принадлежит языку $\mathcal{P}$.

Ясно, что мы не уйдём за пределы ленты от $-Q$ до Q.

Теперь создадим несколько переменных:

- переменные $v_{ij\sigma}$ для всех $0 \leqslant i \leqslant Q$, $-Q \leqslant j \leqslant Q$ и $\sigma \in \overline{A}$. Верно ли, что в момент i (после i шагов выполнения МТ) позиция с номером j содержала символ σ.
- переменные w_{ijn} для всех $0 \leqslant i \leqslant Q$, $-Q \leqslant j \leqslant Q$ и $-1 \leqslant n \leqslant N$. Верно ли, что в момент i на позиции j МТ была в состоянии n. За минус 1 отвечает финальное состояние.

Поэтому, если у нас есть конфигурация МТ на i-ом шаге с позицией $\pi^{(i)}$, символами $s^{(i)}$ и состоянием $n^{(i)}$, тогда мы должны выставить $v_{ij\sigma} := \mathit{true}$ тогда и только тогда, когда $s_j^{(i)} = \sigma$ (s_j — позиция на j-ом месте ленты МТ). И мы должны выставить $w_{ijn} := \mathit{true}$ тоже тогда и только тогда, когда $\pi^{(i)} = j$ и $n^{(i)} = n$.

Сейчас мы предъявим набор клауз $Z(x)$, который будет выполняться тогда и только тогда, когда существует строка c полиномиального размера, что МТ Φ на входе $x\#c$ выдаёт единицу.

В каждый момент времени на каждой позиции стоит строго один символ:

- дизъюнкт $(v_{ij\sigma} \mid \sigma \in \overline{A})$ по всем $0 \leqslant i \leqslant Q$ и $-Q \leqslant j \leqslant Q$.
- $(\overline{v_{ij\sigma}} \vee \overline{v_{ij\tau}})$ по всем $0 \leqslant i \leqslant Q$, и $-Q \leqslant j \leqslant Q$, и $\sigma \neq \tau \in \overline{A}$.

В каждый момент времени единственная позиция в строке сканируется и единственная инструкция выполняется:

- $(w_{ijn} \mid -Q \leqslant j \leqslant Q, -1 \leqslant n \leqslant N)$ для всех $0 \leqslant i \leqslant Q$.
- $(\overline{w_{ijn}} \vee \overline{w_{ij'n'}})$ по всем $0 \leqslant i \leqslant Q$ и $-Q \leqslant j, j' \leqslant Q$ и $-1 \leqslant n, n' \leqslant N$ с условием, что $(j, n) \neq (j', n')$.

Алгоритм корректно начинает свою работу:

- (v_{0,j,x_j}) по всем $1 \leqslant j \leqslant |x|$.
- $(v_{0,|x|+1,\#})$ — разделяющий символ.
- $(v_{0,|x|+1+j,0} \vee v_{0,|x|+1+j,1})$ по всем $1 \leqslant j \leqslant p(|x|)$.
- $(v_{0,j,\sqcup})$ по всем $-Q \leqslant j \leqslant 0$ и $|x| + 2 + p(|x|) \leqslant j \leqslant Q$ — пустые символы не с входа.
- $(w_{0,1,0})$ — начальное положение головки.

Алгоритм работает корректно:

- $(\overline{v_{ij\sigma}} \vee \overline{w_{ijn}} \vee v_{i+1,j,\tau})$ и $(\overline{v_{ij\sigma}} \vee \overline{w_{ijn}} \vee w_{i+1,j+\delta,m})$ по всем $0 \leqslant i < Q, -Q \leqslant j \leqslant Q; \sigma, \tau \in \overline{A}, \delta \in \{-1, 0, 1\}$, где $\Phi(n, \sigma) = (m, \tau, \delta)$ (переход по состоянию n, символу σ в какое-то состояние m, символ τ пишем на j-ом месте и сдвиг на $-1, 0$ или 1).

Когда алгоритм достигает финального состояния (минус 1 в нашем случае), алгоритм останавливается:

- $(\overline{w_{i,j,-1}} \vee w_{i+1,j,-1})$ и $(\overline{w_{i,j,-1}} \vee \overline{v_{i,j,\sigma}} \vee v_{i+1,j,\sigma})$ по всем $0 \leqslant i < Q; -Q \leqslant j \leqslant Q$ и $\sigma \in \overline{A}$ — как раз, если достигли состояния -1, тогда все состояния в каждое время должны быть минус один и символ меняться не должен.

Позиции, которые не были просмотрены, должны остаться неизменными:

- $(\overline{v_{ij\sigma}} \vee \overline{w_{ij'n}} \vee v_{i+1,j,\sigma})$ по всем $0 \leqslant i \leqslant Q; \sigma \in \overline{A}; -1 \leqslant n \leqslant N$ и $-Q \leqslant j, j' \leqslant Q$ при условии, что $j \neq j'$.

Алгоритм выводит единицу:

- $(v_{Q,1,1})$ и $(v_{Q,2,\sqcup})$ — на первой позиции стоит единица, а потом пробел (можно считать, что только один, можно, что все последующие, это не играет на роль полиномиальности МТ и полиномиальности размера КНФ).

Заметим, что размер $Z(x)$ будет не больше, чем $\mathcal{O}(Q^3 \log Q)$, существует $\mathcal{O}(Q^3)$ литералов и нужно $\mathcal{O}(\log Q)$ памяти, чтобы закодировать индексы.

Осталось показать, что $Z(x)$ выполняется тогда только тогда, когда x принадлежит языку.

Если $Z(x)$ выполняется, то пусть T — это набор переменных, удовлетворяющим всем клаузам. Поставим $c_j = 1$ по всем j для которых $T(v_{o,|x|+1+j,1}) = true$ и $c_j = 0$ в ином случае. Сверху мы

описали работу МТ Ф на входе $x\#c$. Поэтому мы можем заключить, что $\Phi(x\#c) = 1$. Так как Ф — алгоритм верификации, то это значит, что x принадлежит языку.

Пусть x принадлежит языку. Тогда пусть c — сертификат проверки для x. Тогда пусть конфигурация МТ на входе $x\#c$ на i-ом шаге пусть будет равна $(n^{(i)}, s^{(i)}, \pi^{(i)})$. Тогда давайте поставим $T(v_{i,j,\sigma}) = true$ тогда и только тогда, когда $s_j^{(i)} = \sigma$ и $T(w_{i,j,n}) = true$ тогда и только тогда, когда $\pi^{(i)} = j, n^{(i)} = n$ по всем $i \leqslant m$. Для бóльших i выставим $T(v_{i,j,\sigma}) := T(v_{i-1,j,\sigma})$ и $T(w_{i,j,n}) := T(w_{i-1,j,n})$ по всевозможным j, n, σ. Тогда можно убедиться, что формула выполняется, что завершает наше доказательство. □

Лекция 10 от 18.10.2016. NP классы, сведения, различные другие классы алгоритмов

> «Будет вообще уморительно, если кто-то сядет и скажет: «Ох ё, поиск Гамильтонова цикла — это просто динамика за квадрат». И полстраницы кода...»
>
> Глеб

В прошлый раз мы поговорили о просто задаче SAT. Теперь у нас есть мощный инструмент к сведению сложных на данное время задач. Первая из них — 3-SAT — булева формула в конъюктивно нормальной форме, где каждый дизъюнкт содержит не более 3 литералов (ну или ровно 3, мы докажем эквивалентность чуть позже).

Сведение SAT к 3-SAT

Теорема 1. $\mathsf{SAT} \leqslant_p \text{3-}\mathsf{SAT}$.

Доказательство. Возьмём один дизъюнкт и сделаем из него много дизъюнктов с количеством литералов не более 3.

Действительно, пусть у нас будет дизъюнкт $(x_1 \vee \ldots \vee x_k)$ — возможно с отрицаниями, нам не важно. Разобьём на 2 примерно равных множества этот дизъюнкт — в одном $\left\lfloor \frac{k}{2} \right\rfloor$ литералов, в другом $\left\lceil \frac{k}{2} \right\rceil$. Добавим новую переменную x_{n+1}, тогда покажем эквивалентность $(x_1 \vee \ldots \vee x_{\lfloor k/2 \rfloor} \vee x_{n+1}) \wedge (x_{\lfloor k/2 \rfloor + 1} \vee \ldots \vee x_k \vee \overline{x_{n+1}})$. Действительно, если эти 2 дизъюнкта выполнены, то в одном из них литерал с x_{n+1} равен 0, значит один из литералов в множестве $x_1, \ldots, x_k$ равен 1 и изначальный дизъюнкт выполнен.

В другую сторону — если изначальный дизъюнкт выполнен, тогда существует какой-то литерал из $x_1, \ldots, x_k$, который равен 1, тогда поставим x_{n+1} так, что оно равно 0, там где литерал из $x_1, \ldots, x_k$ равен 1. Тогда обе скобки будут равны единице (там, где есть литерал, который равен 1, будет всегда 1, а в другой скобке литерал с x_{n+1} равен 1).

Будем так делать для каждого дизъюнкта, добавляя новую (обязательно не совпадающую с предыдущими созданными) переменными. И будем останавливаться, когда дизъюнкт содержит не более 3 литералов. Заметим, что из дизъюнкта, состоящего из 3 литералов нельзя сделать 2 дизъюнкта со строго меньшим количеством литералов. Легко проверить, что такая процедура работает только при $k \geqslant 4$.

Осталось совсем немного — доказать, что $\text{3-}\mathsf{SAT} \in \mathbf{NP}$ и сведение действительно полиномиально. Первое совсем очевидно, так как 3-SAT это частный случай SAT, а про SAT мы точно знаем, что эта задача из класса **NP**.

Заметим, что количество уровней при процедуре с одним дизъюнктом будет не более, чем $\mathcal{O}(\log n)$ (это легко показать, сказав и проверив, что если проделать 2 уровня, то максимальная длина дизъюнкта уменьшается хотя бы в 2 раза). И на каждом уровне мы создаём не более 2^k новых литералов. Если всё просуммировать, получим линейное сведение одного дизъюнкта. Для осталальных проделаем то же самое. □

Также стоит отметить по лемме в прошлой лекции следует, что 3-SAT $\in$ **NPC**.

Пока никто не умеет сводить SAT к 2-SAT, так как последняя решается за линейное время.

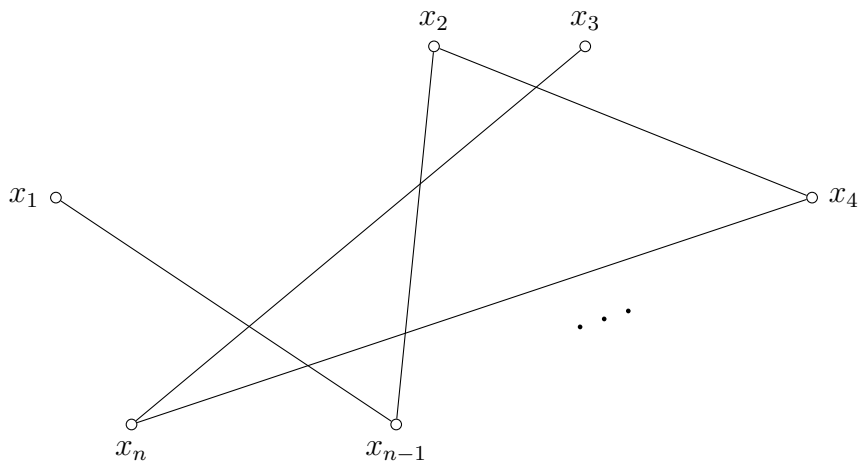
Байка от Глебаса: Только испанские составители контестов могут это делать. Писали испанский контест. Читаешь условие — вот просто дана задача о рюкзаке. $n \leqslant 50000, a_i \leqslant 10^9$, TL 2 секунды. Зашли в админку, увидели супер-эвристику, долго ржали, возможно, даже отослали решение, но не помню. Ну также они дали задачу на гамильтонов путь на 200 вершинах.

NP-полнота задач клики, доминирующего множества и вершинного покрытия

Ну теперь мы займёмся сведениями. Мы много говорили, что некоторые задачи на графах очень сложны. Теперь надо ответить за базар:

Теорема 2. *Задача «существует ли клика размера k в графе $G(V, E)$» является* **NPC**.

Доказательство. Сведем эту задачу к SAT (просто красивое рассуждение). Потом сделаем в другую сторону.



Переменные будут отвечать за вершины. Соорудим нашу формулу:

Любые 2 вершины, между которыми нет ребра, не могут быть взяты обе.

- $(\overline{x_v} \vee \overline{x_u})$ при всех $(v, u) \notin E$.

Введём такую величину — d_{ij}, отвечающую на вопрос, верно ли, что среди первых i вершин есть клика размера j. Заметим, что $d_{ij} = d_{i-1,j} \vee (d_{i-1,j-1} \wedge x_i)$, то есть либо есть клика размера k среди первых $i-1$ вершины, либо среди первых $i-1$ есть клика размера $j-1$ и взята вершина i.

- $(\overline{d_{ij}} \vee d_{i-1,j} \vee d_{i-1,j-1}) \wedge (\overline{d_{ij}} \vee d_{i-1,j} \vee x_i)$ — если $d_{i-1,j} = 0$ и $x_i = 0$, тогда точно $d_{ij} = 0$ и выполняются оба дизъюнкта. Если $d_{i-1,j} = 0$ и $d_{i-1,j-1} = 0$, то $d_{ij} = 0$ в обоих случаях. Это делаем при $1 \leqslant i \leqslant n; 1 \leqslant j \leqslant k$.
- $(d_{ij} \vee \overline{d_{i-1,j}}) \wedge (d_{ij} \vee \overline{d_{i-1,j-1}} \vee \overline{x_i})$. Это те же условия, только мы здесь хотим сделать $d_{ij} = 1$, если выполнено хотя бы одно условие. Это делаем при $1 \leqslant i \leqslant n; 1 \leqslant j \leqslant k$.

Начальные условия (среди первых):

- $d_{0j} = 0$ при $1 \leqslant j \leqslant k$ — среди нуля вершин нет клики размера хотя бы 1.
- $d_{i0} = 1$ при $0 \leqslant i \leqslant n$ — среди первых i вершин есть клика размера 0.

Финальное состояние:

- (d_{nk}) — ответ на задачу.

Легко видеть, что это и есть SAT, притом формула выполняется тогда и только тогда, когда есть клика размера k. Причём сведение, очевидно, полиномиально.

Теперь в другую сторону:

Рассмотрим любую SAT формулу. Пусть у нас есть k дизъюнктов. Выпишем всех их (каждый литерал — отдельная вершина) в виде графа. Внутри дизъюнкта вершины не будем соединять, а также не будем соединять вершины в различных дизъюнктах, которые отвечают сразу за x_i и $\overline{x_i}$. Теперь запустим решение задачи о поиске клики размера k. Если решение нашлось, то в каждой части графа, отвечающего за отдельный «дизъюнкт», выбрана ровно 1 вершина. Иначе в каком-то дизъюнкте выбрано 2, а мы не соединяли вершины в одном и том же дизъюнкте. Поэтому в каждом дизъюнкте выбран ровно 1 литерал. Сделаем эти литералы равными единице. Те переменные, которые мы не выбрали, положим единице. Заметим, что мы не сделали одновременно x_i и $\overline{x_i}$ равными единице, так как иначе между ними было бы ребро. А мы договорились, что такого ребра нет.

То, что любому решению SAT формулы соответствует какая-то клика в построенном графе следует из почти дословных рассуждений выше, что завершает док-во, что задача про поиск клики данного размера лежит в **NPC**. □

Теперь поговорим про задачу о доминирующем множестве размера k. Напомним, что мы хотим в данной задаче найти k вершин так, что оставшиеся вершины соединены хотя бы с одной из выбранных вершин. Докажем следующее сведение:

Теперь поговорим о вершинном покрытии графа размера k. Вспомним, что вершинное покрытие это множество вершин такое, что любое ребро инцидентное хотя бы одной вершине из множества. Неудивительно, эта задача тоже **NPC**.

Теорема 3. Vertex_cover $\in$ **NPC**.

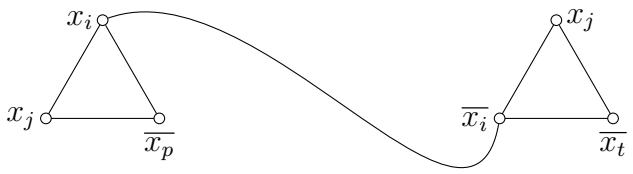
Доказательство. Сведём 3-SAT к этой задаче.

Ну здесь как раз нам и понадобится факт, что в любой КНФ, где дизъюнкты имеют размер не более 3, можно сделать ровно 3.

Добавим 3 переменных x, y, z. Мы хотим их сделать всегда $false$. Давайте напишем 7 дизъюнктов длины 3 с этими переменными, кроме одного — $(x \vee y \vee z)$. Если хотя бы одна переменная равна 1, то несложно убедиться, что формула будет равна 0 (просто перебор). Значит все равны 0. То есть решения существуют тогда и только тогда, когда $x = y = z = 0$. Поэтому их не жалко добавлять в те дизъюнкты, где не хватает литералов до количества 3.

Теперь каждый дизъюнкт содержит ровно 3 переменных.

Теперь построим такой граф: каждый дизъюнкт отвечает треугольнику, причем в различных треугольниках мы соединяем вершины, соответствующие x_i и $\overline{x_i}$. Проиллюстрируем это рисунком для любых 2 различных треугольников:



Теперь, если есть решение формулы, давайте докажем, что у нас есть решение задачи о вершинном покрытии размера $2k$, где k — количество дизъюнктов.

По решению выберем в каждом треугольнике, где значение литерала равно 1. И отметим 2 другие вершины в качестве покрытия. Докажем, что это действительно вершинное покрытие. Ясно, что в каждом треугольнике все рёбра будут инцидентные какой-то вершине, так как выбраны ровно 2 вершины. Осталось разобраться с ребрами между x_i и $\overline{x_i}$. Если ни одна вершина не покрывает это ребро, тогда и x_i, и $\overline{x_i}$ были равны 1, что невозможно. Значит мы нашли вершинное покрытие размера $2k$.

Обратно. Пусть у нас есть покрытие размера $2k$, тогда в каждом треугольнике выбрано не меньше 2 вершин, иначе какое-то ребро не будет покрыто вершиной. С другой стороны их не больше 2, так как иначе по принципу Дирихле найдётся треугольник, в котором покрыто меньше 2 вершин.

Теперь возьмём во всех треугольниках и сделаем литерал равным единице, который не лежит в этом покрытии. Остальным переменным, которые мы не использовали, выставим 1 (с ними всё корректно, мы их не использовали). Осталось понять, что мы корректно выставили всем переменным значения. Если мы вдруг захотели выставить $x_i = 1$ и $\overline{x_i} = 1$, то эти обе вершины не лежали в вершинном покрытии, а между ними есть ребро, поэтому вершинное покрытие было некорректным.

Осталось проверить, что эта задача из **NP**. Действительно, легко по множеству вершин определить, является ли это множество вершинным покрытием (надо просто просмотреть все рёбра). Значит эта задача является **NPC**. □

Другие классы алгоритмов

Все мы знаем, что проблема останова невычислима. Все классы алгоритмов, которые считают, что проблема останова невычислима обозначают за $\mathbf{H_0}$. Вот начинают рассматривать некоторые классы алгоритмов, при условии, что мы умеем решать проблему остановки. Этот класс обозначают за $\mathbf{H_1}$, но это ещё не всё! Проблема остановки с оракулом проблемы остановки на обычной МТ тоже невычислима. И если уметь решать и эту проблему, то такие классы алгоритмов обозначают за $\mathbf{H_2}$ и так далее. Нужно ли это кому-то? Вряд ли. Какая разница, если мы уже не умеем решать проблему остановки, то зачем рассматривать случаи, когда мы умеем это делать? -Непонятно, но знать об этом стоит.

Определение 1. *Класс* $\mathbf{L}$ *— это те языки, для которых существует МТ, которая работают с* $\mathcal{O}(\log n)$ *дополнительной памятью. Легко показать, что* $\mathbf{L} \subseteq \mathbf{P}$ *(оставим, как упражнение читателю).*

Пример задачи — узнать длину строки на входе. Нам нужно все $\mathcal{O}(\log n)$ бит, чтобы закодировать длину на входе длины n.

Определение 2. *Класс* **BPP** *(от англ. Bounded-Error Probabilistic Polynomial) те языки, для которых существует недетерминированная МТ (использующая генератор случайных чисел, МТ выбирает переход по таблице переходов с некоторой равной вероятностью), которые ошибаются с вероятностью не более* $\frac{1}{3}$.

Почему $\frac{1}{3}$? По схеме Бернулли, если $p < 1/2$, то мы можем быть сильно уверены в правильности ответа после многократного запуска (например, $\mathcal{O}(n)$ вероятность будет сравнима с экспонентной от входа). Про $\frac{1}{3}$ просто договорились.

Определение 3. *Класс* **RP** *(от англ. Randomized Polynomial) это те языки, для которых, если слово не принадлежит языку, то вероятность, что МТ допустит это слово равна 0. Если принадлежит, то вероятность не меньше* $\frac{1}{2}$, *что МТ допустит (мы опять рассматриваем недетерминированные МТ с генератором случайных чисел).*

Примеры таких алгоритмов — хэши.

Класс **coRP** определяется также, только поменяны местами выражения «принимает» и «не принимает».

Определение 4. *Класс* **ZPP** *(от англ. Zero-Error Probabilistic Polynomial) это языки, для которых существует вероятностная МТ, которая всегда отвечает правильно и математическое ожидание времени работы полиномиально.*

Упражнение читателю: $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$.

Зачем мы приводим здесь все эти классы? При приёме в аспирантуру всегда есть вопрос про классы алгоритмов, поэтому просто полезно об этом знать.

На этом наш курс подошёл к концу.

Благодарности

Спасибо большое за вычитку конспектов следующим людям: Павлу Корозевцеву, Алексею Данилюку, Тамаре Гурциевой, Александру Андрееву за нахождение огромного количества опечаток и неточностей, Алексею Калинову за вычитку лекции про RSA, Михаилу Дискину за вычитку лекции про **P** и **NP**.