Эффективные алгоритмы — I (ЧЕРНОВИКИ КОНСПЕКТОВ — НЕ РАСПОСТРАНЯТЬ!)

Э. А. Гирш

10 декабря 2003 г.

Предисловие

Настоящий файл отражает лекции спецкурса «Эффективные алгоритмы. Часть I», читавшегося на математико-механическом факультете Санкт-Петербургского государственного университета в 1999, 2001 и 2003 годах. Материал соответствует преимущественно лекциям 2001 года (с более поздними изменениями и дополнениями, в частности, из других курсов). Автор выражает благодарность студентам, составившим первоначальные конспекты лекций: Ф. Александрову, А. Бережному, О. Ионовой, А. Кожевникову, А. Куликову, Н. Петровой, М. Плискину и У. Тюляковой, а также Д. Ицыксону, указавшему на многочисленные неточности в конспекте.

Оглавление

Глава 1

Алгоритмы, обрабатывающие вход по мере поступления

При написании программ, работающих в условиях изменяющегося окружения (например, системных программ, контролирующих работу операционной системы), зачастую оказывается, что входные данные поступают постепенно, а не сразу все. При этом программа должна производить действия (принимать решения или даже записывать что-то на выход) до того, как поступят все данные; такие алгоритмы называются работающими в реальном времени, или *online*-алгоритмами. Ниже мы рассмотрим несколько примеров таких задач, разберемся, как следует оценивать качество таких программ, и проанализируем некоторые простые алгоритмы.

1.1 Задача кэширования (paging problem)

Следующая задача должна быть решена тем или иным образом практически в любой операционной системе.

У компьютера есть два вида памяти: дисковая, большая по объему, но медленная, и оперативная, быстрая, но меньше по объему. Все операции над данными (в частности, запуск программ) производятся в оперативной памяти; дисковая память используется лишь для их хранения.

Таким образом, периодически данные считываются с диска в оперативную память, на что уходит существенное время. Если обращения к одному и тому же месту дисковой памяти повторяются, можно сэкономить время на том, что выделить в оперативной памяти определенное место (*кэш*), в которой сохранять считанные данные некоторое время после того, как они были обработаны. Однако, поскольку объем кэша ограничен, периодически все же придется стирать из него данные (записывать на их место другие); качество алгоритма определяется тем, насколько он сможет предугадать, какие данные еще понадобятся (и их не следует стирать), а какие — нет.

Поскольку алгоритм должен работать в реальном времени, он не может знать, какие данные понадобятся в будущем. Поэтому «абсолютно оптимального» алгоритма нам построить не удастся; наша задача будет состоять в том, чтобы построить алгоритм, который работал бы не слишком хуже любого другого (который могут придумать наши конкуренты).

Формальная постановка задачи. Вся память (и дисковая, и оперативная) разбита на блоки равной длины. Блоки пронумерованы, причем кэш состоит из k блоков, а диск — из t блоков. На очередном шаге к нам поступает запрос на какой-то блок с диска; если этот блок в кэше уже имеется, нам достаточно сообщить его номер в кэше; если нет, нам надо предварительно решить, в какое место кэша записать этот блок и сделать это, считав блок с диска. Время работы алгоритма измеряется количеством считываний с диска. Будем считать, что в начальный момент времени кэш пуст.

Оценка качества алгоритма. Мы договорились, что является временем работы нашего алгоритма. Однако, для разных данных время работы будет различно (сравните последовательности запросов «$1, 1, \ldots, 1$» и «$1, 2, \ldots, t$»). Поэтому нас больше будет интересовать время работы нашего алгоритма по сравнению с некоторым другим алгоритмом (возможно, написанным нашим конкурентом).

Путь A — алгоритм, а $r = (r_1, r_2, \ldots, r_n)$ — последовательность запросов ($1 \leq r_i \leq t$). Стоимость обработки последовательности запросов алгоритмом — $cost_A(r)$ — это время его работы (т.е. количество обращений к диску). Пусть opt — некий воображаемый алгоритм, который мы считаем «самым лучшим». Алгоритм A называется *c-оптимальным* (c-competitive), если для любого r и для любого opt выполняется неравенство

$$cost_A(r) \leq c \cdot cost_{opt}(r) + c_1.$$

Подразумевается, что c не зависит от r (и даже его длины), но она может зависеть от k или других параметров.

Наш алгоритм может быть и вероятностным. Тогда определение c-оптимальности превращается в

$$\mathbf{E} cost_A(r) \leq c \cdot \mathbf{E} cost_{opt}(r) + c_1.$$

Конечно, определение c-оптимальности зависит от того, из какого класса алгоритмов берется алгоритм opt. Мы рассмотрим два варианта.

«Забывчивый противник» (oblivious adversary). Рассмотрим случай, когда A — вероятностный online-алгоритм, а opt — детерминированный *offline*-алгоритм («oblivious adversary»), т.е. он может быть даже лучше, чем самый лучший алгоритм наших конкурентов: он *заранее* всю знает последовательность запросов r. Важное ограничение: opt должен работать, не зная наших действий (конечно, он может знать наш алгоритм, но он не знает случайных чисел, которые мы использовали).

Алгоритм 1.1 (MARKER). Будем помечать каждый блок кэша некоторым битом (0 или 1). Все время работы разделяется на «периоды». В начале каждого периода все элементы кэша помечены 0.

Итак, приходит очередной запрос. Если это запрос на блок, который уже есть в кэше, то наш алгоритм помечает его битом 1. Если запрашивают элемент, которого нет в кэше, то мы случайным образом выбираем блок из помеченных битом 0 и считываем туда требуемый блок с диска. Затем мы помечаем этот блок битом 1.

Рано или поздно у нас не останется блоков, помеченных битом 0. В этом состоянии мы можем работать, пока приходят запросы на блоки, находящиеся в кэше. Как только поступит запрос на блок, которого нет в кэше, мы обнулим все пометки и начнем новый период. □

Теорема 1.1. *Алгоритм MARKER является $2H_k$-оптимальным против offline-алгоритмов.*

Доказательство. Разделим все поступающие запросы на три вида: «помеченные», «устаревшие» и «чистые». Помеченный запрос — это запрос на блок, образ которого находится в кэше и помечен 1. Устаревший — это запрос на блок, образ которого находится в кэше и помечен 0 (значит, он остался в кэше с предыдущего периода), или на блок, которого в кэше нет, но который там был в предыдущем периоде. И, соответственно, чистый запрос — это запрос на блок, которого в кэше нет и не было в предыдущем периоде.

Обозначим через l_i количество чистых запросов за i-й период; пусть $d_{I,i}$ — количество различающихся элементов в кэшах A и *opt* (т.е. половина мощности симметрической разности этих двух мультимножеств) в начале i-того периода, а $d_{F,i}$ — в конце.

Оценим снизу количество считываний блоков, которое сделает *opt*. С одной стороны, оно составляет не менее $l_i - d_{I,i}$, так как l_i блоков алгоритм A обязан подгрузить, и *opt* может выиграть у A только за счет того, что в начале периода у него в кэше уже будут блоки, на которые поступят запросы (таких блоков — не более $d_{I,i}$).

С другой стороны, это количество составляет не менее $d_{F,i}$. Действительно, все блоки, лежащие в кэше алгоритма A к концу периода, были запрошены (по построению алгоритма A). Следовательно, они должны были побывать и в кэше алгоритма *opt*; если же какого-то из них там не осталось, значит, на его место был загружен другой блок. Количество таких загрузок, очевидно, составляет не менее $d_{F,i}$.

Таким образом, количество считываний, которое должен проделать алгоритм *opt* за i-й период, составляет не менее

$$\max(l_i - d_{I,i}, d_{F,i}) \geq \frac{l_i - d_{I,i} + d_{F,i}}{2}.$$

Просуммировав по все периодам, получим, что число загрузок, которые должен сделать *opt* за все периоды не менее

$$\sum_i \frac{l_i - d_{I,i} + d_{F,i}}{2} = \sum_i \frac{l_i}{2} + \frac{d_{F,n}}{2} \geq \frac{L}{2},$$

где $L = \sum_i l_i$.

Теперь оценим сверху, сколько загрузок сделает алгоритм A. Ясно, что на чистые запросы ему придется подгружать блоки, на помеченные — нет. Сложнее дело обстоит с устаревшими запросами. Во-первых, сколько таких запросов может быть? Поскольку в результате устаревших и чистых запросов количество помеченных битом 1 блоков увеличивается, то в i-м периоде будет ровно $k - l_i$ устаревших запросов. При устаревшем запросе мы обращаемся к блоку, который на предыдущем периоде был в кэше, но есть ли он там сейчас, мы сказать не можем. Он мог быть выгружен, когда пришел чистый запрос либо устаревший запрос на блок, который до этого был уже выгружен.

Вероятность выгрузить нужный нам блок при j-том устаревшем запросе не превосходит $l/(k-j+1)$.

Таким образом, математическое ожидание количества загрузок устаревших блоков меньше или равно

$$l/k + \ldots + l/(k-(k-l)+1) = l \cdot (1/k + \ldots + 1/(l+1)) = l \cdot (H_k - H_l),$$

где $H_n = 1 + 1/2 + \ldots + 1/n$. Вспомнив, что мы должны еще загрузить l блоков при чистых запросах, мы получим, что

$$\mathbf{E}(\text{число загрузок}) \leq l \cdot (1 + H_k - H_l) \leq l \cdot H_k.$$

(В последнем неравенстве использовано, что если $l \neq 0$, то $H_l \geq 1$.) Это за один период. Просуммируем по всем периодам. Получим: $\mathbf{E}cost_A \leq H_k \cdot L$. Но $cost_{opt} = L/2$, и теорема доказана. □

Активный противник (adaptive adversary).

Определение 1.1. Рассмотрим ситуацию, когда adversary не только знает все запросы наперед, но и может их придумывать в зависимости от наших действий.

Adaptive offline adversary так и поступает; придумав все запросы, он порождает свой вариант их обработки стоимостью $cost_{opt}$. Заметим, что хотя он порождает новые запросы, зная наши ответы на предыдущие, он не может предугадать наших *будущих* действий, ибо они зависят от выпавших нам случайных чисел. Естественно, в определении *c-оптимальности относительно adaptive offline adversary* фигурируют не все r, а лишь r, порождаемые adversary.

Adaptive online adversary отличается от adaptive offline adversary тем, что он обязан обрабатывать (“порождать свой вариант обработки”) придумываемые им запросы online.

Задача 1.1. Оценить силу алгоритма MARKER относительно adaptive online и adaptive offline или показать, что этого сделать нельзя.

Задача 1.2. Придумать алгоритм лучше.

1.2 Задача о k официантах (k-server problem)

Обобщим задачу кэширования. Пусть у нас есть некоторое метрическое пространство с метрикой d, точки которого мы будем рассматривать как столики, и k официантов, то есть некоторое k-элементное подмножество точек этого пространства. Время от времени со столиков поступают заказы. Формально, заказ — это координаты столика. Заказы нужно обслуживать, то есть надо выбрать одного из официантов и переместить его в данную точку пространства. Стоимостью такого перемещения будет расстояние между исходной и конечной его точками. Каждый официант в каждый момент времени находится у какого-то столика, и состояние (конфигурация) системы описывается множеством координат столиков, у которых в данный момент находятся официанты. То есть состояние X — это мультимножество (множество с кратными элементами), состоящее из k точек пространства. Раз и навсегда зафиксируем начальное состояние A_0 нашей системы.

Пусть $r = r_1 r_2 \dots r_n$ — последовательность заказов. Тогда *рабочей функцией* W_r называется функция, сопоставляющая состоянию X наименьшую стоимость обслужить r из начального состояния A_0, придя в состояние X. (Стоимость обслуживания — это сумма пройденного всеми официантами расстояния, причем наименьший путь определяется с помощью оптимального offline алгоритма, т.е. знающего r.)

Алгоритм 1.2 (WORK FUNCTION ALGORITHM). Предположим, в данный момент наша система находится в состоянии X и к нам приходит запрос r. Тогда мы выбираем новое состояние X', которое содержит r и минимизирует следующую функцию от X': $W_{\rho r}(X') + D(X, X')$, где ρ — это последовательность запросов, которые мы уже обслужили, а $D(X, X')$ — это стоимость (оптимального) перехода из состояния X в X'. (Почему этот минимум достигается, будет ясно из леммы 1.0.) □

Лемма 1.0. $\exists a \in X : X' = X - a + r$.

Доказательство. Вытекает из неравенства треугольника. □

($X - a$ обозначает здесь и в дальнейшем мультимножество $X \setminus \{a\}$, а $Y + b$ — мультимножество $Y \cup \{b\}$.)

Замечание 1.1. В лемме 1.0 не исключается $a = r$.

Введем обозначения: $W := W_\rho$ — "старая" рабочая функция и $W' := W_{\rho r}$ — "новая" рабочая функция. Заметим, что

$$W'(A) = \min_{a \in A}\{W'(A-a+r)+d(r,a)\} = \min_{a \in A}\{W(A-a+r)+d(r,a)\} \quad (\geq W(A))$$

для $\forall$ состояния A. (Первое равенство ясно из определения рабочей функции, второе — очевидно.)

Докажем, что любая рабочая функция обладает свойством квазивыпуклости:

Определение 1.2. W назвается *квазивыпуклой*, если для $\forall$ состояний A и B существует биекция $h : A \to B$ такая, что для любого разбиения A на мультимножества X и Y, справедливо

$$W(A) + W(B) \geq W(X + h(Y)) + W(h(X) + Y). \tag{1.1}$$

Замечание 1.2. Смысл заключается в том, что A можно переделать в B (параллельно переделывая B в A) "маленькими шажками", не увеличивая суммарную работу.

Сначала докажем следующую лемму.

Лемма 1.1. *В определении 1.2 можно считать, что* $\forall x \in A \cap B$ $h(x) = x$.

Доказательство. Предположим, что h удовлетворяет условиям из определения квазивыпуклости, но для нее не выполняется утверждение леммы. Построим другую биекцию, которая будет удовлетворять обоим условиям. Построение будем проводить по индукции, постепенно удаляя из $A \cap B$ “плохие элементы”.

Итак, пусть $\exists a \in A \cap B : h(a) \neq a$ ($\Rightarrow h^{-1}(a) \neq a$). Рассмотрим новую биекцию h': $h'(a) = a$, $h'(h^{-1}(a)) = h(a)$, и h' совпадает с h на всех остальных точках.

Возьмем произвольное разбиение множества A на X и Y и докажем, что для h' верно неравенство из определения квазивыпуклости. Пусть $h^{-1}(a) \in X$ (если $h^{-1}(a) \in Y$, то доказательство проводится аналогично).

Если $a \in X$, то неравенство верно и для h', поскольку оно верно для h (ведь в этом случае $h(X) = h'(X)$ и $h(Y) = h'(Y)$).

Пусть теперь $a \in Y$. Рассмотрим множества $X' = X + a$, $Y' = Y - a$. Тогда

$$
\begin{aligned}
&W(X + h'(Y)) + W(h'(X) + Y) = \\
&W(X' + h'(Y')) + W(h'(X') + Y') = \\
&W(X' + h(Y')) + W(h(X') + Y') \geq \\
&W(A) + W(B),
\end{aligned}
$$

так как h квазивыпукла, а в определении квазивыпуклости требуется, чтобы неравенство (1.1) выполнялось для любого разбиения, в частности, для X' и Y'. То есть неравенство (1.1) выполнено и для h', а количество элементов a таких, что $h'(a) \neq a$, по крайней мере на 1 меньше, чем для h.

Так как A и B конечны, то за конечное число таких шагов мы исправим h так, чтобы она удовлетворяла утверждению леммы. □

Лемма 1.2. *Все рабочие функции квазивыпуклы.*

Доказательство. Указание: доказывать по индукции и использовать лемму 1.1.

ПРОБЕЛ В КОНСПЕКТЕ.

□

Определение 1.3. Состояние A назовем *минимизатором* точки a относительно W, если на нем достигается минимум $W(A) - \sum_{x \in A} d(x, a)$.

Лемма 1.3. *Если A — минимизатор для r относительно W_ρ, то A — минимизатор для r относительно $W_{\rho r}$.*

ПРОБЕЛ В КОНСПЕКТЕ.

Доказательство. □

Оценим, насколько хорош этот алгоритм. Вспомним, что

$$W'(X) = \min_{x \in X} \{W'(X - x + r) + d(x, r)\}.$$

Значит,

$$\exists x_0 \in X : W'(X) = W'(X - x_0 + r) + d(x_0, r) = W'(X') + d(x_0, r).$$

Стоимость одного шага нашего алгоритма при переходе от X к X' равна $d(x_0, r)$, а стоимость одного шага оптимального (offline) алгоритма — не менее $W'(X') - W(X)$ "в среднем" (т.е., если сложить $W'(X') - W(X)$ для всех шагов, то почти все члены сократятся и останется как раз суммарная стоимость всех шагов). Рассмотрим сумму этих величин $W'(X') - W(X) + d(x_0, r) = W'(X) - W(X)$.

Определение 1.4. Введем характеристику алгоритма — *обобщенную стоимость*, равную $\max\limits_X \{W'(X) - W(X)\}$.

Определение 1.5. *Полная обобщенная стоимость* — это сумма обобщенных стоимостей для всех шагов (запросов).

Замечание 1.3. Очевидно, полная обобщенная стоимость является оценкой сверху на сумму стоимостей нашего и оптимального алгоритмов.

Лемма 1.4. *Пусть полная обобщенная стоимость $\le (c+1)\cdot$ полная стоимость оптимального алгоритма$+c'$. Тогда WORK FUNCTION ALGORITHM является c-оптимальным.*

Доказательство. Очевидно. □

Лемма 1.5. *Пусть очередной запрос — r. Тогда обобщенная стоимость на этом шаге достигается на минимизаторе для r.*

Задача 1.3. Доказать лемму 1.5.

Теорема 1.2. *WORK FUNCTION ALGORITHM является $(2k-1)$-оптимальным.*

Доказательство. Введем

$$\Psi_W(U, B_1, \ldots, B_k) := k \cdot W(U) + \sum_{i=1}^{k} \left(W(B_i) - \sum_{b \in B_i} d(u_i, b) \right),$$

где $U = \{u_1, \ldots, u_k\}$ — состояние, $B_1, \ldots, B_k$ — тоже состояния. Введем также *потенциал*

$$\Phi(W) := \min_{U, B_1, \ldots, B_k} \Psi_W(U, B_1, \ldots, B_k).$$

Оценим $\Phi(W') - \Phi(W)$ (покажем, что эта величина $\ge$ обобщенной стоимости).

Можно считать, что $\exists j$**:** $r = u_j$**.** Предположим, что это не так, но $\exists i$: $W'(U) = W'(U - u_i + r) + d(r, u_i)$. Заменим u_i на r. Тогда $\Psi_{W'}(U, \ldots)$ уменьшится на

$$d(u_i, r) \cdot k + \sum_{b \in B_i} d(u_i, b) - \sum_{b \in B_i} d(r, b).$$

Это выражение неотрицательно по неравенству треугольника $\Rightarrow \Psi_{W'}$ разве что уменьшится, т.е. можно считать, что минимум $\Psi_{W'}$ достигается именно на таком U (содержащем r).

Также можно считать, что B_j — минимизатор для r.

Остальные элементы, минимизирующие $\Psi_{W'}(\ldots)$, зафиксируем, и именно их будем подразумевать, обозначая многоточием (и в самом $\Psi_{W'}$, и в Ψ_W!).

Тогда

$$\Phi(W') - \Phi(W) \geq \Psi_{W'}(\ldots) - \Psi_W(\ldots),$$

так как $\Phi(W') = \Psi_{W'}(\ldots)$, а $\Phi(W) \leq \Psi_W(\ldots)$: поскольку

$$\Phi(W) = \min_{V, C_1, \ldots, C_k} \Psi_W(V, C_1, \ldots, C_k) \leq \Psi_W(\ldots).$$

Далее, вспомним, что

$$\begin{aligned} \Psi_{W'} &= k \cdot W'(U) + \sum (W'(B_i) - \sum d(u_i, b)), \\ \Psi_W &= k \cdot W(U) + \sum (W(B_i) - \sum d(u_i, b)) \end{aligned}$$

и будем подставлять, что знаем, в их разность. Очевидно, $W'(U) \geq W(U)$. Разность всех элементов внешних сумм кроме j-х также оцениваем нулем. Итого, $\Phi(W') - \Phi(W) \geq W'(B_j) - W(B_j)$, а последняя разность по лемме 1.5 — обобщенная стоимость, так как B_j — минимизатор. Таким образом, полная обобщенная стоимость не превосходит

$$\Phi(W_{r_1}) - \Phi(W_\varepsilon) + \Phi(W_{r_1 r_2}) - \Phi(W_{r_1}) + \ldots = \Phi(W_\pi) - \Phi(W_\varepsilon),$$

где ε — пустая последовательность, а π — последовательность всех заказов. Нетрудно доказать, что $\Phi(W_\varepsilon)$ для фиксированного A_0 — константа.

Упражнение 1.1. Доказать это.

С другой стороны,

$$\begin{aligned} &\Phi(W_\pi) = \min \Psi_{W_\pi} \leq \Psi_{W_\pi}(A_n, \ldots, A_n) = \\ &k \cdot W_\pi(A_n) + \sum_{i=1}^{k} \left(W_\pi(A_n) - \sum_{a \in A_n} d(a_{ni}, a) \right) \leq \\ &2k \cdot W_\pi(A_n). \end{aligned}$$

Вспомним, что $W_\pi(A_n)$ — это стоимость работы offline алгоритма на нашей последовательности запросов π. Применяя лемму 1.4, получаем требуемое. □

1.3 Задача о покрытии множествами (the online set cover problem)

Этот конспект отсутствует. См. `http://www.math.tau.ac.il/~nogaa/PDFS/aaabnproc2.pdf` (случай без весов).

$\boxed{\text{ПРОБЕЛ В КОНСПЕКТЕ.}}$

Глава 2

Приближенные алгоритмы

2.1 Оптимизационные задачи и приближенные алгоритмы

$\boxed{\text{ПРОБЕЛ В КОНСПЕКТЕ.}}$

2.2 Задача о покрытии множествами

2.2.1 Задача о покрытии множествами (вариант с весами)

Задача. Дано некоторое множество $U = \{u_1, u_2, u_3, \ldots, u_n\}$ и семейство его подмножеств: $S = \{S_1, S_2, \ldots, S_k\}$, $S_i \subseteq U$, каждому из которых сопоставлена стоимость $p_i \geq 0$. В сумме все эти множества $S_1, \ldots, S_k$ покрывают множество U; т.е. U содержится в объединении этих множеств. Задача заключается в том, что нам надо выбрать подмножество множества S, покрывающее U, наименьшей суммарной стоимости. Мы предъявим приближенный алгоритм для этой задачи, а затем используем его для решения другой (более практической) задачи.

Алгоритм. На каждом шаге мы будем пытаться покрыть как можно больше элементов. Т.к. некоторые множества стоят дороже, мы будем минимизировать удельную стоимость элементов.

$I := \emptyset$;

while $\bigcup\limits_{j \in I} S_j \neq U$ do

begin

$\forall i \notin I$ cost$[i] := p_i / |S_i \setminus \bigcup\limits_{k \in I} S_k|$;

(Знаменатель — количество элементов, которые не были покрыты до сих пор, но будут покрыты данным множеством. Т.е., мы вычислили для каждого множества удельную стоимость покрываемых им элементов, теперь выбираем наилучшее по эффективности.*)*

Найдем такое i_0, что cost$[i_0]=\min\limits_{i\notin C}(\text{cost}[i])$;

$I := I \cup \{i_0\}$;

end;

Теперь докажем, что этот алгоритм является H_n-приближенным, где $H_n = 1+1/2+\ldots+1/n$, а $n = |U|$. Для начала мы пронумеруем все элементы множества u в том порядке, как мы их покрывали. $U = \{u_1, u_2, \ldots, u_n\}$. Каждому u_i из этих элементов мы припишем c_i — ту самую стоимость cost$[i_0]$, которая была у множества, которым этот элемент впервые покрыли.

Лемма 2.1. $c_i < P^*/(n-i+1)$, *где* P^* *— стоимость оптимального решения задачи.*

Доказательство. Мы собираемся покрыть i-ый элемент множества U, к этому моменту времени у нас уже что-то покрыто, а что-то нет. Но мы точно можем покрыть оставшееся при помощи множеств общей стоимости $\le P^*$ — значит, мы можем найти конкретное множество, эффективность (cost) которого $\le P^*/|U \setminus \bigcup\limits_{i\in I} S_i|$ (ясно из соображений о среднем). Но знаменатель $\ge n-i+1$, поскольку мы покрываем i-ый элемент, т.е. покрыто пока лишь $i-1$ элементов. □

Теперь при помощи этой леммы докажем, что алгоритм является H_n-приближенным. Для этого мы должны подсчитать суммарную стоимость нашего решения:

$$\sum_{i\in I} p_i = \sum_{j=1}^{n} c_j \le P^*(1 + 1/2 + \ldots + 1/n)$$

(равенство — поскольку все элементы можно разделить между множествами в соответствии с тем, каким множеством элемент был впервые покрыт), что и требовалось доказать.

Применим этот алгоритм к решению одной «биологической» задачи.

2.2.2 Задача о кратчайшей общей надпоследовательности

Задача: дано множество строк $\{s_1, \ldots, s_k\}$; нас интересует самая короткая строка u, которая содержит в качестве подстроки каждую из s_i.

Решение. Сведем к предыдущей задаче. Мы должны построить универсальное множество U, и множество, которое его покрывает. Не умаляя общности, можно считать, что среди строк нет подстрок друг друга. Построим строки $w_{ijk} = s_i \cdot s_j[k+1..|s_j|]$ для тех i, j, k, для которых суффикс s_i длины k совпадает с префиксом s_j длины k. Для любой строки s определим set$(s) = \{s_i \mid s_i$ — подстрока $s\}$. Стоимость множества set(s) — длина строки s. Вход задачи о покрытии — все множества set(s_i) и set(w_{ijk}). (Ясно, что $U = \{s_1, \ldots, s_k\}$.) Запускаем алгоритм, он выдает строки, мы их сливаем и получаем строку-ответ. То, что алгоритм является $2H_k$-приближенным, является очевидным следствием доказанного выше про алгоритм для задачи о покрытии множествами и следующей леммы.

Лемма 2.2. *Стоимость оптимального решения этой задачи о покрытии $\leq 2\cdot$длина решения исходной задачи.*

Доказательство.

```
-------------------------   s
--------- s1
     -------  s2
       ---------  s3
             ------------  s4
```

Пусть строка s — оптимальное решение исходной задачи. Перенумеруем s_i в порядке их первого вхождения в строку s. Очевидно, каждая следующая строка начинается и заканчивается позже предыдущей.

Разделим наши строки на блоки. Строим 1-ый блок: берем s_1 и все, первые вхождения в строку s которых начинаются до конца первого вхождения s_1. Сделаем из них (точнее, из первой и последней в этом блоке) общую строку (например, $w_{1,3,k}$) — это одно из множеств из условия задачи о покрытии.

Строим следующий блок, начиная его с первой невзятой строки, и т. д. Строки, которые мы сконструировали (по одной для каждого блока) задают некоторое решение задачи о покрытии множествами. Оно может и не быть оптимальным, но оптимальное — разве что еще меньше по стоимости.

Чтобы доказать, что стоимость этого решения — такая, как в формулировке, достаточно показать, что каждый символ строки s входит не более, чем в два блока. Однако, в одной позиции может пересекаться не более двух блоков (по построению). □

2.3 Задача о максимальном сечении

Определение 2.1. Задача о максимальном сечении (MAX-CUT). Пусть есть граф $G = (V, E)$, каждому ребру $e \in E$ которого приписан некоторый вес w_e. Рассмотрим все возможные раскраски вершин графа в белый и черный цвета (то есть отображения $j \colon V \to \{\text{Ч}, \text{Б}\}$), и каждой такой раскраске M сопоставим число $f(M)$, вычисляемое как сумма весов ребер, концы которых имеют в рассматриваемой раскраске M разные цвета (множество таких ребер назовем *сечением*). Наша задача состоит в том, чтобы найти такую раскраску M_0, чтобы $f(M_0)$ было бы максимальным. □

Будем решать эту задачу приближенно. Сначала переформулируем задачу как задачу оптимизации:

Определение 2.2. MAX-CUT как задача целочисленного линейного программирования. Сопоставим каждой вершине $v_i \in V$ с номером i число y_i, вычисляемое как

$$y_i = \begin{cases} -1, & v_i \text{ — белая вершина} \\ 1, & v_i \text{ – черная вершина.} \end{cases}$$

Тогда ребро $\{v_i, v_j\}$ принадлежит сечению в том, и только в том случае, когда

$$y_i y_j = -1,$$

то есть

$$f(M) = f((y_1, y_2, \dots, y_n)) = \sum_{i<j} \frac{1 - y_i y_j}{2} w_{ij}.$$

Наша задача состоит в том, чтобы найти максимум $f((y_i)_i)$ по всем наборам $(y_i)_i$, где каждое $y_i \in \{-1, 1\}$. □

Обозначим решение нашей задачи через M^* и попытаемся сначала решить более простую задачу:

Определение 2.3. Релаксация MAX-CUT: оптимизация на сфере большей размерности. Будем искать

$$\max_{v_1, \dots, v_n \in S^n} g((v_1, \dots, v_n)),$$

где

$$g((v_1, \dots, v_n)) = \sum_{i>j} \frac{1 - v_i \cdot v_j}{2} w_{ij},$$

а S^n — n-мерная единичная сфера (n — число вершин в графе G); умножение понимается как скалярное произведение. □

Обозначим этот максимум через R^*. Отметим, что

$$M^* \le R^*.$$

Этот факт следует из того, что наша новая задача содержит старую как частный случай (достаточно зафиксировать все переменные, кроме одной).

Теперь нам необходимо описать способ получения решения старой задачи из решения новой.

Алгоритм 2.1 (Randomized rounding). Пусть у нас есть набор $(v_i)_i$, являющийся решением новой задачи. Можно считать, что начало у всех векторов помещено в начало координат. Проведем через начало координат случайным образом гиперплоскость T размерности $n-1$. Эта гиперплоскость разделит пространство на два полупространства A и B. Теперь по каждому вектору v_i построим число y_i следующим образом:

$$y_i = \begin{cases} 1, & v_i \in A \\ -1, & v_i \in B. \end{cases}$$

Иначе говоря, выберем случайным образом вектор $u \in S^n$ и построим число y_i так:

$$y_i = \begin{cases} 1, & u \cdot v_i > 0 \\ -1, & u \cdot v_i < 0 \end{cases}$$

(случай $u \cdot v_i = 0$ будем интерпретировать как попало: вероятность этого события равна нулю). □

Теперь докажем следующую лемму про только что описанный алгоритм.

Лемма 2.3. *Пусть у нас теперь есть набор векторов v_i — (приближенное) решение задачи 2.3, R' — значение g на этом наборе. Пусть также $(y_i)_i$ - набор, полученный с помощью вышеописанного алгоритма из набора векторов v_i. Обозначим также $U' = f((y_i)_i)$. Тогда*

$$\mathbf{E}U' \geq \alpha R',$$

где $\alpha \approx 0.87$.

Доказательство. Нам надо доказать следующее неравенство:

$$\mathbf{E}\sum_{i<j} \frac{1 - y_i y_j}{2} w_{ij} \geq \alpha \sum_{i<j} \frac{1 - v_i \cdot v_j}{2} w_{ij}.$$

Возьмем некоторые y_i и y_j. Тогда $y_i y_j = -1$ в том, и только в том случае, когда вектора v_i и v_j лежат в разных полупространствах относительно нашей гиперплоскости T. Вычислим вероятность того, что $y_i \neq y_j$:

$$P(y_i \neq y_j) = \frac{\theta_{ij}}{\pi},$$

где θ_{ij} — угол между векторами v_i и v_j. Отсюда получаем, используя определение математического ожидания:

$$\mathbf{E}\sum_{i<j} \frac{1 - y_i y_j}{2} w_{ij} = \sum \frac{\theta_{ij}}{\pi} w_{ij}.$$

При этом

$$\sum_{i<j} \frac{1 - v_i \cdot v_j}{2} w_{ij} = \sum_{i<j} \frac{1 - \cos\theta_{ij}}{2} w_{ij}.$$

Теперь осталось взять

$$\alpha = \min_{\theta} \frac{2}{\pi} \frac{\theta}{1 - \cos\theta} \approx 0.87,$$

и утверждение леммы доказано. □

Определение 2.4. Матрица Y является положительно полуопределенной ($Y \succcurlyeq 0$), если для любого вектора x верно $x^{\mathrm{T}} Y x \geq 0$. □

Теперь еще раз переформулируем нашу задачу.

Определение 2.5. MAX-CUT как задача полуопределенного программирования. Необходимо максимизировать выражение

$$\sum \frac{1 - y_{ij}}{2} w_{ij}$$

по всем матрицам

$$Y = \{y_{ij}\}_{i,j=1}^{n}$$

таким что $Y \succcurlyeq 0$ и $\forall i \; y_{ii} = 1$. □

Приведем следующие три факта без доказательств:

Факт 2.1. *Задачу 2.5 можно решить приближенно за полиномиальное время и при этом получить результат $R^* - \varepsilon$.*

Факт 2.2. *Если матрица Y положительно полуопределена ($Y \succcurlyeq 0$), то она является матрицей скалярных произведений некоторого набора векторов v_i (заметим, что условие $y_{ii} = 1$ гарантирует, что эти вектора — единичные), причем эти вектора можно найти за полиномиальное время.*

Теорема 2.1. *Мы построили полиномиальный по времени вероятностный $(\alpha - \varepsilon)$-приближенный алгоритм для $\alpha \approx 0.87$ и произвольного $\varepsilon > 0$.*

Доказательство. Решение нашей задачи, согласно факту 2.2, свелось к решению задачи 2.5. Она, по факту 2.1, решается за полиномиальное время с аддитивной погрешностью не более ε'. Таким образом, по лемме, мы построили искомый $(\alpha - \varepsilon)$-приближенный алгоритм. □

Факт 2.3. *Этот алгоритм можно сделать детерминированным.*

Факт 2.4. *Если бы существовал такой алгоритм для $\alpha > 0.95$, то было бы $\mathbf{P} = \mathbf{NP}$.*

Факт 2.4 утверждает, что мы получили довольно хороший результат с теоретической точки зрения.

2.4 Задача о минимальном вершинном покрытии

Определение 2.6. Пусть $G = (V, E)$ — граф. Назовем множество $A \subseteq V$ *вершинным покрытием графа* G, если для каждого ребра этого графа хотя бы один из его концов принадлежит покрытию.

Рассмотрим взвешенный граф (т.е. каждой вершине сопоставлен некоторый вес). Наша задача состоит в том, чтобы приближенно найти минимальное в смысле веса вершинное покрытие. *Будем постепенно сводить эту задачу к другим, пока не сведем ее к нахождению максимального потока.* Именно,

вершинное покрытие $\xrightarrow{\times 2}$ двойное вершинное покрытие $\longrightarrow$ вершинное покрытие для двудольного графа $\longrightarrow$ s-t сечение для двудольного графа $\longrightarrow$ максимальный поток.

Таким образом, мы построим 2-приближенный алгоритм.

Сначала построим двудольный граф B следующим образом: его множество вершин есть удвоенное множество вершин исходного графа G (т.е. для каждой вершины $v \in V_G$ существуют вершины $v_1, v_2 \in V_B$), а ребра подчиняются следующем правилу: если между исходными вершинами u и v в графе G было ребро, то в графе B ребра будут между u_1 и v_2 и между u_2 и v_1. Ясно, что граф B является двудольным, причем все вершины с индексом «1» принадлежат одной доле, а с индексом «2» — другой.

Определим теперь понятие двойного покрытия графа G.

Определение 2.7. Отображение $f \colon V_G \to \{0, 1, 2\}$ называется *двойным покрытием графа* G, если для каждого ребра либо один из его концов имеет пометку 2, либо оба конца имеют пометку 1.

Лемма 2.4. *Минимальное вершинное покрытие графа B соответствует минимальному двойному покрытию графа G.*

Упражнение 2.1. Доказать лемму 2.4.

Лемма 2.5. *Пусть C — минимальное двойное покрытие G. Тогда его мощность как множества не превосходит удвоенной мощности минимального покрытия G.*

Упражнение 2.2. Доказать лемму 2.5.

Определение 2.8. Пусть G — некоторый связный граф. Назовем $s-t$-*сечением* графа G разбиение его вершин на два множества V_s и V_t, в одном из которых содержится вершина s, а в другом — t. *Вес сечения* — это суммарный вес всех ребер, соединяющих вершины V_s с вершинами V_t.

Добавим в граф B две вершины s и t и соединим вершину s со всеми вершинами одной доли, а t — со всеми вершинами другой. Припишем вновь добавленным ребрам такие же веса, какие были у соответствующих вершин в исходном графе, а старым ребрам — вес $+\infty$.

Лемма 2.6. *Минимальному покрытию графа B соответствует $s-t$-сечение минимального веса для только что построенного графа B'.*

Доказательство. Сопоставим каждому сечению множество вершин — концов его ребер, отличных от s и t. Получим покрытие. Аналогично обратный переход. Веса согласованы по построению. □

Лемма 2.7. *Минимальное $s-t$-сечение находится за время $O(n^3)$.*

Доказательство. Доказательство данной леммы непосредственно следует из следующего утверждения и оценки алгоритма нахождения максимального потока по методу проталкивания предпотока. □

Сначала дадим определение.

Определение 2.9 (Поток в графе). Пусть есть ориентированный граф G и в нем каждому ребру сопоставлен вес w. Этот вес понимается как пропускная способность данного ребра. Тогда *потоком* из вершины s в вершину t называется пометка ребер графа, удовлетворяющая условию: на каждом ребре значение потока не превосходит его пропускной способности, и, кроме того, сумма потоков на ребрах, входящих в вершину, не превышает суммы потоков на ребрах, выходящих из нее (кроме вершин s и t). Значением потока является сумма (со знаком) пометок ребер с началом (концом) в s.

Теперь сформулируем наше утверждение.

Лемма 2.8. *Максимальный поток определяет минимальное $s-t$-сечение.*

Доказательство. Действительно, рассмотрим максимальный поток. Вычтем его из пропускной способности ребер. Тогда на каждом пути из s в t найдется (иначе поток по этому пути можно увеличить) нулевое ребро. Его и добавим в сечение. Оно, по построению, минимально. □

2.5 Задача о рюкзаке

Определение 2.10 (Задача о рюкзаке). Из заданных предметов нужно выбрать такие, чтобы их суммарный вес был не более W, а стоимость – наибольшей. Точнее – заданы два множества, содержащие по n натуральных чисел: $w_1, \ldots, w_n$ и $p_1, \ldots, p_n$. Необходимо найти такое множество I, содержащееся в $[1..n]$, что $\sum\limits_{i \in I} w_i \le W$, и $\sum\limits_{i \in I} p_i$ максимально. НУО, $\max_{i \in [1..n]} w_i \le W$.

Как известно, в общем случае эта задача является $\mathcal{NP}$-полной. Так что будем строить приближенный алгоритм. Приведем так называемый псевдополиномиальный алгоритм для рюкзака, который будет полиномиальным от длины входа и $\max\limits_{i \in [1..n]} p_i$.

Пусть $S = \sum\limits_{i \in [1..n]} p_i$. Введем функцию w: $w(k, p)$:=минимальный объем, необходимый для того, чтобы уложить предметы с номерами, не превосходящими $k \in [1..n]$, общей стоимостью не менее $p \le S$ (если такого набора предметов нет, то приравняем функцию $+\infty$).

Вычислять эту функцию будем индуктивно. В цикле по k от 1 до n вычисляем $w(k, p)$ для каждого p от 1 до S следующим образом:

$$w(k+1, p) = \min\{w(k, p), w(k, p - p_{k+1}) + w_{k+1}\}.$$

Ясно, что этот алгоритм работает не более, чем nS, т.е. не более $n^2 \max p_i$.

Таким образом, мы могли бы решить нашу задачу за полиномиальное время, если бы p_i были достаточно маленькими.

Введем обозначения. Пусть задано ϵ, определяющее, с какой точностью мы хотим найти ответ. Обозначим A_ϵ общую стоимость набора предметов, который находится алгоритмом, который мы построим и про который покажем, что он дает $(1-\epsilon)$-приближение; $P = \max p_i$, $K_\epsilon = \frac{P}{(1+1/\epsilon)n}$.

Поделим все p_i на K_ϵ и округлим: $p'_i = \lceil p_i / K_\epsilon \rceil$. Теперь запустим наш псевдополиномиальный алгоритм для полученного набора чисел. Заметим, что $\max p'_i \le nP(1 + 1/\epsilon)/P = O(n(1 + 1/\epsilon))$. Таким образом, мы потратили время $\mathrm{poly}(n, 1/\epsilon)$.

Заметим теперь следующее неравенство: $A_\epsilon \ge A_0 - K_\epsilon n$, где A_0 — оптимальная стоимость предметов. Действительно, псевдополиномиальный алгоритм находит точное решение, поэтому отклонение в стоимости могло появиться только при округлении. При округлении мы могли потерять не более 1 на каждом предмете, которая домножилось на K_ϵ при обратном переходе от p'_i к p_i; итого, потеряли не более $K_\epsilon n$.

Далее,

$$\frac{A_\epsilon}{A_0} \ge \frac{A_0 - K_\epsilon n}{A_0} = 1 - \frac{Pn}{nA_0(1 + 1/\epsilon)} \ge 1 - \frac{1}{1 + 1/\epsilon} = \frac{1}{\epsilon + 1}.$$

В последнем неравенстве мы воспользовались очевидным фактом: $A_0 \ge P$ (действительно, ведь в рюкзак можно просто положить самый дорогой предмет). Итак, $A_\epsilon \ge (1/(1+\epsilon))A_0 \ge (1-\epsilon)A_0$.

2.6 Задача о раскраске графа

Предположим, что мы хотим покрасить вершины графа правильным образом (то есть так, чтобы концы любого ребра были покрашены в разные цвета).

Замечание 2.1. Ясно, что любой граф можно покрасить в $\delta + 1$ цвет, где δ — максимальная степень вершин этого графа.

Будем теперь рассматривать 3-раскрашиваемый граф. (Задача выяснения, можно ли покрасить граф в три цвета правильным образом, является $\mathcal{NP}$-полной.)

Рассмотрим произвольную вершину A и ее окрестность. В силу того, что рассматриваемый граф является 3-раскрашиваемым, окрестность вершины A можно покрасить в два цвета. Заметим, что покрасить граф в два цвета очень легко (а в нашем случае это, как мы выяснили, возможно): покрасим произвольную вершину в какой-нибудь из цветов; далее всех соседей уже покрашенных вершин будем красить в противоположные цвета (так нужно будет сделать для каждой компоненты связности). Теперь будем поступать следующим образом: если в графе есть вершина степени не менее $\sqrt{n}$, то красим ее окрестность в новые два цвета и выкидываем все эти вершины. Повторив не более $\sqrt{n}$ таких операций мы получим граф, содержащий лишь вершины степени менее $\sqrt{n}$. Его мы покрасим в новые $\sqrt{n}$ цветов (это сделать можно по замечанию, сделанному выше). Итого, мы использовали $3\sqrt{n}$ цветов для раскраски исходного графа.

Теперь зададимся целью покрасить граф в $\delta^{1/3}$ цветов. Назовем граф векторно 3-раскрашиваемым, если каждой его вершине можно сопоставить единичный вектор (из пространства $\mathbb{R}^n$), так что для любого ребра (i,j) будет выполняться равенство $v_i{\cdot}v_j = -1/2$. Ясно, что любой 3-раскрашиваемый граф является векторно 3-раскрашиваемым (достаточно три цвета заменить на такие три вектора плоскости, что угол между каждыми двумя из них будет $2\pi/3$).

Итак, найдем какой-нибудь[1] набор векторов, удовлетворяющий указанному свойству. Пусть r – случайный вектор пространства $\mathbb{R}^n$ (быть может, не единичный), заданный в соответствии с нормальным распределением. Обозначим: $U = \{i | r \cdot v_i \geq c\}$ (константу c мы определим позже). Если в этом множестве есть ребра, то выкинем по одной вершине от каждого ребра. Пусть $n' = |U|$, $m' = |\{(i,j) \in E | i,j \in U\}|$.

Найдем теперь $\mathbf{E}(n' - m')$ (число $n' - m'$ будет размером полученного множества; само множество будет независимым (определяли в лекции 9)). $\mathbf{E}n' = n \cdot P\{$вершина попадет в множество $U\} = n \cdot P(c)$, где $P(c) = P\{v_i \cdot r \geq c\} = \int_c^\infty \phi(x)dx$, а $\phi(x) = (1/\sqrt{2\pi})e^{-x^2/2}$ — плотность нормального распределения.

$$\mathbf{E}m' = \sum_{(i,j)\in E} P\{v_i{\cdot}r \geq c,\ v_j \cdot r \geq c\} \leq \sum_{(i,j)\in E} P\{(v_i{+}v_j){\cdot}r \geq 2c\} = P(2c){\cdot}|E|,$$

так как $|v_i + v_j|^2 = (v_i)^2 + (v_j)^2 + 2\, v_i \cdot v_j = 1$.

[1] Как? При помощи задачи полуопределенного программирования.

Итак, $\mathbf{E}(n'-m') \geq P(c)\cdot n - P(2c)\cdot(n\delta)/2 = n(P(c)-(\delta/2)P(2c))$. Теперь воспользуемся таким фактом:

$$\left(\frac{1}{x}-\frac{1}{x^3}\right)\cdot\phi(x) \leq P(x) \leq \frac{1}{x}\phi(x).$$

Тогда

$$P(c)/P(2c) \geq ((1/c-1/c^3)\cdot e^{-c^2/2})/(1/2c)\cdot e^{-2c^2} \geq 2(1-1/c^2)\cdot e^{3c^2/2}.$$

Хотим, чтобы получившееся число было не меньше, чем δ. Для этого достаточно взять $c=\sqrt{2/3\ln\delta}$ (тогда $c=O(\sqrt{\ln\delta})$). Итого,

$$\mathbf{E}(n'-m') \geq n\cdot(P(c)-\frac{1}{2}P(c)) = \frac{P(c)n}{2} \geq$$
$$\frac{1}{2}\cdot\frac{1}{\sqrt{2\pi}}\cdot(\frac{1}{c}-\frac{1}{c^3})\cdot e^{-c^2/2} = \omega(n/(\delta^{1/3}\sqrt{\ln\delta})).$$

Таким образом, на каждом шаге мы будем получать независимое множество размера $\omega(n/(\delta^{1/3}\sqrt{\ln\delta}))$. Ясно, что за $O(\delta^{1/3}\sqrt{\ln\delta}\log n)$ шагов мы покрасим весь граф (на каждом шаге мы красим найденное независимое множество в новый цвет и выкидываем покрашенные вершины) в $O(\delta^{1/3}\mathrm{polylog}(n))$ цветов.

Задача 2.1. Пользуясь полученными фактами, показать, что за полиномиальное время граф может быть покрашен в $O(n^{1/4}\mathrm{polylog}(n))$ цветов.

2.7 Задача об объединении множеств

2.7.1 Метод Монте-Карло

Рассмотрим следующую проблему, связанную с теорией вероятностей: пусть мы проводим m испытаний, вероятность успеха в каждом из которых есть p_i. Определим случайную величину y_i следующим образом: y_i принимает значение 1 с вероятностью p_i и значение 0 с вероятностью $1-p_i$. Определим

$$\mu = \mathbf{E}\sum_{i=1}^{m} y_i = \sum_{i=1}^{m} p_i.$$

Теперь, если обозначить

$$Y = \sum_{i=1}^{m} y_i,$$

то справедливы следующие *оценки Чернова*:

Факт 2.5.

$$P\{Y > (1+\varepsilon)\mu\} < \left(\frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}\right)^{\mu} \leq e^{-\frac{\mu\varepsilon^2}{4}},$$
$$P\{Y < (1-\varepsilon)\mu\} < e^{-\frac{\mu\varepsilon^2}{2}}.$$

Теперь рассмотрим собственно метод Монте-Карло.

Определение 2.11. Задача о нахождении мощности подмножества. Пусть у нас есть два конечных множества: U (элементы которого можно эффективно выбирать случайно с равномерным распределением) и его подмножество G. При этом мощность множества U известна. Мы хотим найти мощность множества G. $\square$

Для решения этой задачи существует следующий хорошо известный вероятностный алгоритм, называемый *методом Монте-Карло.*

Алгоритм 2.2. Будем брать случайным образом точки множества U и проверять их на принадлежность к множеству G. Произведя m экспериментов, определим величину

$$\widetilde{\rho} = \frac{\sum_{i=1}^{m} y_i}{m}$$

и будем считать, что оня является приближенным значением величины

$$\rho = \frac{|G|}{|U|}.$$

Отсюда найдем приближенно мощность множества G. $\square$

Очевидно, что чем больше экспериментов мы произведем, тем точнее мы определим ρ. Из оценок Чернова вытекает следующее соотношение:

$$P\{\widetilde{\rho} \notin [(1-\varepsilon)\rho; (1+\varepsilon)\rho]\} \leq 2e^{-\frac{m\rho\varepsilon^2}{4}},$$

из которого следует основная теорема:

Теорема 2.2. *Для того, чтобы с вероятностью как минимум $1-\delta$ получить $\widetilde{\rho}$, удовлетворяющее*

$$(1-\varepsilon)\rho \leq \widetilde{\rho} \leq (1+\varepsilon)\rho,$$

достаточно провести

$$\frac{4}{\varepsilon^2\rho} \ln \frac{2}{\delta}$$

экспериментов.

Таким образом, чтобы знать, сколько экспериментов надо произвести, надо знать значение ρ, но это как раз то, что мы ищем! Эта проблема преодолима, если мы знаем какую-нибудь не слишком маленькую нижнюю оценку для ρ. Однако, может оказаться что ρ просто мало само по себе.

2.7.2 Мощность объединения множеств

Рассмотрим следующую задачу:

Определение 2.12. Задача о мощности объединения множеств. Пусть есть n множеств H_i, мощности которых известны. Необходимо найти мощность их объединения $\bigcup_{i=1}^{n} H_i$. $\square$

Для решения этой задачи воспользуемся методом Монте-Карло. Однако, интересующая нас величина, как мы знаем, может быть мала, поэтому мы построим другой универсум. Введем следующие обозначения:

$$
\begin{aligned}
&G = \bigcup_{i=1}^{n} H_i,\\
&U = \{(v,i) \mid v \in H_i\},\\
&G' = \left\{(v,i) \in U \,\middle|\, i = \min_{k \in 1..n} \{k : v \in H_k\}\right\}.
\end{aligned}
$$

Важно, чтобы из каждого из множеств H_i можно было эффективно выбирать элементы случайно с равномерным распределением. Тогда мы сможем это делать и с множеством U (выбрать индекс i с вероятностью, пропорциональной размеру множества, а затем — элемент множества H_i).

Отметим, что мощность множества G' совпадает с мощностью нужного нам объединения:

$$
|G'| = \left|\bigcup_{i=1}^{n} H_i\right|.
$$

Определим теперь величину ρ как

$$
\rho = \frac{|G'|}{|U|}.
$$

Эту величину мы можем узнать из метода Монте-Карло. Мощность множества U нам также известна:

$$
|U| = \sum_{i=1}^{n} |H_i|.
$$

Таким образом, мы найдем мощность множества G', то есть решим нашу задачу.

Осталось лишь оценить количество испытаний. Для этого оценим величину ρ снизу:

$$
\rho \geq \frac{1}{n}.
$$

Эта оценка вытекает из того, что

$$
|F'| = |\cup H_i| \geq \max |H_i| \geq \frac{1}{n} \sum |H_i|.
$$

Отсюда получаем выражение для количества испытаний

$$
m \geq \frac{4}{\varepsilon^2 \rho} \ln \frac{2}{\delta} \geq \frac{4n}{\varepsilon^2} \ln \frac{2}{\delta},
$$

достаточного для того, чтобы вычисленное значение ρ отклонялось от истинного не более чем на ε с вероятностью $1 - \delta$.

2.8 Задача о коммивояжере в метрическом пространстве

Имеется полный граф G с расстояниями, удовлетворяющими неравенству треугольника. Необходимо обойти все вершины графа, вернувшись в начальную и пройдя при этом как можно меньше, то есть найти гамильтонов цикл минимального веса.

Предъявим 2-приближенный алгоритм. Найдем сначала минимальное остовное дерево T графа G. Теперь продублируем каждое ребро найденного дерева T и в полученном графе найдем эйлеров цикл (цикл, проходящий по всем ребрам графа ровно по одному разу). И наконец, преобразуем эйлеров цикл в гамильтонов следующим образом: вершины в гамильтоновом цикле будут идти в порядке их первого появления в эйлеровом при его обходе с произвольной вершины. (Другими словами, едем по эйлерову циклу и те вершины, в которых мы еще не были, записываем в гамильтонов цикл. Когда же встретим вершину, в которой уже были, просто ее перепрыгнем.) Видно, что все эти операции могут быть выполнены за полиномиальное (даже, в большинстве случаев, за линейное) время.

Покажем, что предъявленный алгоритм действительно 2-оптимальный. Пусть W_T — вес минимального остовного дерева, W_{opt} — вес оптимального гамильтонова цикла. Ясно, что $W_T \le W_{opt}$, так как при выкидывании любого ребра из гамильтонова цикла мы получаем остовное дерево. Каждое ребро построенного гамильтонова цикла заменяет какой-то путь эйлерова цикла, длина которого составляет не менее длины этого ребра (по неравенству треугольника). Таким образом, вес построенного гамильтонова цикла не превосходит $2W_T$, а значит, не превосходит и $2W_{opt}$, чтд.

Теперь улучшим наш алгоритм до 3/2-оптимального. Вместо того, чтобы дублировать каждое ребро остовного дерева, поступим следующим образом: найдем минимальное паросочетание всех вершин дерева T нечетной степени (ясно, что таких вершин четное количество). Добавив ребра найденного паросочетания в дерево T, получим эйлеров граф (то есть такой, в котором степени всех вершин четны). Найдем в этом графе эйлеров цикл и преобразуем его в гамильтонов (как это сделать, было описано выше).

Задача 2.2. Найти минимальное паросочетание.

Докажем, что получившийся алгоритм является 3/2-оптимальным. Аналогично предыдущему доказательству вес построенного гамильтонова цикла будет не более $W_T + W_P$, где W_P — вес минимального паросочетания вершин нечетной степени дерева T. Остается показать, что $W_P \le W_{opt}/2$. Пусть A — это множество всех вершин нечетной степени дерева T. Рассмотрим такой гамильтонов цикл множества A: в нем вершины множества A будут идти в такой последовательности, в какой они идут в оптимальном гамильтоновом цикле графа G. Ясно, что его вес будет не более W_{opt}. (Естественно, сам этот цикл мы не строим. Нам важно лишь то, что он существует.) Теперь разобьем множество вершин построенного гамильтонова цикла на четные и нечетные. Ясно, что мы получим два паросочетания, вес одного из которых будет меньше $W_{opt}/2$. Итак, мы показали, что существует паросочетание множества A веса не более $W_{opt}/2$, значит, и вес минимального паросочетания не превосходит $W_{opt}/2$, чтд.

Глава 3

Вероятностные алгоритмы

3.1 Вероятностные алгоритмы

$\boxed{\text{ПРОБЕЛ В КОНСПЕКТЕ.}}$

3.2 Проверка результата алгоритма перемножения матриц

Итак, мы знаем уже несколько алгоритмов перемножения матриц, но у нас нет хорошего способа проверки таких алгоритмов (и реализующих их программ). Рассмотрим вероятностный алгоритм, который даст нам возможность проверять результат быстрее, чем считать произведение.

Возьмем случайный вектор $\mathbf{r}$, составленный из случайных битов (принимают и 1, и 0 с равными вероятностями). У нас уже есть результат перемножения матриц $\mathbf{A}$ и $\mathbf{B}$ – матрица $\mathbf{C}$, полученная нами. Будем проверять равенство:

$$\mathbf{A} \times \mathbf{B} = \mathbf{C}.$$

Домножим обе части справа на случайный вектор $\mathbf{r}$. Теперь проверим новое равенство

$$\mathbf{AB} \times \mathbf{r} = \mathbf{C} \times \mathbf{r}.$$

На проверку его уйдет меньше времени, трудоемкость такой проверки равна $O(m * n)$, где n – длина вектора $\mathbf{r}$, m – количество строк матрицы $\mathbf{C}$. Если $\mathbf{C}$ квадратная, то трудоемкость выражается проще – $O(n^2)$. Вспомним, что трудоемкость при перемножении матриц была близка к $O(n^{2.8})$. Докажем, что этот алгоритм действительно проверяет результат перемножения.

Теорема 3.1. $\forall$ *матриц* $\mathbf{A}, \mathbf{B}, \mathbf{C}$
a) $\mathbf{AB} = \mathbf{C} \Rightarrow$ *алгоритм проверки не ошибается,*
b) $\mathbf{AB} \neq \mathbf{C} \Rightarrow$ *алгоритм ошибается с вероятностью не более чем* $\frac{1}{2}$.

Доказательство. Пункт a) очевиден, рассмотрим пункт b).
Известно, что $(\mathbf{AB} - \mathbf{C})\mathbf{r} \neq \emptyset$. В каком случае алгоритм ошибется? Если

скажет, что $\mathbf{AB} = \mathbf{C}$, то есть если $(\mathbf{AB}-\mathbf{C})\mathbf{r} = \emptyset$. Возьмем строчку матрицы $\mathbf{X} = \mathbf{AB} - \mathbf{C}$, не равную $\emptyset$ (помним, что сейчас у нас $\mathbf{AB} \neq \mathbf{C}$). Пусть x_{kl} – ненулевой элемент этой строчки. Тогда произведение k-ой строки на $\mathbf{r}$ выглядит так:

$$\sum_{i\in\{1,2,\ldots,\widehat{l},\ldots\}} x_{ki}r_i + x_{kl}r_l = 0, \quad \text{где } x_{kl} \neq 0. \tag{3.1}$$

Обозначим

$$c := -\frac{1}{x_{kl}} \sum_{i\in\{1,2,\ldots,\widehat{l},\ldots\}} x_{ki}r_i.$$

С какой вероятностью $r_l = c$? С вероятностью выбрать бит r_l равным биту c, то есть с вероятностью $\frac{1}{2}$. Следовательно, алгоритм ошибается с вероятностью не более $\frac{1}{2}$. □

Такой метод проверки называется *fingerprinting*.

Итак, наш алгоритм правильно решает задачу (т.е. говорит, что данная ему программа верно вычисляет произведение $\mathbf{A}$ и $\mathbf{B}$), если $\mathbf{AB} = \mathbf{C}$, и ошибается (говорит "верно", хотя на самом деле "неверно") с вероятностью не более $\frac{1}{2}$, если $\mathbf{AB} \neq \mathbf{C}$. Алгоритмы такого типа называются вероятностными алгоритмами с *односторонней ограниченной вероятностью ошибки (one-sided bounded error)*. Какова реальная польза от такого алгоритма? Ведь вероятность ошибки очень велика. Но если этот алгоритм применить 10 раз, то вероятность ошибки станет $\left(\frac{1}{2}\right)^{10}$, а это уже менее 0.001.

3.3 Метод «отпечатков пальцев» в применении к задачам со строками

1. Сравнение на равенство двух строк. Есть две строки a, b (можно считать их битовыми), которые необходимо сравнить на совпадение, затратив как можно меньше информации на это сравнение. Например, надо сравнить файлы по сети.

Идея алгоритма – сравнивать не сами строки, а функции от них. Пусть длина $\sharp a$ строки a составляет n бит.

Пусть $p \in \mathbb{P}$, $\mathbb{P}$ – множество простых чисел. В качестве функции-хэша возьмем $\mod p$. То есть будем сравнивать уже

$$a \mod p \quad \text{и} \quad b \mod p \tag{3.2}$$

Для такого сравнения достаточно передать $\log p$ битов (здесь и далее поумолчанию берется двоичный логарифм, $\log_2$) и еще столько же битов понадобится для передачи числа p.

Будем брать случайное простое число p из интервала $[2..\tau]$ для некоторого τ, которое определим позже. Плохими p для нас будут такие, которые будут давать равенство в (3.2) при неравенстве исходных строк a, b. Количество таких чисел равно

$$\sharp\{p \in P : (a-b)\vdots p\} \tag{3.3}$$

Задача 3.1. Как выбрать простое число из заданного интервала случайным образом с равномерным распределением?

Лемма 3.1. *$c \leq 2^n \Rightarrow c$ имеет не более n различных простых делителей.*

Доказательство. Очевидно. □

Пусть $\tau = n^2 \log n^2$. Тогда вероятность ошибки при сравнении остатков $\bmod p$ можно оценить

$$P_{\text{ошибки}} \leq \frac{n}{\frac{\tau}{\log \tau}} = \frac{n(\log n^2 + \log\log n^2)}{n^2 \log n^2} = O\left(\frac{1}{n}\right).$$

Таким образом, привели вероятностный алгоритм с односторонней ошибкой с вероятностью ошибки $O(\frac{1}{n})$, требующий передачи всего $O(\log n)$ битов.

Определение 3.1. *Расстоянием* между двумя строками a, b будем считать количество несовпадающих у них битов.

Задача 3.2. Придумать алгоритм для нахождения расстояния d между двумя строками a, b длины n. Посчитать количество передаваемых битов, как функцию $T(n, d)$.

Определение 3.2. Под *editing distance* между a, b будем понимать минимальное количество операций редактирования, необходимых для преобразования строки a в строку b. Операциями редактирования считаем:

1. вставку бита
2. замену бита
3. уничтожение бита
4. перемещение сплошного блока битов

Задача 3.3. Придумать алгоритм для нахождения editing distance между двумя строками a, b длины n. Посчитать трудоемкость $T(n, d)$.

2. Поиск вхождения строки a в строку b. Займемся теперь такой задачей. Нужно определить, входит ли строка a в строку b. Длины строк a и b равны, соответственно, m и n. Пусть $m \leq n$. Ясно, что решая ее в лоб, получим сложность почти $O(mn)$.

Предъявим вероятностный алгоритм с линейной сложностью $O(m+n)$.

Замечание 3.1. Существует детерминированный алгоритм поиска подстроки в строке за время $O(m+n)$, но он гораздо сложнее.

Определим

$$b(i) = b_i b_{i+1} \dots b_{i+m-1}.$$

Необходимо провести $n - m + 1$ сравнений строк на равенство, а это мы уже умеем делать: будем сравнивать

$$(a \mod p) \quad \text{и} \quad (b(i) \mod p).$$

Но можно упростить задачу, вычисляя $(b(i) \mod p)$ через $(b(i-1) \mod p)$.

$$b(i) = b_i + 2b_{i+1} + .. + 2^{m-1}b_{i+m-1}$$
$$b(i-1) = 2b_i + 2^2 b_{i+1} + .. + 2^{m-1}b_{i+m-2} + b_{i-1}$$
$$\Rightarrow b(i) = \frac{b(i-1) - b_{i-1}}{2} + 2^{m-1}b_{i+m-1}$$

Опять будем брать простое число $p \in [2, \tau]$. $\tau = n^2 m \log n^2 m$,

$$P_{\text{ошибки}} \leq \frac{m}{\frac{\tau}{\log \tau}} \leq \frac{2m \log n^2 m}{n^2 m \log n^2 m} = O\left(\frac{1}{n^2}\right)$$

Можно вообще избавить этот алгоритм от необходимости ошибаться. Если

$$a \mod p = b(i) \mod p,$$

то честно проверим равенство $a = b$. Этот алгоритм уже не ошибается. Какова его сложность? Математическое ожидание времени его работы $T(n, m)$ легко посчитать:

$$\mathbf{E}T(n, m) \leq mn * \frac{1}{n} + (m+n)(1 - \frac{1}{n}) = O(m+n).$$

То есть предьявили искомый линейно-быстрый алгоритм поиска подстроки в строке, причем это *вероятностный алгоритм с нулевой ошибкой*.

Упражнение 3.1. Исследовать работу (вероятность ошибки и сложность) такого алгоритма: если $(a - b(i)) \vdots p$, то выбираем новое простое p' и меняем p на p'.

Задача 3.4. Обобщить алгоритмы на случай двумерного пространства. То есть реализовать поиск блока в матрице.

3.4 Минимальное сечение (MIN-CUT)

Пускай нужно построить минимальное сечение графа с n вершинами и m ребрами (о том, что такое сечение, см. лекцию 5 в первой части курса). Будем считать, что все веса в графе равны единице.

Поступаем так: берем случайное ребро в графе и стягиваем две вершины в одну, получая граф с кратными ребрами (но без циклов). После некоторого количества таких операций у нас останется две вершины; ребра между ними соответствуют какому-то (быть может, не минимальному) сечению в исходном графе. Посчитаем вероятность ошибки, т.е. того, что сечение – не минимально. Пусть k — это вес минимального сечения (зафиксируем конкретное минимальное сечение M); тогда степень любой вершины — не менее k. Ошибка возникнет, если на каком-то шаге мы стянем в точку ребро из M. Пусть p — вероятность этого (на каждом конкретном шаге),

$$p \leq \frac{k}{\frac{nk}{2}} \leq \frac{2}{n}.$$

Тогда

$$P\{\text{успеха}\} \geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\dots\left(1-\frac{2}{3}\right) = \frac{2(n-2)!}{n!} = \frac{2}{n(n-1)}$$

Это плохо, т.к. придется повторять процедуру $O(n^2)$ раз, чтобы получить вероятность ошибки, ограниченную константой. Вместо этого будем производить стягивание ребер только до тех пор, пока не останется $\lceil \frac{n}{\sqrt{2}}+1\rceil$ вершин. Сделаем для данного графа H это дважды; так мы получим два графа: H_1 и H_2. Применим этот алгоритм рекурсивно к обоим и вернем то сечение, которое окажется меньше. База рекурсии тривиальна (граф, состоящий из двух вершин).

Пусть t — остающаяся глубина рекурсии. Пусть $P_i^{(t)}$ — вероятность того, что ни одно ребро из выбранного нами выше минимального сечения M не было стянуто при переходе от графа H (полученного при данном рекурсивном вызове) к графу H_i (передаваемому далее по рекурсии),

$$P_i^{(t)} \geq \left(1-\frac{2}{n}\right)\dots\left(1-\frac{2}{\lceil \frac{n}{\sqrt{2}}+2\rceil}\right) = \frac{2}{n(n-1)}\frac{\lceil \frac{n}{\sqrt{2}}+1\rceil\lceil \frac{n}{\sqrt{2}}\rceil}{2} \geq \frac{1}{2}.$$

Пусть $P^{(t)}$ — вероятность того, что, получив на вход мультиграф, в котором сечение M еще есть, алгоритм вернет именно это сечение (при условии, что будет достаточно рекурсивных вызовов глубины t),

$$P^{(t)} \geq 1-\prod_{i=1,2}\left(1-P_i^{(t-1)}P^{(t-1)}\right) \geq 1-\left(1-\frac{1}{2}P^{(t-1)}\right)^2 = P^{(t-1)}-\frac{(P^{(t-1)})^2}{4}.$$

Возьмем q_t так, что $P^{(t)} = \frac{4}{q_t+1}$. Тогда

$$\frac{4}{q_t+1} = \frac{4}{q_{t-1}+1} - \frac{4}{(q_{t-1}+1)^2} = \frac{4q_{t-1}}{(q_{t-1}+1)^2},$$

$$q_t = q_{t-1}+1+\frac{1}{q_{t-1}}.$$

Далее, так как $P^{(0)} = 1$, то $q_0 = 3$. Отсюда $t < q_t < 3+t+H_t$, где $H_t = 1+\frac{1}{2}+\ldots+\frac{1}{t}$. Значит, $P^{(t)} = \Omega(\frac{1}{t})$; поскольку глубина рекурсии $O(\log n)$, имеем $P^{(t)} = \Omega(\frac{1}{\log n})$, т.е. достаточно $O(\log n)$ повторов для того, чтобы вероятность ошибки стала ограничена константой.

Теперь оценим трудоемкость алгоритма. На стягивание ребер на одном шаге тратится время $O(n^2)$.

(Лучше написать рекуррентное уравнение.)

ПРОБЕЛ В КОНСПЕКТЕ.

Оно повторяется $O(\log^2 n)$ раз (один логарифм от рекурсии, а другой — чтобы получить вероятность ошибки, ограниченную константой). Итого: время работы алгоритма составляет $O(n^2\log^2 n)$.

3.5 Минимальное остовное дерево

Остовное дерево — это дерево, состоящее из некоторых ребер графа и содержащее все его вершины. Требуется построить остовное дерево с минимальным суммарным весом ребер. (Считаем, что все ребра разного веса: этого можно добиться, если к равным ребрам добавить достаточно малые ε_i.) НУО будем считать веса всех ребер различными.

Сперва рассмотрим простой детерминированный алгоритм: **алгоритм Borůvka**. Выберем у каждой вершины ребро наименьшего веса. После этого в полученном графе каждую компоненту связности стянем в вершину. При каждой такой итерации число вершин уменьшается вдвое. Так что сложность этого алгоритма $O(m \log n)$. Фактически, этот алгоритм мало чем отличается от алгоритма Краскала, но его можно улучшить следующим образом:

Алгоритм 3.1. По графу G_1 с n вершинами и m ребрами строим минимальный остовный лес (ибо граф может быть несвязен) для него.

1. Применим к G_1 3 шага алгоритма Borůvka - получим граф G_2 и некое множество ребер S, которые принадлежат остовному дереву. В графе G_2 максимум $\frac{n}{8}$ вершин.

2. Из G_2 выкинем примерно половину ребер (именно, каждое ребро выкинем с вероятностью $\frac{1}{2}$) — это будет граф $G_2(\frac{1}{2})$ (в нем $\leq \frac{n}{8}$ вершин, а мат. ожидание количества ребер $\leq \frac{m}{2}$.

3. Для $G_2(\frac{1}{2})$ применяем алгоритм рекурсивно; получим F — минимальный остовный лес для этого графа.

   Далее требуется определение. Пусть в графе H есть остовное дерево (или лес) T. Ребро, соединяющее вершины v и w называется **тяжелым в H относительно T**, если его вес больше веса максимального ребра на пути из v в w в T. Иначе ребро называется **легким**. Понятно, что тяжелые ребра нас не интересуют (если такое ребро есть в “минимальном” остовном дереве, то дерево можно уменьшить, заменив это ребро другим).

   Задача 3.5. Найти все легкие ребра за линейное время.

4. Пусть V_2 — множество ребер G_2, легких относительно F. Возьмем только их — получим граф G_3.

5. Рекурсивно обрабатываем G_3 и выдаем полученный результат.

□

Пусть $T(n, m)$ — мат. ожидание времени работы нашего алгоритма на графе с n вершинами и m ребрами (более строго, максимум этого мат. ожидания по всем графам). На рекурсивный вызов от $G_2(\frac{1}{2})$ тратится время $T(\frac{n}{8}, \frac{m}{2})$. Покажем, что на рекурсивный вызов от G_3 тратится время $T(\frac{n}{8}, \frac{n}{4})$.

Лемма 3.2. *Пусть $G(p)$ — граф, полученный из какого-то графа G выкидыванием каждого ребра с вероятностью $1-p$, и F — минимальный остовный лес в $G(p)$. Тогда мат. ожидание числа легких ребер в G относительно F не превосходит $\frac{n}{p}$, где n — число вершин.*

Доказательство. Лес F будем строить одновременно с графом $G(p)$. Упорядочим ребра G по возрастанию весов. Очередное ребро с вероятностью p добавляем в $G(p)$. Если добавленное ребро соединяет в F разные компоненты связности, то добавляем его в F (как в алгоритме Краскала). Таким образом, в F попадет $n-1$ ребро. Легкими в G будут эти $n-1 \leq n$, а также те, которые попали бы в F, если бы их случайно не отбросили. Так как в среднем "отсев проходит" каждое $\frac{1}{p}$-е ребро, то всего легких ребер будет в среднем не более, чем $\frac{n}{p}$. □

Упражнение 3.2. Доказать лемму 3.2 более формально.

ПРОБЕЛ В КОНСПЕКТЕ.

Лемма 3.3. *Мат. ожидание количества ребер в графе G_2, легких относительно леса F, не превосходит $n/4$.*

Доказательство. В этом графе $\frac{n}{8}$ вершин, а $p = \frac{1}{2}$. Применяем предыдущую лемму. □

Таким образом, для мат. ожидания времени работы нашего алгоритма, очевидно, верно соотношение:

$$T(n,m) \leq T\left(\frac{n}{8}, \frac{m}{2}\right) + T\left(\frac{n}{8}, \frac{n}{4}\right) + c(m+n).$$

Упражнение 3.3. Доказать, что $T(n,m) = O(n+m)$.

3.6 Проверка простоты числа

См. лекции для 1го курса.

ПРОБЕЛ В КОНСПЕКТЕ.

На протяжении этой лекции все числа — неотрицательны. Для начала вспомним несколько определений.

Символ Лежандра:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & , \text{ уравнение } x^2 \equiv a \pmod{p} \text{ имеет корни} \\ -1 & , \text{ в противном случае} \end{cases},$$

где p — простое, $a \neq 0$.

Символ Якоби:

$$\left(\frac{a}{N}\right) = \prod_i \left(\frac{a}{p_i}\right),$$

если $N = p_1 \cdots p_k$ — разложение N на простые множители (среди p_i могут быть одинаковые). Некоторые свойства:

$$\begin{aligned}
\left(\frac{a}{N}\right) &= (-1)^{\frac{N-1}{2}\frac{a-1}{2}}\left(\frac{N}{a}\right) \quad (\text{здесь } a, N \not\vdots 2, \text{НОД}(a,N)=1),\\
\left(\frac{a}{N}\right) &= \left(\frac{a'}{N}\right) \quad (\text{при } a \equiv a' \pmod{N}),\\
\left(\frac{1}{p}\right) &= 1.
\end{aligned}$$

Упражнение 3.4. Вспомнить, как вычислять $\left(\frac{2}{p}\right) = (-1)^?$.

Алгоритм 3.2.
Вход: число N.
Выход: «composite» или «prime».

$$\text{Если } N\vdots 2 \text{ или } N = 1\text{, выдать правильный ответ.} \tag{3.4}$$
$$M \leftarrow random[2..N-1] \tag{3.5}$$
$$\text{if } (M,N) \neq 1 \text{ then answer “composite”} \tag{3.6}$$
$$\text{else if } \left(\frac{M}{N}\right) \not\equiv M^{\frac{N-1}{2}} \pmod{N} \tag{3.7}$$
$$\text{then answer “composite”} \tag{3.8}$$
$$\text{else answer “prime” (тут алгоритм может ошибиться)} \tag{3.9}$$

□

Корректность шага (3.7) доказывает следующая лемма:

Лемма 3.4. $N \in \mathbb{P}$, $N \neq 2 \implies M^{\frac{N-1}{2}} \equiv \left(\frac{M}{N}\right) \pmod{N}$.

Доказательство. Доказывается в курсе алгебры. □

Лемма 3.5. *Пусть* $N \not\vdots 2$, $N \neq 1$. *Если для всех* M, *таких что* $(M,N) = 1$, *выполняется* $\left(\frac{M}{N}\right) \equiv M^{\frac{N-1}{2}} \pmod{N}$, *то* N *— простое.*

Доказательство. Будем доказывать от противного:

1. Пусть N не содержит квадратов: $N = p_1 \cdots p_k$, $p_i \neq p_j \in \mathbb{P}$. Фиксируем r такое, что $\left(\frac{r}{p_1}\right) = -1$ (такое есть: пересчитаем все квадраты $\bmod\ p_i \ldots$). По китайской теореме об остатках можно выбрать такое M, что

$$\begin{aligned}
M &\equiv r \pmod{p_1},\\
M &\equiv 1 \pmod{p_i} \quad \text{при } i \neq 1.
\end{aligned}$$

С одной стороны,

$$\left(\frac{M}{N}\right) = \left(\frac{M}{p_1}\right) \cdot \prod_{i\neq 1}\left(\frac{M}{p_i}\right) = -1.$$

С другой стороны,

$$M^{\frac{N-1}{2}} \equiv 1 \pmod{p_2} \not\equiv -1 \pmod{N}.$$

Противоречие.

2. Пусть N содержит квадраты: $N = p^2 n$, $p \in \mathbb{P}$. Пусть r — первообразный корень по модулю p^2. По предположению,

$$r^{N-1} \equiv (r^{(N-1)/2})^2 \equiv \left(\frac{r}{N}\right)^2 \equiv 1 \pmod{N},$$

а значит, и $(\bmod\ p^2)$. Т.е., одновременно $N-1 \vdots p(p-1)$ и $N \vdots p$, т.е., два последовательных числа делятся на p. Противоречие.

□

Лемма 3.6. *Если $N \notin \mathbb{P}$, то для более чем половины всех $M \in [2..N-1]$, взаимно простых с N, $\left(\frac{M}{N}\right) \not\equiv M^{\frac{N-1}{2}} \pmod{N}$.*

Доказательство. По лемме 3.5 существует такое число a, взаимно простое с N, что $\left(\frac{a}{N}\right) \not\equiv a^{\frac{N-1}{2}} \pmod{N}$. Пусть $b_1,\ b_2, \ldots,\ b_k$ — это все остатки, для которых выполнено сравнение $\left(\frac{M}{N}\right) \equiv M^{\frac{N-1}{2}} \pmod{N}$.

Рассмотрим $ab_1,\ ab_2, \ldots,\ ab_k \pmod{N}$. Они все различны, так как если $ab_i \equiv ab_j \pmod{N}$, то $b_i \equiv b_j \pmod{N}$ (ведь $(a, N) = 1$). Значит, их не менее k. При этом для них сравнение не выполняется:

$$\left(\frac{ab_i}{N}\right) = \left(\frac{a}{N}\right)\left(\frac{b_i}{N}\right) = \left(\frac{a}{N}\right) \cdot b_i^{\frac{N-1}{2}} \not\equiv (ab_i)^{\frac{N-1}{2}}.$$

□

Тем самым, вероятность ошибки нашего алгоритма не превосходит $1/2$.

3.7 Кратчайшие пути до всех вершин графа

3.7.1 Предисловие: постановка задачи

Наша задача такова. Представим, что нам нужно найти кратчайшие пути между парами вершин связного неориентированного графа (длины всех ребер мы считаем равными 1, так что расстояние между вершинами — это просто количество ребер на пути из одной в другую).

Как известно, алгоритм Дейкстры работает за время $O(n^3)$. Попытаемся найти алгоритм, работающий быстрее.

Будем искать такую матрицу S, что $S_{i,j}$ — это номер первой вершины на кратчайшем пути из вершины i в j. Ясно, что по такой матрице мы будем знать все кратчайшие пути между парами вершин.

3.7.2 Поиск матрицы расстояний между вершинами

Найдем матрицу расстояний между всеми парами вершин графа (назовем ее D; $D_{i,j}$ — расстояние между вершинами i и j).

Пусть A — матрица смежности исходного графа (размера $n \times n$). Рассмотрим матрицу A^2. Заметим, что $(A^2)_{i,j} = 0$ тогда и только тогда, когда в нашем графе не существует пути из i в j длины ровно 2. Действительно, если в графе нет пути длины ровно два из i в j, то множества $N(i)$ и $N(j)$ (множества соседей вершин i и j) не пересекаются, то есть не существует такого k, что $A_{i,k} = A_{k,j} = 1$ (а именно такое k нужно, чтобы $(A^2)_{i,j} = \sum_k A_{i,k}A_{k,j}$ было отлично от нуля).

Теперь рассмотрим квадрат исходного графа G (назовем его G') — такой граф, в котором ребро между вершинами u и v есть тогда и только тогда, когда в графе G существует путь между этими двумя вершинами длины не более двух, причем $u \neq v$. В матрице A' будут стоять единицы только там, где стояли единицы в матрице A или A^2, а на главной диагонали в любом случае поставим все нули.

Особый случай: граф G' - полный. Ясно, что тогда $D = 2 \cdot A' - A$.

В общем же случае будем искать матрицу D рекурсивно. Пусть D' - матрица расстояний для графа G' (найденная рекурсивным вызовом нашего алгоритма для графа G').

Интуитивно ясно, что расстояния в G примерно в 2 раза больше, чем в G'. Примерно, но не совсем, а именно:

Лемма 3.7. *Если $D_{i,j} \vdots 2$, то $D_{i,j} = 2 \cdot D'_{i,j}$; в противном случае $D_{i,j} = 2 \cdot D'_{i,j} - 1$.*

Доказательство. Очевидно. □

Но этого еще не достаточно для нахождения числа $D_{i,j}$ через $D'_{i,j}$, ведь для этого нам нужно знать четность $D_{i,j}$.

Лемма 3.8. *Пусть k — сосед i ($k \in N(i)$). Тогда, если $D_{i,j} \vdots 2$, то $D'_{k,j} \geq D'_{i,j}$. Если $D_{i,j} \not\vdots 2$, то $D'_{k,j} \leq D'_{i,j}$; более того, в этом случае найдется такое k, что $D'_{k,j} < D'_{i,j}$ (ясно, что этой вершиной будет первая вершина на кратчайшем пути из i в j).*

Рассмотрим матрицу M, равную произведению матриц A и D'. Заметим, что $M_{i,j} = \sum_k A_{i,k}D'_{k,j} = \sum_{k \in N(i)} D'_{k,j}$. Таким образом, если $D_{i,j} \vdots 2$, то $M_{i,j} \geq (A^2)_{i,i} \cdot D_{i,j}$; в противном случае — $M_{i,j} < (A^2)_{i,i} \cdot D_{i,j}$ (заметим, что $(A^2)_{i,i}$ — степень вершины i в графе G).

Итак, проверив полученное неравенство для каждого элемента матрицы, мы найдем D для графа G рекурсивно, зная D' для графа G'. Рекурсия закончится не более, чем за $O(\log n)$ шагов, ибо на каждом шаге рекурсии диаметр графа (наибольшее расстояние между вершинами) сокращается примерно вдвое (а точнее? — упражнение!); в итоге мы придем к нашему особому случаю — графу G диаметра 2 (соответственно, полному графу G'), разобранному отдельно.

3.7.3 Поиск «виновников» в умножении булевых матриц

Интуитивное соображение: Для расстояния $D_{i,j}$ "виновник" того, что расстояние между i и j именно такое — это путь длины $D_{i,j}$. Но сейчас мы будем искать "виновников" для другой задачи.

Пусть $A \wedge B = P$, где A, B, P — булевы матрицы. Ясно, что если $P_{i,j} = 1$, то для некоторого k выполняется $A_{i,k} = B_{k,j} = 1$. Такое k назовем *"виновником"*.

Рассмотрим теперь матрицу A^*, такую что $A^*_{i,k} = k \cdot A_{i,k}$. Пусть $P^* = A^* \cdot B$ (внимание: здесь умножение целое, а раньше было $\wedge$ — булево). Тогда $P^*_{i,j}$ — сумма всех виновников. Значит, если существует ровно один виновник, то $P^*_{i,j}$ — виновник.

Упражнение 3.5. Пусть среди n шаров имеется w белых и $n - w$ черных. Случайно выбирается r шаров. Тогда если $n/(2w) \le r \le n/w$, то вероятность того, что среди выбранных шаров будет ровно один белый, не меньше $1/(2e)$.

Рассмотрим теперь случайный вектор $R = (r_1, \dots, r_n)$, в котором ровно r координат равны 1. Вычислим матрицы $A^{*,R}$ и B^R следующим образом:

$$A^{*,R}_{i,j} := kA_{i,k}r_k, \quad B^R_{k,j} := B_{k,j}r_k.$$

Вычислим также $C^R := A^{*,R} \cdot B^R$. Заметим, что вероятность того, что $C^R_{i,j}$ — виновник, не меньше $1/(2e)$, если соотношение между r и числом виновников w — такое же, как в упражнении 3.5.

Чтобы угадать r, для $r = 1, 2, 4, 8, \dots, 2^{\lfloor \log n \rfloor}$ повторим: выбираем случайно вектор R и проверяем за $O(1)$ операций, является ли $C^R_{i,j}$ виновником. Эту процедуру повторим $4 \log n$ раз.

Для данных i и j, оценим вероятность того, что за эти итерации мы найдем виновника. Ясно, что когда-нибудь r попадет в отрезок $[n/(2w); n/w]$, при этом по Факту 3.5 мы найдем виновника с вероятностью, не меньшей $1/(2e)$. После $4 \log n$ таких итераций вероятность того, что мы найдем виновника, не меньше $1 - (1 - 1/(2e))^{4 \log n} \le 1 - 1/n$. Таким образом, мат. ожидание количества ненайденных виновников не превосходит $n^2 \cdot 1/n$, т.е. n. На поиск оставшихся виновников мы затратим время $O(n^2)$ (ненайденных виновников, как мы выяснили, не более n).

На все описанные действия мы затратим время $F(n) = O(T(n) * \log^2 n)$, где $T(n)$ — время, необходимое для умножения двух матриц размера $n \times n$ с числами от 0 до n (ясно, что $F(n)$ больше n^2, но меньше n^3).

3.7.4 Объединяем все вместе

Покажем, как найти матрицу S. Заметим, что первая вершина на кратчайшем пути из i в j — это такое k, что $D_{i,j} = D_{k,j} + 1$. Ясно, что для любой вершины $k \in N(i)$, выполняется одно из трех следующих неравенств:

$$D_{i,j} = D_{k,j} + 1,$$
$$D_{i,j} = D_{k,j},$$
$$D_{i,j} = D_{k,j} - 1.$$

Итак, нас интересуют такие вершины k, что $D_{i,j} - D_{i,k} \equiv 1 \pmod 3$.

Рассмотрим теперь три матрицы $D^{(0)}$, $D^{(1)}$, $D^{(2)}$, такие что $D^{(s)}_{k,j} = 1$ тогда и только тогда, когда $D_{k,j} = (s-1) \pmod 3$. Теперь для (произведения) булевых матриц A и $D^{(s)}$ мы получим нашим алгоритмом матрицы виновников - назовем их $W^{(s)}$. И тогда матрицу S мы сможем найти следующим образом: $S_{i,j} = W_{i,j}^{(D_{i,j} \mod 3)}$.

Итого, время работы алгоритма: $O(T(n) \cdot \log^2 n)$, где $T(n)$ — время на умножение двух целочисленных матриц порядка $n \times n$, содержащих числа от 0 до n^2 (точнее, это их произведение может содержать такие числа — найдите, в каком именно месте алгоритма?..).

Глава 4

Параллельные алгоритмы

4.1 Параллельные вычисления

Введем следующие обозначения:

- $c(n)$ — количество процессоров,
- $T(n)$ — время работы,
- $W(n) = c(n)T(n)$ — общая работа.

Заметим, что если у нас есть программа, написанная для $c(n)$ процессоров, то мы всегда сможем переписать ее для меньшего количества $c_1(n)$; полученная программа будет работать время порядка $O(W(n) \cdot \frac{c(n)}{c_1(n)})$ (эту формулу надо, конечно, уточнить для конкретной модели вычислений).

Модель вычислений можно выбрать, например, такую: RAM-машина с $c(n)$ процессорами; у каждого — одна и та же программа; нулевой регистр у каждого свой (перед началом работы там записан номер процессора), остальные — общие. У нас может возникнуть проблема коллизий, то есть два процессора решат одновременно записать данные в один регистр. Ее можно разрешить несколькими способами, например,

- запретить такие програмы;
- записывать то, что записал процессор с большим номером.

Однако, лучше всего пользоваться другой моделью: булевыми схемами. Схема состоит из гейтов, у каждого из них есть «результат»:

- входных — в них закладываются данные (биты), входной бит и есть результат такого гейта;
- выходных — в них получается ответ (тот же, что подан такому гейту на вход);
- рабочих — у них есть входы (один или два) и результат: во входы закладываются операнды (биты), а на выходе получается результат операции (можно считать, что операций всего две: конъюнкция и отрицание).

Результат каждого гейта можно использовать несколько раз, направив его в качестве операндов в разные рабочие гейты. В итоге все гейты соединяются в ориентированный граф без циклов и получается подобная картинка:

ПРОБЕЛ В КОНСПЕКТЕ.

Каждый гейт схемы можно рассматривать как процессор, выполняющий одну свою операцию. Тогда временем работы этого многопроцессорного устройства, очевидно, соответствует длина самого длинного ориентированного пути от входного гейта к выходному (глубина схемы), а объему работы (т.е. общему количеству шагов, сделанному всеми процессорами) — размер схемы (заметим, что это определение работы *отличается* от определения для RAM-машин; однако, мы не будем доказывать в этом курсе эквивалентность схем и параллельных RAM-машин и выяснять, как меняются сложностные параметры при переходе от одной модели вычислений к другой). Конечно, параллельная RAM-машина способна проинтерпретировать схему размера $W(n)$ из $c(n)$ гейтов на $c'(n) \leq c(n)$ процессорах за время $W(n) \cdot \frac{c(n)}{c'(n)}$.

Однако, схема способна обрабатывать лишь входы заданной длины. Поэтому мы будем рассматривать семейства схем, по одной для каждой возможной длины входа. Чтобы такое семейство можно было использовать автоматически, необходимо, чтобы можно было *эффективно построить схему* для заданной длины входа. Обычно требуют, чтобы существовала детерминированная машина Тьюринга, порождающая схему (в разумном виде — так, чтобы ее вычисления можно было моделировать на машине Тьюринга, которой она будет задана вместе со входом) для входов длины n по записи числа n в «палочной» системе счисления и при этом использующая лишь $O(\log n)$ ячеек памяти (и, следовательно, лишь полиномиальное от n время). Такие семейства схем называются *равномерными.*

4.2 Достижимость в графе

Пусть A — матрица смежности. Если мы умножим ее булевски на себя, то получим те ребра, которые достижимы за два шага. Если в матрицу A записать на диагональ единички, то квадрат даст нам вершины, достижимые за два или менее шагов. Матрицу A^2 умножим еще раз на себя: мы получим вершины, достижимые за четыре или менее шагов, и т. д. Итого: нам нужен алгоритм, возводящий A в $(n-1)$-ую (или большую) степень. Будем считать[1], что $n-1=2^k$.

$A \ A \ A \ A \dots A \ A$

\/ \/ \/

$A^2 \quad A^2 \quad \dots \quad A^2$

$\dots\dots\dots\dots\dots\dots$

\/

A^{n-1}

[1] Ясно, что это несущественно.

При этом, конечно, не надо перемножать одни и те же матрицы много раз; это дерево можно заменить на DAG (ориентированный граф без циклов): вычислим A^2, затем возведем его в квадрат, и т. д. Всего одно вычислительное устройство может это проделать за $O(\log(n))$ операций над матрицами. Но произведение матриц не есть элементарная операция. Пусть мы хотим умножить матрицы B и C, из булево произведение — D; тогда $d_{ij} = \bigvee\limits_{k=1}^{n} b_{ik} \wedge c_{kj}$ можно вычислить следующим способом:

$b_{i1} \wedge c_{1j} \quad b_{i2} \wedge c_{2j} \ \ldots \ b_{in} \wedge c_{nj}$

```
      \ /   . . .   \ /
       \  /     \  /
        \/       \/
         \       /
          \    /
           \ /
```

d_{ij}

При этом каждый из элементов d_{ij} мы вычисляем на своем наборе процессоров. За $O(\log(n))$ операций и n^3 процессоров мы перемножим две матрицы. Всего же мы управимся за время $O(\log^2 n)$ и $O(n^3)$ процессоров.

4.3 Максимальное по включению независимое множество

Определение 4.1. $V' \subseteq V$ — *независимое множество* в графе (V, E), если $\forall u, v \in V'$ $\{u, v\} \notin E$.

Определение 4.2. Обозначим через $deg(v)$ степень (количество соседей) вершины v.

Алгоритм.
Вход: граф $G = (V, E)$.
Выход: независимое множество S.

Инициализация: $S := \emptyset$.

На каждой итерации мы будем делать следующее.

1. $\forall v \in V$:
   если $deg(v) = 0$, то добавить v в S;
   если $deg(v) \neq 0$, то пометить v с вероятностью $1/(2deg(v))$.

2. $\forall e \in E$:
   если оба конца ребра e помечены, то снять пометку с того, у которого степень меньше.

3. Запихать в S все помеченные вершины.

4. $V := V \setminus \{S \cup$ соседи вершин из $S\}$ (при этом надо и E обновить соответственно).

Повторять эти шаги будем, пока граф не станет пустым.

Анализ алгоритма. Корректность алгоритма очевидна. Шаги этого алгоритма можно выполнять параллельно для всех вершин или ребер, при этом (при разумной реализации) на один шаг будет тратиться время $O(\log n)$. Остается выяснить, сколько будет итераций: если их также $O(\log n)$, то мы получили искомый «быстрый» параллельный алгоритм. В дальнейшем этот факт не будет доказан полностью; однако, будут приведены соображения, подтверждающие это интуитивно.

Будем называть вершину *хорошей*, если $\geq 1/3$ ее соседей имеют степень меньшую, чем степень этой вершины.

Лемма 4.1. *Пусть v — хорошая вершина. Тогда вероятность, что один из ее соседей помечен $\geq 1 - e^{-1/6}$.*

Доказательство. Действительно, вероятность, что это не так, не превосходит $(1 - 1/2deg(v))^{deg(v)/3}$, что, в свою очередь, не больше $e^{-1/6}$. □

Лемма 4.2. *Вероятность снятия пометки — не более $1/2$.*

Доказательство. Данная вероятность не превосходит вероятности того, что помечен один из соседей большей степени; что не меньше, чем $deg(v) \cdot 1/(2deg(v))$, то есть $1/2$. □

Следствие 4.1. *Если v — хорошая, то с вероятностью $\geq (1 - e^{-1/6})/2$ хотя бы один из ее соседей попадет в S.*

Ребро e будем называть *хорошим*, если хотя бы один из его концов — хороший.

Лемма 4.3. *Количество хороших ребер $\geq |E|/2$.*

Доказательство. Направим ребра из вершин с меньшей степенью в вершины с большей[2] (теперь наш граф — ориентированный; пусть $deg_{in}(v)$ и $deg_{out}(v)$ — входная и выходная степени вершины v соответственно).

Через $E(V_1, V_2)$ обозначим количество ребер, идущих из вершин множества V_1 в вершины множества V_2; пусть V_g — множество всех хороших вершин, V_b — множество всех плохих вершин.

$$
\begin{aligned}
&2E(V_b, V_b) + E(V_b, V_g) + E(V_g, V_b) \leq \\
&\sum_{v\text{ — плохая}} deg(v) \leq \\
&3 \sum_{v\text{ — плохая}} (deg_{out}(v) - deg_{in}(v)) = \\
&3(E(V_b, V_g) + E(V_b, V_b) - E(V_g, V_b) - E(V_b, V_b)) \leq \\
&\qquad\qquad\qquad\qquad 3(E(V_b, V_g) + E(V_g, V_b)).
\end{aligned}
$$

Сократим начало и конец этого неравенства. Итого, у нас хороших ребер больше, чем плохих, а значит, их больше половины. □

Хороших ребер больше половины; каждое из них пропадает с вероятностью $\geq (1 - e^{-1/6})/2$; значит, мат. ожидание количества пропавших ребер $\geq (1 - e^{-1/6})|E|/4$. Выходит, что на каждой итерации количество ребер сокращается в константное число раз[3]; значит, алгоритм будет совершать лишь логарифмическое число итераций.

[2]И как угодно для ребер, соединяющих вершины одинаковой степени.

[3]Увы, лишь в среднем. Поэтому последующий вывод доказывается не так просто.

4.4 Задача о максимальном по включении независимом множестве

Рассмотрим неориентируемый граф.

Определение 4.3. *Независимое множество* — это множество вершин графа, между которыми нет ребер.

Следующий параллельный алгоритм находит максимальное (*по включению*) независимое множество за время $O(\log^2 |\text{длина входа}|)$.

Алгоритм 4.1.
Вход: граф $G = (V, E)$.
Выход: независимое множество S.

Инициализация: $S := \emptyset$.
На каждой итерации мы будем делать следующее.

1. $\forall v \in V$:
   если $\deg(v) = 0$, то добавить v в S;
   если $\deg(v) \neq 0$, то пометить v с вероятностью $1/(2\deg(v))$.
2. $\forall e \in E$:
   если оба конца ребра e помечены, то снять пометку с того, у которого степень меньше.
3. Добавить в S все помеченные вершины.
4. $V := V \setminus \{S \cup \text{соседи вершин из } S\}$ (при этом надо и E обновить соответственно).

Повторять эти шаги будем, пока граф не станет пустым. □

Замечание 4.1. Алгоритм корректный, каждый его шаг занимает (при разумной реализации) время не более $O(\log |E|)$ и требует не более $|E|$ процессоров.

Определение 4.4. v — *хорошая*, если не менее $\frac{\deg(v)}{3}$ ее соседей имеет степень не более чем $\deg(v)$.

Лемма 4.4. *Пусть v — хорошая.* $\Pr\{$*хотя бы один из соседей v помечен на шаге 2*$\} \geq 1 - e^{-\frac{1}{6}}$.

Доказательство. $\Pr\{\text{не так}\} = \prod\limits_{(u,v)\in E, \deg(u)\leq\deg(v)} \Pr\{u \text{ — не помечена}\} \leq (1 - \frac{1}{2\deg(v)})^{\deg(v)/3} \leq e^{-1/6}$. □

Лемма 4.5. *Вероятность снятия пометки не превосходит $\frac{1}{2}$.*

Доказательство. Эта вероятность для вершины v не превосходит

$$\text{кол-во соседок } v \text{ не меньшей степени} \cdot \Pr\{\text{отметить такую соседку}\}$$
$$\leq \deg(v) \cdot \frac{1}{2\deg(v)} = \frac{1}{2}.$$

□

Следствие 4.2. *Пусть v — хорошая. Тогда вероятность того, что хотя бы одна соседка вершины v включена в S на шаге 4 — не менее $\frac{1-e^{-1/6}}{2}$.*

Заметим, что событие, вероятность которого оценивается в следствии 4.2, влечет тот факт, что v будет удалена из графа на текущем шаге. Вместе с ней будут, разумеется, удалены и включающие ее ребра.

Определение 4.5. Ребро e хорошее, если хотя бы один из концов хороший.

Определение 4.6. $e(S,T) = \{$количество ребер из S в $T\}$.

Лемма 4.6. *Хороших ребер — не менее $\frac{|E|}{2}$.*

Доказательство. Ориентируем ребра так, что бы они были направлены из вершины меньшей степени в вершину большей степени (и как угодно, если степени совпадают); пусть $d_i(v)$ и $d_o(v)$ — входная и выходная степени вершины v, соответственно. Если v — плохая вершина, то строго менее 1/3 ее соседей имеют степень не более $\deg v$; стало быть, строго более 2/3 соседей имеют степень строго меньше. Таким образом, $d_o(v) > \frac{2}{3}\deg(v)$; следовательно, $d_i(v) < \frac{1}{3}\deg(v)$. Итого, $d_o(v) - d_i(v) > \frac{\deg(v)}{3}$. Пусть V_G — множество хороших вершин, V_B — множество плохих вершин, $e(V_1, V_2)$ — множество всех ребер из вершин множества V_1 в вершины множества V_2.

$$
\begin{aligned}
&2 \cdot e(V_B, V_B) + e(V_B, V_G) + e(V_G, V_B) = \\
&\sum_{v \text{ — плохая}} \deg(v) \leq \\
&3 \cdot \sum_{v \text{ — плохая}} (d_o(v) - d_i(v)) = \\
&3((e(V_G, V_B) + e(V_B, V_B)) - (e(V_B, V_G) + e(V_B, V_B))) = \\
&3(e(V_G, V_B) - e(V_G, V_B)) \leq 3(e(V_G, V_B) + e(V_B, V_G)),
\end{aligned}
$$

т.е.

$$e(V_B, V_B) \leq e(V_B, V_G) + e(V_G, V_B).$$

□

Следовательно, на одной итерации в среднем количество ребер уменьшается в константу раз:

Теорема 4.1. *Математическое ожидание количества ребер, исчезающих на одной итерации составляет не менее $\frac{1-e^{-1/6}}{4}|E|$ (где $|E|$ — текущее количество ребер в графе).*

Отсюда *не* следует непосредственно, что количество итераций составляет $O(\log|E|)$.

Упражнение 4.1. Тем не ме нее, доказать этот факт. (Указание: рассмотреть случайный процесс на отрезке $1..k$: как только количество ребер становится меньше c^k (c — некоторая константа), процесс сдвигается из k в $k-1$.)

4.5 Продолжение, возможно, следует