Теория сложности вычислений

IV семестр, весна 2016

лектор: Гирш Эдуард Алексеевич

Собрано: 31 августа 2016 г. 17:59

Оглавление

Автор: Черникова Ольга

Глава 1

Теория сложности

1.1. классы P и NP

1.1.1. Задачи поиска и распознования

Def 1.1.1. *Алфавит* — по умолчанию будем использовать $\{0,1\}$.
Обозначения алфавита — Σ

Def 1.1.2. *Слово* — элемент $\{0,1\}^*$, длина слова x обозначается $|x|$.
Множество слов длины n обозначается $\{0,1\}^n$

Def 1.1.3. *Язык* — некоторое множество слов (возможно, пустое или бесконечное).

Замечание 1.1.1. Мы считаем, что битовыми строками можно кодировать вообще любые объекты, которые можно описать. Кодировать будем, разумеется, наиболее разумно (если не сказано иное), чтобы размер строк был как можно меньше. Где важен тип кодирования — будем явно указывать.

Пример 1.1.1. Натуральные числа можно естественно закодировать битовой строкой — представление числа в двоичной системе счисления.

Пример 1.1.2. Пары битовых строк: (a,b) можно закодировать какой-то битовой строчкой. Тогда за $|(a,b)|$ естественным образом будет обозначаться длина этой строчки. В принципе, ожидаемо, что $|(a,b)| = \theta(|a|+|b|)$, по-хорошему надо выбрать один способ кодирования и для него доказать, но мы не стали.

Def 1.1.4. *Индивидуальная задача* — это пара $(a,b) \in \{0,1\}^* \times \{0,1\}^*$, где a, b — слова. a называется условием или входом, b называется решением для данного входа.

Def 1.1.5. *Массовая задача* R — некоторое множество R индивидуальных задач (возможно, пустое или бесконечное):

$$R \subseteq 2^{\{0,1\}^* \times \{0,1\}^*}$$

Также можно рассматривать как бинарное отношение на строках: $aRb \iff (a,b) \in R$ (читать как «b является решением задачи R с условием a») Мы будем писать и так, и так.

Замечание 1.1.2. Наиболее интересные массовые задачи — бесконечные, с возможностью проверить корректность решения.

Пример 1.1.3. Пологаем $\mathbb{N} \subset \{0,1\}^*$
$FRAC = \{(n,d) | n : d, 1 < d < n\}$

Def 1.1.6. Алгоритм решает задачу поиска для массовой задачи R, если для условия x он находит решение w, удовлетворяющее $(x, w) \in R$

Def 1.1.7. Алгоритм решает задачу распознования(языки), задачу типа да/нет. То есть алгоритм, который по x умеет понимать, существует ли w, такое, что $(x, w) \in R$

Замечание 1.1.3. Массовая задача для R задает язык:

$$L(R) = \{x | \exists w (x, w) \in R\}$$

Пример 1.1.4. $L(FRAC) =$ множество всех составных слов.

1.1.2. Машина Тьюринга

Общее определение

Машина Тьюринга (МТ) — вычислительная модель. Есть много модификаций, но у каждой МТ есть следующее:

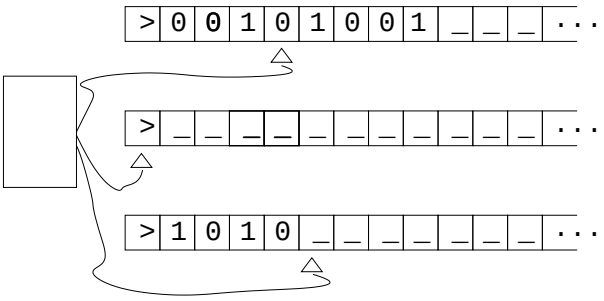
- Фиксированный конечный алфавит Σ, в котором имеется как минимум два специальных символа: пробел ($\textvisiblespace$) и специальный символ начала ленты ($\triangleright$).
- Фиксированное конечное число *лент*, бесконечных в одну сторону. Лента — это бесконечно много ячеек, занумерованных натуральными числами.
- Читающие/пишещие голвки, по одной для каждой ленты, каждая в один момент видет одну позицию
- Конечное множество *состояний* q_i. Выделяется начальное состояние q_s. Так же будем выделять терминальные состояния q_y и q_n.
- Конечное множество *переходов*. Каждый переход — это функция:

  $$(q, c_1, c_2, \dots, c_l) \to (c'_1, c'_2, \dots, c'_l, m_1, m_2, \dots, m_l r)$$

  при этом:

  - l — число лент
  - $c_i, c'_i \in \Sigma$
  - q, r — состояния
  - $m_i \in \{\leftarrow, \cdot, \to\}$ — *сдвиг головки* i

  Говорим, что это переход из q в r по символам на лентах $c_1, \dots, c_l$, который пишет на ленты $c'_1, \dots, c'_l$ и сдвигает головки в направлениях m_i.



Чтобы МТ что-нибудь вычисляла, вводится понятие *конфигурации* конкретной МТ M. Конфигурация машины тьюринга это все, что определяет ее дальнейшую судьбу, где располагается головка, память, переходы. Оно состоит из:

- Для каждой ленты хранится одно натуральное число, соответствующее какой-то ячейке — *позиция головки на этой ленте*
- Для каждой ленты хранится отображение $f\colon \mathbb{N} \to \Sigma$, которое в конечном числе точек может возвращать что угодно, а в остальных возвращает ␣. Также можно дополнительно потребовать $f(1) = \triangleright$, но это необязательно (соответствующие определения будут эквивалентны).
- Выделяется *текущее состояние* МТ q.

Дальше вводится понятие *корректного перехода из конфигурации* A в какие-то другие конфигурации. Чтобы сгенерировать список этих конфигураций, надо сделать следующее:

1. Рассмотреть все переходы из текущего состояния конфигурации A.
2. Оставить из них только те переходы, у которых символ c_i совпадает с символом на позиции головки на ленте i
3. Для каждого из оставшихся сказать, что мы можем перейти в конфигурацию, у которой текущее состояние изменено с q на r, символы c_i на лентах заменены на символы c'_i, соответственно, а позиции головок на каждой ленте независимо изменены в соответствии с m_i:
   - При $\leftarrow$ головка сдвигается на единицу влево
   - При $\cdot$ головка остаётся на месте
   - При $\rightarrow$ головка сдвигается на единицу вправо

Замечание 1.1.4. Изначально машина находится в начальной конфигурации, $q = q_s$, все голочки находятся на начальных позициях.

Замечание 1.1.5. Можно формально определить бинарное отношение «быть корректным переходом» между состояниями, аккуратно расписав алгоритм выше.

Замечание 1.1.6. Никакая головка не должна уезжать за край ленты. Есть разные способы этого добиться.

Модификации

Все модификации МТ ниже эквивалентны друг другу, мы выбираем ту, в которой удобнее работать в данной задаче. Для доказательства эквивалентности двух моделей, надо выбрать машину в одной модели и переделать.

Чтобы головка не уезжала налево за край ленты, можно считать по-разному:

- Переход, при котором головка уезжает налево некорректен
- Если головка уезжает налево, вычисление прекращается.

Def 1.1.8. *Детерминированная машина Тьюринга* (ДМТ) — если зафиксировать начальное состояние и множество символов под лентой, то переход либо есть и ровно один, либо его нет. Если перехода нет, то можно либо отвергать слово, либо переходить в какое-то спецсостояние (эквивалентно). Можно считать, что все переходы всегда должны быть (эквивалентно).

Замечание 1.1.7. Если у ДМТ зафиксировать конфигурацию, то следующая конфигурация (если есть) всегда определяется однозначно. Поэтому можно говорить о том, что ДМТ что-то однозначно вычисляет (или зацикливается).

Def 1.1.9. На ДМТ можно вычислять функции: на фиксированную (входную) ленту пишем вход, запускаем ДМТ, пока не завершиться, после этого ожидаем на фиксированной ленте (выходной) получить ответ. То есть можем решать задачу поиска.

Def 1.1.10. На ДМТ также можно вычислять ответы «да/нет»: говорим, что ДМТ обязана завершать работу в ровно одном из двух состояний — q_{yes} или q_{no}, и именно этим определяется результат работы. То есть решать задачу распознавания.

Def 1.1.11. Говорим, что ДМТ M распознает язык A, если принимает все $x \in A$, отвергает все $x \notin A$, пишем, что $A = L(M)$.

Метрики

Def 1.1.12. *Время работы M* — количество шагов до остоновки машины.

Def 1.1.13. *Используемая M память* — сумма следующей величины по всем лентам: максимальное правое положение головки на ленте.

Замечание 1.1.8. При сублинейных ограничениях на память первая лента(где вход) $read - only$ и положение головки на ней не считается.

Пример 1.1.5. Если машина на двух лентах сначала отвела первую головку вправо на 10, потом вернула, потом сделала то же самое со второй головкой, то используемая память — $10 + 10 = 20$.

1.1.3. Классы P и NP

Def 1.1.14. Массовая задача R полиномиально ограничена, если существует полином p, ограничивающий длину кратчайшего решения: $\forall x(\exists u\colon (x,u) \in R \Rightarrow \exists w\colon (x,w) \in R \wedge |w| \leqslant p(|x|))$

Def 1.1.15. Массовая задача R полиномиально проверяема, если существует полином q, ограничивающий время проверки решения: для любой пары (x,w) можно проверить принадлежность R за время $q(|(x,w)|)$.

Def 1.1.16. $\tilde{NP}$ — класс задач поиска, задаваемых полиномиально ограниченными, полиномиально проверяемыми массовыми задачами.

Def 1.1.17. $\tilde{P}$— класс задач поиска из $\tilde{NP}$, разрешимых за полиномиальное время, то есть задаваемых отношениями R, такими, что $\forall x \in \{0,1\}^*$ за полиномиальное время можно найти w, для которого $(x,w) \in R$.

Замечание 1.1.9. Ключевой вопрос теории сложности: $\tilde{P}? = \tilde{NP}$

Def 1.1.18. NP — класс задач распознавания(языков), задаваемых полиномиально ограниченными полиномиально проверяемыми массовыми задачами, то есть $NP = \{L(R) | R \in \tilde{NP}\}$

Иначе говоря, $L \in NP$, если имеется п.о. п.п. R, такая, что

$$\forall x \in \{0,1\}^*\colon x \in L \Leftrightarrow \exists w(x,w) \in R$$

Def 1.1.19. P — класс задач распознавания(языков) A, разрешимых за полиномиальное время
То есть существует ДМТ M и полином q, такой, что для любого слова x $M(x) = A(x)$ и машина отработает не больше, чем за $q(|x|)$.

Замечание 1.1.10. Ясно, что $P \subset \{L(R) | R \in \tilde{P}\}$

Очевидно, $P \subset NP$.

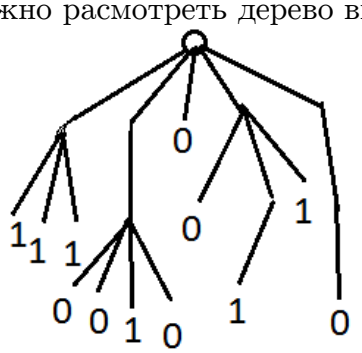
Замечание 1.1.11. Ключивой вопрос теории сложности: $P? = NP$

1.1.4. Еще определения NP

Определим НМТ.

Def 1.1.20. *Недетерминированная машина Тьюринга* (НМТ) — почти вся та же ДМТ, кроме того, что функция перехода может быть многозначной. То есть при данном состояние и данными символами под головкой возможны написания разных символов и разные сдвиги головки.

В НМТ может быть несколько вариантов переходов из одной конфигурации. В таком случае можно расмотреть дерево вычислений на НМТ:



То есть дерево вычислений это все варианты переходов в НМТ и на конце 0 если закончили в состояние q_N и 1, если в q_Y.

Def 1.1.21. НМТ принимает слово x, если остановилась в состоянии q_{yes} на хотя бы одной ветке.

Def 1.1.22. Время и память в НМТ это максимум по времени и памяти по всем возможным веткам развития.

Замечание 1.1.12. В машины(ДМТ, НМТ) с заведомо ограниченным временем работы можно встроить **будильник** и считать время вычисления на входах одной длины всегда одним и тем же.

Теперь определение НМТ, через ДМТ:

Def 1.1.23. Недетерминированная машина Тьюринга(НМТ) — это просто ДМТ, у которой есть дополнительный аргумент (подсказка w на второй ленте)

Def 1.1.24. НМТ M принимает вход x, если существует w, для которой вычисление заканчивается в q_y(пишем $M(x, w) = 1$).

Замечание 1.1.13. Вычислительный путь в старом определение это тоже самое, что подсказка в новом. Можно считать, что длина подсказки определяется длиной входа.

Def 1.1.25. NP — класс языков, принимаемых полиномиальными по времени НМТ.

Замечание 1.1.14. Последние определения класса NP, можно считать двумя, так как у нас есть несколько определений НМТ.

1.2. Сводимости

1.2.1. Сводимость по Карпу

Эта сводимость обычно имеется в виду в случае задач распознавания.

Def 1.2.1. Сведение по Карпу: Язык L_1 сводится к языку $L_2(L_1 \to L_2)$ если, имеется полиномиально вычислимая f: $\forall x : x \in L_1 \Leftrightarrow f(x) \in L_2$

Замечание 1.2.1. Мы не можем поменять местами ответ, то есть мы не можем сказать, что там где нет мы скажем да, а там где да сажем нет. Это важно. Так как есть подсказка для второго языка, то эта же подсказка для первго языка, а подсказки у нас для "да".

1.2.2. Сводимость по Левину

Сводимость применяется в случае задач поиска.

Def 1.2.2. Сведение задач по Левину: Задача R_1 сводится к задаче $R_2 (R_1 \to R_2)$, если $\exists f, g, h \colon \forall x_1, y_1, y_2$:

1. $R_1(x_1, y_1) \Rightarrow R_2(f(x_1), g((x_1, y_1)))$
2. $R_1(x_1, h((f(x_1), y_2))) \Leftarrow R_2(f(x_1), y_2)$
3. f и h полиномиально вычислимы, а g ограничена полиномом.

Замечание 1.2.2. Функцию g мы не собираемся запускать, но нам важно, что бы решение для второго случая было полиномиально ограницено, если полиномиально ограничено в первом случае.

Видимо, наше седение будет происходить следующим образом. Мы отображаем наше x_1 с помощью функции f, находим в R_2 нужные нам решения и отображаем их назад с помощью функции h.

Замечание 1.2.3. Классы $P, NP, \tilde{P}, \tilde{NP}$ замкнуты относительно этих сведений.

1.3. Полные задачи

1.3.1. Определение полных задач

Def 1.3.1. Задача A — трудная для класса C, $\forall B \in C \colon B \to A$. То есть любая задача из класса C сводится к задаче A.

Def 1.3.2. Задача полная для класса C, если она трудная и принадлижит C.

Теорема 1.3.1. Если A — NP-трудная и $A \in P$, то $P = NP$.

Следствие 1.3.1.1. Если A — NP-полная, то $A \in P \Leftrightarrow P = NP$.

1.3.2. Задача об ограниченной остановке

Def 1.3.3. 1^t — обозначение для записи слова, состоящего из t единиц.

Def 1.3.4. Задача об ограниченной остановке: $\tilde{BH}(< M, x, 1^t >, w)$ = НМТ M с подсказкой w принимет вход x за $\leqslant t$ шагов.

Теорема 1.3.2. Задача об ограниченной остановке — $\tilde{NP}$-полная, а соответствующей язык — NP-полный.

Замечание 1.3.1. Принадлежность $\tilde{NP}$ использует существование универсальной ДМТ, которая может промоделировать вычисление ДМТ, описание которой дано ей на входе. Причем лишь с полиномиальным замедлением.

▶ **NP-трудность** Есть фиксированная задача L из класса NP, то есть есть соответствующая ей машины Тьюринга M, которая работает $p(n)$
Воспользуемся сведением по Карпу. $f(x) =< M, x, 1^{p(|x|)} >$).
Давай-те убедимся, что это правильное сведение. Если у x существовала подсказка w, то и у новой машины эта же подсказка подходит и наоборот.
Действительно, если в новом решение подошла подсказка w, значит для изначальной M подходила подсказка w. Теперь, если у изначальной машины была подсказка w' и слово принадлежало языку, то существовала подсказка, w, которую можно было получить за меньше, чем $p(n)$ шагов, значит уже это подсказка подойдет для $f(x)$.
Функция f является полиномиально вычислимой.

Принадлежность NP Теперь, что бы удостоверится, что новая задача принадлежит NP, нужно значть, что бывает универсальная детерминированная машина тьюринга, которая с неболее чем полиномиальным ухудшением сможет промоделировать переданную машину тьюринга на t шагов. Пока в такую машину поверим, чуть поже будет рассказано, как именно ее построить.

◀

1.3.3. $CIRCUIT\_SAT$

Def 1.3.5. Булева схема это:

1. Ориентированный граф без циков.
2. Бинарные (и унарные) операции над битами: $\wedge, \vee, \oplus$ в вершинах графа
3. Есть выделеные вершины, которые являются входом схемы. Вершины без входящих ребер.
4. Есть выделенные вершины, которые являются выходом схемы.

Def 1.3.6. $CIRCUIT\_SAT = \{(C, x) | C —$ схема, $C(x) = 1\}$
То есть массовая задача для $CIRCUIT_SAT$ — это для данной схемы найти вход, на котором эта схема будет давать 1.

В случае задачи распознавания, узнать, существует ли такой набор.

Теорема 1.3.3. $CIRCUIT\_SAT$ — NP-полная задача.

▶ **Принадлежность NP:** Если у нас есть схема и подсказка, то достаточно не сложно проверить результат за полином от размера схемы. Можно, например, сделать топ сорт вершин и за квадрат последовательно вычислять значения в ячейках.

NP-трудность: Что бы доказать NP-трудность, сведем уже известную нам NP-полную задачу BH к нашей. Тогда все задачи из NP будут сводится к BH, BH будет сводится к нам и как следствие все задачи из NP будут сводится к нам.

1. Каждый этаж схемы будет соответсовать конфигурация ДМТ.
   Конфигурацию шифруем как [состояние, положение головки, память до головки, символ под головкой, память после головки; так же для следующих лент]

   Ну да, каждый символ алфавита шифруется каким-то количеством бит.
2. Размер этажа можем ограничесть $const \cdot t$. Так как больше памяти, чем работает машина тьюринга мы не израсходуем.
3. Количество этажей схемы соответствует времени работы ДМТ. Каждый этаж конфигурация на данном шаге выполнения ДМТ.
4. Переход между этажами реализует один шаг ДМТ
5. Вход схема — подсказка НМТ.
   Так, как схема строится для конкретной Машины тьюринга, входа и времени, то все остальное уже намертво вбито в нашу построенную схему.

Теперь подробнее разберем, как устроен переход между этажами.
Для этого научимся программировать с помощью схем несколько конструкций.
if x then y else z такую инструкцию можно записать, например в виде (x and y) | (not x and z)

Понятно как проверить, что какой-то символ чему-то равен. Берем ячейки отвечающие за этот символ и соответсвенно and.
Теперь хотим узнать, что будет в какой-то конкретной ячейке.
if (головка где-то недалеко) то что-то содержательное
else оставляем символ таким же.

Что-то содержательное - это много if. У нас будет отдельно в схеме хранится переходы по состояниям.
if вида, если такое-то состояние и такие-то сиволы под головкой, то записать в данную ячейку то-то.
И в конце исправить положенеи головки.
Получается не так мало if, но все равно константа, поэтому не так важно.
Ячеек в схеме получается порядка $\Theta(|x|^2)$

Можно уменьшить до $|x|log(|x|)$ с использованием oblivious универсальной машины тьюринга. То есть если изначальная Машина Тьюринга была oblivious, то мы все сделали за линию, пересчитывая гейты только в окрестности головки и мы так могли бы сделать, так как все время знаем положение головки в любой момент времени вне зависимости от входа.

◀

1.3.4. 3-SAT

Def 1.3.7. $3-\widetilde{SAT} = \{(F,A)|F$— в 3-КНФ$, F(A)=1\}$
У нас есть формула в 3—КНФ, хотим для нее найти выполняющий набор.
В случае задач распознавания, узнать, существует ли выполняющий набор.

Теорема 1.3.4. $3-SAT$ — NP-полная задача.

▶ **Принадлежность NP:** Проверка выполняющего набора — не очень хитрое дело. Подставляем 0 и 1 и дальше уже распарсить выражение, тем более в 3-КНФ - совсем просто.

NP-трудность: Сведем $CIRCUIT\_SAT$ к $3-SAT$.
Теперь как мы это будем делать:

1. Заведем для каждого гейта новую переменную.
2. Теперь гейт выражает какую-то операцию $g(x,y)$. Для примера $\oplus$. Тогда хотим выписать несколько клозов, которую этот гейт выразят.
   Это мы сможем сделать не более чем за 8 клозов.
3. Склеим полученный результат. Напишем конъюнкцию всех этих дизъюнктов.
4. Добавляем клоз, что на выходе получили 1.

◀

Замечание 1.3.2. Это наиболее известная NP-полная задача.

1.4. Универсальная машина тьюринга:

1.4.1. Универсальная машина тьюринга:

Def 1.4.1. $U(M,x) = M(x)$ — универсальная машина тьринга.
Тут мы говорим про ДМТ. Вход для универсальной ДМТ U — пара из ДМТ и входа для неё — (M,x).

Выход у универсальной машины Тьюринга должен совпадать, с выходом машины, которую она моделирует.

Интересно то, сколько шагов затратит U на эмуляцию M, если M работала за t шагов.

Замечание 1.4.1. Стоит помнить, что в универсальной машине фиксированное число лент, число состояний и размер алфавита.

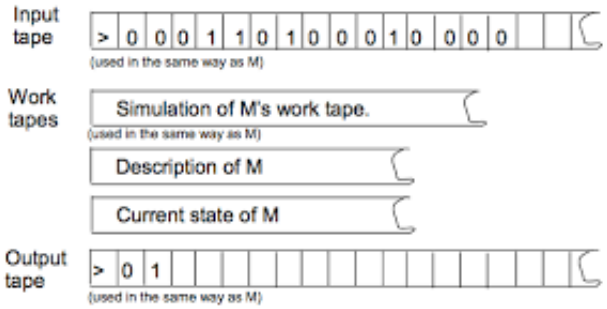
В Машине которую мы эмулируем все это может быть сильно больше.

Проблема с большим алфавитом в машине M лечится преобразованием машины M в машину M', которая пользуется алфавитом $\{0, 1, \sqcup, \triangleright\}$ и кодирует каждый старый символ фиксированным числом бит. Замедление M' по сравнению с M будет лишь константное.

Если нам требуется моделировать только M с ограниченным числом лент $\leqslant k$, то это легко делается машиной U с $k + 1$ лентой с константным замедлением (т.е. U_M будет работать для конкретной M за $\mathcal{O}(t)$). На этой отдельной ленте мы храним описание M и текущее состояние, а первые k лент и позиции головок на них в точности соответствуют машине M. Прочитали за не зависящее от t время символы под головками, нашли нужный переход, перешли.

Теорема 1.4.1. Существует такая универсальная машина тьюринга U, что для любого $x, \alpha \in \{0, 1\}^*$ $U(x, \alpha) = M_\alpha(x)$, где α - описание Машины Тьюринга M_α.

Более того, если машина M_α на входе x делала t шагов, то $U(x, \alpha)$ сделает не более $Ct \log t$, где C не зависит от x и зависит только от размера алфавита машины, числа лент и количества состояний.



▶

Наша машина тьюринга U будет иметь входную ленту для x и использовать ее так же, как и наша прошлая машину. Ленту, где описана машина Тьюринга M и ленту, которая описывает текущее состояние машины тьюринга M, а так же выходная лента.

Пусть k количество лент в машине Тьюринга M, и алфавит Σ, тогда алфавит универсальной машины тьюринга будет порядка Σ^k и мы всегда умеем его сводить в алфавиту $\{0, 1\}$.

Дальше мы сталкиваемся с проблемой, что машина M имела k движущихся головок, а мы имеем только одну.

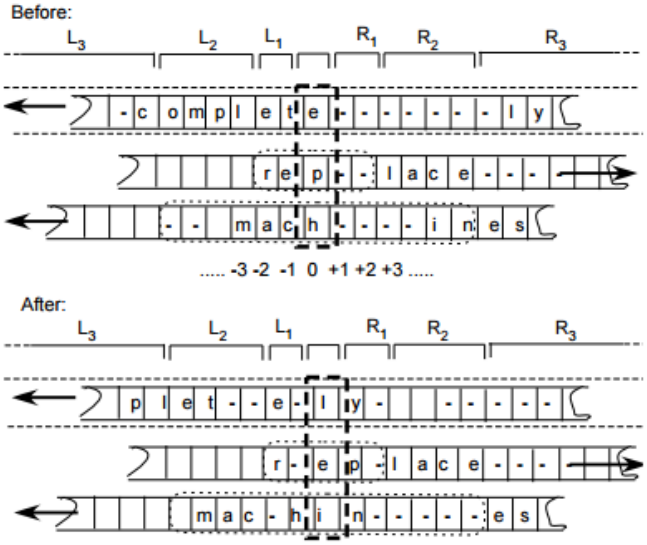
Здесь можно воспользоваться принципом "Если гора не идет к Мухамеду, то Мухамед идет к горе".

То есть, вместо того, что бы двигать головку, будем считать, что головка всегда стоит на одном месте, а мы двигаем ленту. В силу нашего большого алфавита, можем считать, что под головкой сразу много лент параллельно.

Например, если раньше было три ленты и головки сдвигались налево, направо и налево, то теперь сдвигаются ленты направо, налево и направо.

Но с точки зрения машины за $T \log T$ это все не очень хорошо, потому что при каждом сдвиге нам совсем все придется двигать, а такие большие передвижения уже будут работать за квадрат, что не очень хорошо.

Поэтому у нас есть план создать некоторый буфер свободного пространства, что бы лентам не приходилось далеко двигаться, а они могли использовать буфер.



Теперь у ячеек будут слудующие номера $\cdots -3, -2, -1, 0, 1, 2, 3, \cdots$

Теперь разделим все на блоки $L_0, R_0, L_1, R_1, \cdots$. R_i содержит 2^i ячеек с номерами $[2^i \cdots 2^{i+1} - 1]$
А L_i содержит 2^i ячеек с номерами $[-2^{i+1} + 1 \cdots -2^i]$

Головка находится вне блоков на позиции 0.
Далее будем подерживать следующие инварианты:

1. Каждая зона либо пустая, либо полная, либо наполовину полная.
2. Количество элементов в зонах L_i и R_i суммарно 2^i.
3. 0 ячейка не содержит символов пропуска.

Давай-те теперь опишем, как машина будет делать левый сдвиг, из этого все станет понятно.

1. U находит минимальный i, такой что, блок R_i не пустой. Это же минимальный i, такой, что блок L_i не полный.
2. U устанавливает первый не пустой символ в 0 позицию и Сдвигаем оставшиеся $2^{i-1} - 1$ элемент в рание блоки так, что бы все блоки были заполнены наполовину.
   Для этого как раз нужно $\sum_{j=1}^{i-1} 2^{j-1} = 2^{i-1} - 1$ элемент.
3. Делаем симметричную операцию слева, что бы подержать инвариант. То есть $2^{i-1} - 1$ левых элементов сдвигаем в секцию L_i, что бы в каждом блоке $L_0, \cdots L_{i-1}$ осталось только половина элементов.
4. Заметим, что инвариант не нарушен и зоны с номерами меньше i наполовину полные.

Что бы сделать одну такую операвцию требуется $O(2^i)$ действий. Но, после этого что бы добраться до зоны i нам потребуется совершить 2^{i-1} операций с более маленькими блоками как минимум. Нам нужно, что бы все элементы из мелких блоков переехали в какую-то одну сторону.
То есть, если мы проделали T шагов,то обратимся к i-ому блоку мы $\frac{T}{2^{i-1}}$ раз максимум, значит время работы
$O(sum_{i=1}^{logt+1} \frac{T}{2^{i-1}} 2^i) = O(t \log t)$ ◀

1.4.2. oblivious универсальная машина тьюринга

Def 1.4.2. Oblivios машина тьюрига(пофигистическая)

Положение головки на машине тьюринга не зависит от входа, а только от текущего шага.

Теорема 1.4.2. Машину построенную в пердыдущем разделе можно сделать пофигистичной.

▶ Как-то получается так,что в ближайщем к выходу ячейке может нарушаться инвариант. В целом, идея по переделыванию такая. Мы знаем где максимум можем находиться в этот момент времени. Давай-те дойдем до предела на лева и до предела на права. Стырим символ только из нужного места. Просто немного больше прогуляемся. В конце ребалансировку сделаем того блока, который сейчас на очереде в любом случае. ◀

1.5. Эквивалентность задач поиска и распознования для NP-полных языков

Теорема 1.5.1. Пусть есть массовая задача $R \in \widetilde{NP}$, причём $L(R)$ — NP-полон. Пусть также есть алгоритм A для разрешения языка $L(R)$. Тогда имеется алгоритм B для решения задачи поиска R, работающий не более, чем в полином раз медленнее алгоритма A.

Замечание 1.5.1. Смысл такой: есть мы для NP-полной задачи умеем проверять существование ответа, то искать ответ мы умеем не сильно медленнее.

▶ Мы знаем, что так как $L(R) \in \mathsf{NPC}$, то к $L(R)$ можно свести задачу SAT, т.е. есть такая п.о. полиномиально вычислимая f, что для любой формулы φ:

$$\varphi \in \mathsf{SAT} \iff f(\varphi) \in L(R)$$

Значит, у нас есть алгоритм для проверки выполнимости формулы не медленнее, чем в полином раз медленнее A. Давайте теперь научимся, используя этот алгоритм, находить решение. Это просто: мы берём формулу, проверяем выполнимость. Если выполнима, то подставляем $x_1 = 0$, если формула всё ещё выполнима — мы угадали одну переменную, идём дальше. Иначе точно надо подставить $x_1 = 1$ и начать угадывать следующие переменные. Всего у нас будет не более k запусков проверки на выполнимость, если имеется k переменных.

Т.о. есть алгоритм A', который ищет решение SAT довольно быстро.

Давайте учиться по условию x для задачи R быстро искать ответ. У нас есть ДМТ M, которая по входу из условия и решения R проверяет корректность. Аналогично доказательству теоремы Кука-Левина (4 билет) можно построить булеву формулу, эмулирующую M. В неё будет зашито условие x, а переменные будут в точности задавать решение для входа, формула будет верна тогда и только тогда, когда решение действительно является решением. А для этой формулы мы можем применить алгоритм A', который быстро найдёт решение. ◀

Замечание 1.5.2. Краткая схема доказательства:

1. Сводим SAT к нашему NP-полному языку, теперь умеем быстро проверять выполнимость.
2. Угадываем значения переменных по одной, теперь мы умеем быстро искать решения для булевых формул.
3. Строим по машине, проверяющей решение R, эквивалентную булеву формулу, решением которой будут являться в точности решения исходной задачи. Мы уже умеем быстро искать решения для булевых формул, успех.

Замечание 1.5.3. NP-полнота существенна, например, про задачу поиска нетривиальных делителей FACTOR такого неизвестно. Мы знаем детерминированные полиномиальные (от длины числа) алгоритмы проверки на простоту (ABS-тест), но вот искать делители так же эффективно не умеем.

1.6. Пример языка из NP не NP-полного и не из P

В прошлый раз мы говорили про классы NP и P и возникает естественный вопрос, есть ли в классе NP какие-нибудь еще задачи, кроме задач из класса P и NP-трудных. Сейчас мы докажем, что если $P \neq NP$, то такие задачи есть. Понятно, что если $P = NP$, то такого быть не может...

Теорема 1.6.1. Если $P \neq NP$, то существует задача из NP, которая не является NP-полной и не лежит в классе P.

▶ Делать мы будем это с помощью хитрой диаганализации.

Построим новый язык K, который будет похож на выполнимаость, но мы кое-что от туда повыкидываем. Для этого нам нужно посчитать функцию f, которая будет устроена следующим образом: $f \colon \mathbb{N} \to \mathbb{N}$

Если $f(|x|)$ четная, то все хорошо, если нечетная, то мы выкидываем эту выполнимую формулу из языка K.

Другими словами язык $K = x|x \in SAT \wedge f(|x|) \vdots 2$.

Теперь как мы будем считать f:

1. за n шагов считаем функции начиная от 0 $f(0), f(1), \cdots f(i)$. Она конечно мало что успеет, но что-нибудь то успеет. Ну на этом n не успеет, на другом успеет. Она такая, неторопливая функция. Обозначим последнее вычисленное число за k.$(f(i) = k)$
   Положим $f(0) = 0$

2. работаем за n шагов.
   (a) if (k — четно) берем машину номер $\frac{k}{2}$ это нужно для того, что бы перебрать все машины, и пытаемся найти контрпример к тому, что эта машина вычисляет язык K.
   if (существует z) $M_{\frac{k}{2}}(z) \neq K(z)$ return k + 1. Видимо, стоит машинке дать поработать n шагов и выкинуть если n шагов ей не хватило.
   Очень грустная машина, искать контрпример очень долго. Машина для языка K тоже очень грустная, она решает SAT и вычисляет f рекурсивно.
   Если она не успела найти контрпример, то возращаем k.
   M_i — i-ая машина Тьюринга.
   (b) if (k — нечетно), то будем портить жизнь сводимостям, что бы задача не стала NP-полной.
   if (существует z) $K(R_{\frac{k-1}{2}}(z)) \neq SAT(z)$ return $k + 1$
   если не успели, вернули k. R это сведение.

Это очень медленно растущая функция, но если контрпримеры есть, то f увеличивается.

Теперь давай-те докажем, что K не является полиномиально разрешимым и не является $NP-$полным.

1. Пусть K полиномиально разрешим, тогда с какого-то момента f останется навсегда на какой-то четной константе, поскольку просто будет существавать машина, для котрой не найдется контрпримера, но тогда, по определению языка K, наш язык отличается от SAT, только в конечном числе точек, значит язык эквивалентен $NP-$ полному. Значит $P = NP$, но мы предположили, что это не так.
2. Если K NP — полная, то мы во втором случае никогда не найдем контр пример, то есть в какой-то момент остановимся на нечетной константе. Но тогда, по определению языка, в него могут попасть только конечное число элементов, противоречие с тем, что язык $NP-$ полный.

◀

1.7. Полиномиальная иерархия

1.7.1. Оракульная машина тьюринга

Def 1.7.1. Оракульная машина тьюринга имеет доступ к оракулу, который за один шаг дает ответ на вопрос.

Формально: состояния q_{in}, q_{out} и "фантастический переход"из q_{in} в q_{out},заменяющий содержимое третий ленты на ответ оракула.

M^B — оракульная машина M, котрый дали конкртетный оракул B. Класс NP может быть не замкнут относительно этих операций.

Теперь некотрая тонкость про оракульные классы. Когда мы пишем P^{NP}, мы имеем в виду, что язык из класса P задается машиной тьюринга и уже к ней привешивается оракул, а точнее какую-то конкретную задачу из класса NP. Можно, на самом деле считать и что задача одна, и что задач много, так как всегда можно будет самостоятельно свести к полной задаче и попросить решить ее.

Например, в будующем будет пример, когда два классы равны, но существует оракул для которого они равны.

1.7.2. Определение иерархии по времени:

Def 1.7.2. $co - C = L|\bar{L} \in C$

Пример 1.7.1. Например, $SAT \in NP$, а {всюду ложных формул} $\in co - NP$.

Замечание 1.7.1. $co - NP$ пересекается с NP по P, но может быть по чему-нибудь еще.

Def 1.7.3. Полиномиальная иерархия. $\sigma^0 P = \Pi^0 P = \Delta^0 P = P$
$\sigma^{i+1} P = NP^{\Pi^i P}$
$\Pi^{i+1} P = co - NP^{\Sigma^i P}$
$\Delta^{i+1} P = P^{\sigma^i P}$

$\Sigma^i = co - \Pi^i$

$PH = \cup_{i \geqslant 0} \Sigma^i P$

Теорема 1.7.1. $L \in \Sigma^k P \Leftrightarrow \exists$ полиномиально ограниченное отношение $R \in \Pi^{k-1} P$ такое, что $\forall x (x \in L \Leftrightarrow \exists y R((x, y)))$

▶ $\Leftarrow$ L распазнается следующей машиной с оракулом R : недетерменированно выбираем y и проверяем $R(x, y)$ с помощью оракула.

$\Rightarrow$ Индукция, что есть $L \in \Sigma^k P$, то существует R, такая что $\forall x (x \in L \Leftrightarrow \exists y R((x, y)))$.

База: $\Sigma^1 P = NP$ это определение класса NP.

Переход: Для $k - 1$ верно, доказываем для k.

$L = L(M^O)$. По определению задается оракульной недетерменнированой машиной M с оракулом $O \in \Sigma^{k-1} P$. По предположению индукции имеется полиномиально ограниченная $S \in \Pi^{k-2} P$, такое что $\forall q (q \in O \Leftrightarrow \exists w S(q, w) = 1)$.

Строим R.

$R(x, y) = 1$, если y — принимающая ветвь вычисления для нашей оракульной машины M^O, но при каждом обращение в ветке y к оракулу в случае положительного ответа y так же содержит сертификат w: $S(q, w) = 1$.

Докажем, что $R \in \Pi^{k-1}P$:
Во-первых, нужно проверить корректность всех ходов M^O, это мы делаем за полиномиальное время.
Во-вторых, должна проверить отношение $S(q,w)$, это мы можем проверять за время $\Pi^{k-2}P$.
И наконец, для всех отрицательных ответов нужно проверить, что действительно не существует такого w, что $S(q,w)=1$.
Это может быть сделано в рамках $\Pi^{k-1}P$, так как $O \in \Sigma^{k-1}P$, а это со-задача.

◀

Следствие 1.7.1.1. $L \in \Sigma^k P \Leftrightarrow \exists$ полиномиально ограниченное $(k+1)$-арное отношение $R \in P$, такое, что $\forall x(x \in L \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \cdots R(x, y_1, \cdots, y_k))$

Def 1.7.4. QFB_k состоит из замкнутых истиных формул вида

$$\exists X_1 \forall X_2 \exists X_3 \cdots X_k \varphi$$

где φ формула в КНФ или ДНФ, а $\{X_i\}_{i=1}^k$ — разбиение множества переменных этой формулы на непустые непересекающиеся подмножества.

Теорема 1.7.2. QBF_k — $\Sigma^k P$— полные задачи.

▶ Нужно свести произвольный язык из $\Sigma^k P$ к QBF_k.

Что бы это доказать, нужно воспользоваться следствием и это уже почти доказательство. k квантеров у нас уже есть, но у нас после этих квантеров написано полиномиально проверяемое отношение, а мы бы хотели, что бы была записана булева формула.
Из отношения сделать булеву формулу мы можем с помощью теоремы Кука-Левина.
$L \in \Sigma^k P \Leftrightarrow \exists$ полиномиально ограниченые $R \in P$ такое, что $\forall x(x \in L \Leftrightarrow \exists y_1 \forall y_2 \cdots R(x, y_1, \cdots, y_k))$
Считаем, что все y_i полиномиально ограничены от x.
Если последний квантор — $\exists$, то запишем R в виде булевой формулы Φ как в теореме Кука-Левина: $R(z) \Leftrightarrow \exists w \Phi(z,w)$

Итог: $\forall x(x \in L \Leftrightarrow \exists y_1 \forall y_2 \cdots \exists w(\Phi(x, y_1, \cdots, w)))$.
Если последний квантор — $\forall$, то запишем $\bar{R}$ в виде булевой формулы Ψ как в теореме Кука-Левина: $\bar{R}(z) \Leftrightarrow \exists w \Pi(z,w)$
Итог: $\forall x(x \in L \Leftrightarrow \exists y_1 \forall y_2 \cdots \forall w(\bar{\Psi}(x, y_1, \cdots, w)))$.
Здесь мы доказали, что задача Σ^k трудная, но принадлежность Σ^k очевидна. ◀

Теорема 1.7.3. Если $\Sigma^k P = \Pi^k P \Rightarrow PH = \Sigma^k P$, при $k > 0$.

▶ При $k=1$ все просто, значит $NP = coNP$, значит квантор существования можно заменить на квантр всеобщности и все, все кванторы сразу становятся одинаковыми.
Теперь на k-ом уровне более формально.
Достаточно показать, что k + 1 уровень и k совпадают, то есть, что $\Sigma^{k+1}P = \Pi^k P$.
Пусть $L \in \Sigma^{k+1}P$, то есть $L = \{x \colon \exists y R(x,y)\}$, для $R \in \Pi^k P = \Sigma^k P$.
Значит, имеется $S \in \Pi^{k-1}P$, такое что $R(x,y) \Leftrightarrow \exists z S(x,y,z)$, то есть $x \in L \Leftrightarrow \exists y \exists z S(x,y,z)$, то есть $L \in \Sigma^k P$ ◀

Следствие 1.7.3.1. Если существует PH-полная задача, то полиномиальная иерархия конечна.

▶ Так как L лежит в конкретном $\Sigma^k P$. ◀

1.8. Сводимость по Тьюрингу

Def 1.8.1. Задача A *сводится по Тьюрингу* к задаче B, если есть машина M^B решающая задачу A.

Замечание 1.8.1. Про время работы вообще ничего не говорим. Так можно сводить не только языки (т.е. задачи распознавания), но и задачи поиска.

Замечание 1.8.2. Это сведение для языков — более общий случай сведения по Карпу (там мы разрешаем вызвать оракула только один раз, только в конце, и ничего не можем делать с его ответом).

Утверждение 1.8.1. Классы P и $\widetilde{P}$ замкнуты относительно сведения по Тьюрингу, т.е. $P^P = P$ и $\widetilde{P}^{\widetilde{P}} = \widetilde{P}$

▶ Мы можем заменить все магические вызовы оракула на явный запуск оракула. Пусть машина работала t шагов, где $t \leqslant p(n)$ (n — длина входа машины). Тогда, очевидно, она не может скормить оракулу строчку длиннее t. Так как оракул работает за $q(k)$ (где k — длина *входа оракула*, а p — полином), то на каждом входе оракул будет работать не более $q(p(n))$ — тоже какой-то полином.

Если мы вместо обращения к оракулу будем явно вычислять функцию, то получим замедленее не более, чем в $q(p(n))$ раз, т.е. всё равно полином. ◀

Замечание 1.8.3. Обратите внимание, что в общем случае оракул может получать гораздо более длинный вход, чем сама машина получила на вход. Длина входа оракула ограничена только временем работы.

Замечание 1.8.4. А вот про классы NP и $\widetilde{NP}$ мы такой замкнутости не знаем.

Теорема 1.8.1. • Для любой задачи поиска R из NP:

- Существует Лёвинский оптимальный алгоритм поиска A (детерминированный) со следующим свойством:
- Для любого другого алгоритм B, решающего задачу R:
- Существует такой полином p, что:
- На любом входе x, где есть ответ:
- Время работы A не более чем $p(t)$, где t — время работы B.

▶ A будет перебирать вообще все детерминированные машины: не только полиномиальные, не только останавливающиеся, а вообще все. Это можно сделать, перебирая все конечные строки (например, для описаний, не являющимися корректными машинами, можно считать, что они имеют право вести себя как угодно). Назовём эти машины $M_0, \dots, M_n, \dots$.

Давайте будем на шаге с номером $i = 2^l(1+2k)$ моделировать k-й шаг машины M_l, то есть на нечётных шагах моделируем шаги M_1, на делящихся только на 2 (но не на 4) — шаги M_2, на делящихся на 4 — M_3 и так далее. Если какая-нибудь машина завершается, то мы смотрим на её ответ и проверяем, не является ли он ответом на задачу. Если является, то выдаём, иначе продолжаем эмуляцию.

Заметим, что если есть какой-то алгоритм поиска (а какой-то точно есть — можно просто перебрать все решения), то мы на входе x точно когда-то завершимся. Давайте предположим, что есть алгоритм B, которому соответствует машина M_a и который на входе x завершится за время t. Тогда мы обнаружим этот факт через $2^a(1+2t)$ шагов алгоритма (если не найдём ответ раньше), обозначим это число за n. Тогда заметим, что на каждом из предыдущих шагов

мы тратили константное время не более $\mathcal{O}(n)$ времени на эмуляцию очередного шага очередной МТ и не более $\mathcal{O}(p(n))$ времени на проверку возможно выданного ответа (так как ответ не может быть больше времени работы МТ, а его проверка делается за полином). Значит, всего у нас время работы до завершения оптимального алгоритма на входе x не более $n \cdot \mathcal{O}(n) \cdot \mathcal{O}(p(n))$, что есть полином от n, что есть полином от t, что и требовалось показать. ◀

Замечание 1.8.5. На входах без ответа наш алгоритм зацикливается.

Замечание 1.8.6. Это полезное знание, если у есть конечное число оптимальных алгоритмов $A_1, \dots, A_k$, причём каждый из алгоритмов быстро работает на каком-то своём множестве входов (но тормозит на других). Тогда оптимальный алгоритм будет работать на каждом выходе не более, чем в полином раз медленнее, чем *каждый* из этих алгоритмов. А так как их конечно, то мы даже можем сказать, что оптимальный алгоритм на каждом входе не сильно хуже, чем оптимальный для этого входа.

1.9. Классы, ограниченные по времени и памяти

1.9.1. Определения классов DTime и DSpace

Def 1.9.1. $DTime[f(n)] = \{L | L$принимается ДМТ, работающей время O(f(n))$\}$
$f(n)$ должна быть конструктивной по времени. То есть такой, что существует машина тьюринга, которой если дать на вход 1^n, она может вывести $1^{f(n)}$ за время $O(f(n))$.

Замечание 1.9.1.

$$P = \cup_{k \geqslant 0} DTime[n^k]$$

Def 1.9.2. $DSpace[f(n)] = \{L | L$ принимается ДМТ с памятью O(f(n))$\}$.
$f(n)$ конструктивная по памяти, то есть существует машина Тьюринга, которой дали на вход 1^n, она выдала $1^{f(n)}$ и использовала при этом $O(f(n))$ памяти.

Def 1.9.3.

$$PSPACE = \cup_{k \geqslant 0} DSpace[n^k]$$

1.9.2. Time Hierarch Theorem

Теорема 1.9.1. $DTime[n^2] \subset DTime[n^3]$

▶ Рассмотрим язык $L = \{x | < x > (x)$ не принимает за $|x|^{2.5}$ шагов $\}$
Покажем, что $L \in DTime[n^3]$. Чтобы определить, лежит ли x в L, можно просто смоделировать машину $< x >$ на $x^{2.5}$ шагов, это можно сделать за время $O(|x|^{2.5} log|x|) = O(|x|^3)$

Пусть $L \in DTime[n^2]$, пусть m - это такой номер решающий язык L за время Cn^2, что при $n > \log m$ выполняет неравенство $Cn^2 < n^{2.5}$. Но <m> не распознает L, так как дает неправильный ответ на входе m. ◀

Теорема 1.9.2. Пусть есть функция f и g конструктивные по времени функции $f(n) \log f(n) = o(g(n))$, тогда

$$DTime(f(n)) \subset DTime(g(n))$$

▶ Доказательство обобщается с предыдущего частного случая. Функцию, которую надо взять $\frac{g(x)}{\log(g(x))}$ ◀

Моделированное недетерменированных машин на недетерминированными можно производить лишь с константным замедлением. Для каждого шага будем угадывать какое именно правило применится,текущее состояние и символы под головками машин.

Для проверки этой подсказки нужно, что бы вычисление заканчивалось в принимающем состояние, описание соседних шагов согласовались, для каждой отдельной ленты проверить, что символы под головками будут такими, если верить, что на остальных все правильно.

Теорема 1.9.3. $NTime[n^2] \subset NTime[n^3]$

▶ Для начала заметим, что мы не умеем просто так инвентировать ответ в недетерменированном случае. По этому старое решение не подойдет.

Теперь для машины номер i у нас будет целый промежуток входов от n_i до n_i^*, где $n_1 = 1, n_i^* = 2^{n_i^{2.5}}, n_{i+1} = n_i^* + 1$.

Язык L будет унарным. Если $n_i \leqslant n \leqslant n_i^* - 1$, то $L(0^n)$ совпадает с результатом запуска $< i > (0^{n+1})$ на не более чем $(n+1)^{2.5}$, если не успели, значит отвергли.

$L(0^{n_i^*})$ определим противоположным, чем результат на $< i > (0^{n_i})$, запущенного на $(n_i)^{2.5}$ шагов.

Нетрудно понять, что $L \in NTime[n^3]$. Моделирванние работает за туже сложность, то есть за куб, а обращение за $2^{n_i^{2.5}}$, что в данном случае меньше, чем $(n_i^*)^3$.

Покажем, что L не лежит в $NTime[n^2]$. Пусть $L \in NTime[n^2]$, пусть m — это такой номер недетерменированной машины, решающий язык L за время Cn^2, при n > log m выполняется неравенство $Cn^2 < n^{2.5}$. Так как <m> решает L, то $< m > (0^{n_m}) = L(0^{n_m}) =< m > (0^{n_m+1}) = L(0^{n_m+1}) = \cdots = L(0^{n_m^*})$, но по построение результат должен быть противоположным, противоречие. ◀

Теорема 1.9.4. Пусть $h(n)$ — конструктивная по времени функция. f(n) = o(h(n)) и h(n + 1) = o(g(n)). Тогда $NTime[f(n)] \subset NTime[g(n)]$

1.9.3. Space Hierarchy Theorem

Теорема 1.9.5. Для любой функции $s(n)$ выполняется $DTime[s(n)] \subset DSpace[s(n)] \subset NSpace[s(n)]$.

▶ Первое включение следует из того, что за время s(n) машина не успеет поситить более чем s(n) ячеек. Второе следует из того, что детерминированная машина является частным случаем недетерменированной. ◀

Теорема 1.9.6. $DSpace[s(n)] \neq DSpace[S(n)]$, где $s(n) = o(S(n))$ и $\forall n > n_0 \colon S(n) \geqslant log(n)$.

▶ Доказывается точно так же как и для времени. Здесь log исчезает, поскольку нет проблем в моделирование машины тьюринга за столько же памяти. ◀

Теорема 1.9.7. $DSpace[loglongn] \neq DSpace[O(1)]$

Теорема 1.9.8. $\forall \varepsilon > 0 DSpace[(loglogn)^{1-\varepsilon}] = DSpace[O(1)]$

1.9.4. PSPACE-полная задача

𝔇𝔢𝔣 1.9.4. Задача QBF - задача выполнимости формулы с кванторами.

Теорема 1.9.9. QBF $PSPACE$-полная.

▶ $QBF \in PSPACE$. С полиномиальной памятью мы умеем делать совершенно любой перебор. Так как длина сертифекатов полиномиальная. Перебираем все возможные сертификаты и подставляем их в R.

Сводим $L \in PSPACE$ к QBF. L задается какое-то машиной тьюринга с полиномиальной памятью.

Построим граф $2^{p(n)}$ конфигураций машины, принимающей L. Решим задачу достижимости.
Строим $\varphi_i(c_1, c_2)$ ="существует путь из c_1 в c_2 длины $\leqslant 2^i$"

$\varphi_i(c_1, c_2) = \exists d \forall x \forall y(((x = c_1 \wedge y = d) \vee (x = d \wedge y = c_2)) \Rightarrow \varphi_{i-1}(x, y))$
$\varphi_0(c_1, c_2)$ записывается как в теореме Кука-Левина.
Длина как раз log получается от количества вершин и петлю на конец можем добавится, что бы достаточно большая степень двойки подошла.

◀

Следствие 1.9.9.1. Если $PH = PSPACE \Rightarrow$ все сломается, значит что в полиномиальной иерархии завелась полная задача.

1.10. Оракулы для равенства и не равенства P и NP

Теорема 1.10.1. (Baker, Grill, Solovay) Существуют задачи A, B такие, что

1. $P^A = NP^A$

2. $P^B \neq NP^B$

▶ 1. Рассмотрим $A = \{(M, x, 1^n) | M(x) = 1$ за $\leqslant 2^n$ шагов $\}$ Язык машин тьюринга, которые на входе x останавливаются меньше чем за 2^n шагов.
Хотим доказать, что $P^A = EXP$ и $NP^A = EXP$.
$\mathfrak{Def}$ 1.10.1. $EXP = \cup_{p(n)} DTime(2^{p(n)})$
Для NP^A можем перебрать все подсказки, за полином проверять и каждый раз моделировать, все равно получится не более экспаненты, значит $NP^A \subset EXP$.
EXP содержится в P^A, можем просто задавать вопросы оракулу.
Ну и тривиальное включение $P^A \subset NP^A$.
Из этого следует, что $P^A = NP^A$

2. Построим языки B и U_B.
U_B это такой унарный язык, проекция языка B. Если у нас есть слово длины n, то в проекции есть слово из n палочек, а если нет, то нет.
Каким бы не было B, U_B принадлежит NP^B. Просим подсказку это слово такой же длины и спрашиваем у оракула, принадлежит ли это B.
Перебираем все детерминированные оракульные машины $M_1, \cdots, M_n, \cdots$
Запускаем машину $M_n(1^n)$ и в какой-то момент она спрашивает оракула.
Если мы уже добавили в слово в словарь, то говорим да, делать нечего. Если еще не добавили, то говрим нет.
Ждем время $2^{\frac{n}{10}}$. Теперь смотрим, что она сказала.
Если она сказала да, то мы принимаем решение, что строчек такой длины в оракуле нет.

Теперь, если он сказал нет,то находим любую строчку, про которуе не знаем, что нет и отвечаем про нее да.
Если предыдущие машины спрашивали про строчки этой длины, то резерв-то все равно у нас останется, что бы выбрать строчку.

◀

Def 1.10.2. Relative это навешивание оракула. То есть когда факт не зависит от довешивания оракулов на машине.

1.11. Полиномиальные схемы

Def 1.11.1. $L \in Size[f(n)]$, если существует семейство булевых схем $\{C_n\}_{n\in\mathbb{N}}$, такие, что

1. $\forall n |C_n| \leqslant f(n)$
2. $\forall x (x \in L \Leftrightarrow C_{|x|}(x) = 1)$

Def 1.11.2. $P/poly = \cup_{k\in\mathbb{N}} Size[n^k]$

Замечание 1.11.1. $P \subset P/poly$
Можем построить схему переходов конфигураций машины тьюринга. Она окажется полиномиального размера.

Замечание 1.11.2. Обратное включение не верно. Возьмем язык из n палочек, если M_n останавливается, то выдаем нет, если не останавливается, то да. Схема тривиально, это для данного n просто константа.

Замечание 1.11.3. Если $L \in P/poly$ тогда $\exists G\colon 1^n \to C_n$, то язык принадлежит P. То есть существует Машина, которая принимает на вход n 1 и выдает схему и язык принадлежит $P/poly$, тогда язык принадлежит P.

1.11.1. Теорема Карпа-Липтона

Мы не умеем доказывать, что $NP = P$, но может мы можем доказать $NP \subset P/poly$. Конечно же нет...

Теорема 1.11.1. $NP \subset P/poly \Rightarrow PH = \Sigma^2 P$

▶ Что бы доказать эту теорему покажем, что $\Sigma^3 P$-полный язык QBF_3 лежит в $\Sigma^2 P$.
Дали значит нам схемы, и сказали, что они решают SAT. Нужно проверить, правда это или нет.
Проверки корректности схем для SAT: $C_{|G|}(G) = C_{|G[x_1=0]}(G[x_1 := 0]) \vee C_{|G[x_1=1]}(G[x_1 = 1])$ и проверка корректности для тривиальных схем.
То есть формула выполнима,если выподнима она, когда вместо первой переменной подставили 0 или если она выполнима, когда вместо переменной поставили бы 1.
Хотим формулы с тремя кванторами решить с использованием двух.
$(\exists x \forall y \exists z(F)) \in QBF_3 \Leftrightarrow \exists$ схемы $C_1, \cdots, C_{|F|} \exists x, \forall y$
$\forall G$ - булевы формулы длины $\leqslant |F|$
(семейство $\{C_i\}$ корректно для $G) \wedge C_{|F|}(F(x, y, z)) = 1$
То есть у нас есть формула с тремя кванторами, хотим записать ее с двумя. Говорим, что это то же самое, что у нас существуют схемы, существует x, что для любого y, любая булева формула является корректной для этого семейства схем и формула F является выполнимой. ◀

1.11.2. Схемная сложность Σ_2

Теорема 1.11.2. Существует булева функция $\{0,1\}^n \to \{0,1\}$, которая не вычисляется ни одной схемой размера $\frac{2^n}{10n}$

▶ Схемы размера $T(n)$ можно задать с помощью не более чем $4T(n)\log T(n)$ битов. Каждая вершинка кодируется своим номером. операцией и номерами вершин из которой ребра ведут в эту.

Значит, схем размера $\frac{2^n}{10n}$ менее, чем $2^{2^{\frac{n}{2}}}$, а всего различных булевых формул 2^{2^n} ◀

Теорема 1.11.3.

$$\forall k\colon PH \not\subseteq Size[n^k]$$

▶ Из предыдущей теоремы следует, что булевых функций схемной сложности больше, чем n^k можно найти среди функций $\{0,1\}^{(k+1)\log n} \to \{0,1\}$. То есть, существует булева функция, зависящая только от первых $(k+1)\log n$ переменных, схемная сложность которой больше, чем n^k.

Такую функцию можно задать в виде таблицы истиности размера $\mathcal{O}(n^{k+1})$.

Зададим язык L, который будет на входах каждой длины будет решаться самой первой функцией, зависящей только от первых $(k+1)\log n$ переменных, чья схемная сложность строго больше n^k.

$$x \in L$$
$$\Updownarrow$$
$$\begin{array}{l}
\exists f\colon (\quad \textit{существует } f\textit{, вычисляющая } L \textit{ на входах длины } |x| \\
\quad (\forall C\colon (\quad \textit{она не вычисляется небольшими схемами} \\
\qquad (|C| \leqslant n^k) \to (\exists y \in \{0,1\}^n\colon C(y) \neq f(y)) \\
\quad)) \\
\quad \wedge(\forall g\colon (\quad \textit{любая меньшая функция вычисляется схемами} \\
\qquad (g < f) \to (\exists C\colon (|C| \leqslant n^k) \to (\\
\qquad\quad \forall y \in \{0,1\}^n\colon C(y) = g(y) \\
\qquad)) \\
\quad)) \\
\quad \to f(x) = 1 \qquad \textit{при этом } x \textit{ принимается этой функцией} \\
)
\end{array}$$

Замечание 1.11.4. Самый внешний квантор можно заменить с $\exists$ на $\forall$, ничего не изменится, так как функция f единственна.

В этой формуле f и g выбираются среди функций из $\{0,1\}^n \to \{0,1\}$, которые зависят от первых $(k+1)\log n$ битов. Представляются эти функции с помощью таблицы истиности для первых $(k+1)\log n$ переменных, сравниваются лексикографически. Значит, все переменные под кванторами полиномиального от n размера. В получившейся формуле все кванторы можно перенести в начало, получить язык из какого-то уровня иерархии.

Таким образом, мы получили, что $L \in PH$. ◀

Следствие 1.11.3.1. Для любого k:

$$\Sigma^2 P \cap \Pi^2 P \not\subseteq Size[n^k]$$

▶ От противного. Пусть $\Sigma^2 P \cap \Pi^2 P \subseteq Size[n^k]$. Мы знаем, что $Size[n^k] \subseteq P/poly$ и что $NP = \Sigma^1 P \subseteq \Sigma^2 P \cap \Pi^2 P$. Отсюда $NP \subseteq P/poly$, отсюда по теореме Карла-Липтона происходит коллапс $PH = \Sigma^2 P$.

Так как $PH =$ co-PH, то $PH =$ co-$PH =$ co-$\Sigma^2 P = \Pi^2 P$. Значит, $\Sigma^2 P = \Pi^2 P = \Sigma^2 P \cap \Pi^2 P$. Отсюда $\Sigma^2 P \cap \Pi^2 P = PH \subseteq Size[n^k]$. Получили противоречие с предыдущей теоремой. ◀

1.12. Теорема Савича

Теорема 1.12.1. Если s(n) — конструктивная по памяти, то $NSpace[s(n)] \subset DTime[2^{O(s(n))}]$

▶ Пусть язык L решается НМТ M, которая использует $O(s(n))$ памяти. Рассмотрим работу машины M на входе x.

Псевдоконфигурация машины — это положение головок на лентах и содержимое всех лент без входной и текущее состояние. У машины M на входе x длины n может буть не более $2^{cs(n)}$ различных конфигураций. Можно рассматривать ориентированный граф конфигураций, из конфигурации K_1 есть ребро в конфигурацию K_2, если машина Тьюринга получившая на вход x может за один шаг попасть из K_1 в K_2 за один шаг. Машина M принимает x, если существует путь из начальной конфигурации в принимающую. Принимающих конфигураций может быть несколько, но можно считать, что Машина перед окончанием работы стирает все с ленты. Для задачи достижимости в графе можно воспльзоваться, например, поиском в глубину. То есть детерменированно можн смоделировать работу машины M за время $2^{O(s(n))}$. ◀

Теорема 1.12.2. Если $s(n)$ конструктивная по памяти, то $NSpace[s(n)] \subset DSpace[s(n)^2]$

▶ Рассмотрим граф конфигураций. Описание каждой конфигурации использует $O(s(n))$ битов, поэтому мы можем перебрать все конфигурации используя $O(s(n))$ памяти.

Достаточно доказать, что задачу достижимости в графе, в котором n вершин можно решить с использованием $O(\log^2(n))$ памяти. Введем предикат $s(u, v, i)$, который истинен, если из вершины u существует путь в вершину v длины не более 2^i. Чтобы выяснить, можно ли попасть из начальной вершины a в конечную вершину b достаточно вычислить предикат s(a, b, n). Что бы вычислить $s(u, v, i)$ мы будем перебирать все z и делать два рекурсивных запуска: $s(u, z, i-1)$ и $s(z, v, i-1)$. Если найдется такая вершина z, что оба запуска вернут 1, то 1, иначе 0. Теперь как правильно распределить память, что бы было $(\log n)^2$ памяти. В начале ленты всегда написаны аргументы, с которыми вычисляется придекат s. Когда алгоритм перебирает z, то он переиспользует память, для двух рекурсивных вызовов память так же переиспользуется. Глубина рекурсии $O(\log n)$, каждый уровень рекурсии добавляет к используемой памяти $O(\log n)$ битов. Итого используемой памяти $O((\log n)^2)$. ◀

1.13. Параллельные алгоритмы.

Главаня вычислительная модель для параллельных алгоритмов — это булевысхемы.

Def 1.13.1. Время — глубина схемы
Работа — количество гейтов

Если у нас есть булева схема глубины d, то за время d мы можем вычислить результат, если каждый гейт поручить своему процессуру.

Это не самое оптимальное количество процессоров, которое можно использовать. А именно, если в схеме w gates(работа) и глубина схемы d, то процессоров достаточно что-то вроде $\frac{w}{d}$, что бы не сильно увеличить время.

То есть разбиваем все на этажи, если этаж большой, то попилим еще на несколько кусков. За счет этого время, конечно, увеличится.

Теорема 1.13.1. Brent's principle

Если в схеме имеется w гейтов и схема имеет глубину d, тогда достаточно иметь $\frac{w}{d}$ процессов, что бы время работы замедлялось не более чем в константу от оптимального.

▶ Посчитаем, как изменится время работы.
Был g_i гейтов на i-ом уровне $\sum_{i=1}^{d} g_i = w$
$T = \sum_i \left\lceil \frac{g_i}{\frac{w}{d}} \right\rceil \leqslant \sum_i (\frac{g_i}{\frac{w}{d}} + 1) \leqslant d + \frac{w}{\frac{w}{d}} \leqslant 2d$
◀

Если нам выдали меньше процессоров, чем мы способны заиспользовать. Если нам выдали $p' < p$, то время работы изменится $O(d\frac{p}{p'})$, где p столько процессоров, сколько мы могли заиспользовать оптимально.

Если же нам дали больше процессоров, то время могло не увеличится, так как ну что с ними теперь делать.

1.13.1. Определение класса NC

Def 1.13.2. Семейство схем $\{C_n\}_{n\in\mathbb{N}}$ равномерно, если имеется полиномиальный алгоритм A, такое что $A(1^n) = C_n$

То есть схема порождается полиномиальным алгоритмом. Мы подаем количество входов, алгоритм нам выдает схему.

Ясно, что равномерные полиномиальные схемы задают P.

Def 1.13.3. *Logspace*-равномерно: A использует память $O(logn)$

Def 1.13.4. Глубина схемы эквивалентно времени параллельного вычисления.

Def 1.13.5. $NC^1 = \{L|$принимается семейством схем, которые logspace-равномерные и имеют глубину $O(logn)\}$
$NC^i = \{L|$принимается семейством схем, которые logspace-равномерные и имеют глубину $O(log^i n)\}$
$NC = \cup_i NC^i \subset P$

Что бы этот язык за полином посчитать, мы сначала породим схему за полиномиальное время, а потом просто по этой схеме произведем вычисление. Так как ее размер полиномиален, так как никакую другую мы выдать не успеем.

1.13.2. P-полнота

Все сведения, которые мы пока знаем — это полиномиальные сведения. Но в данном случае полиномиальные сведения недостаточно точные, поскольку с точки зрения полиномиального сведения все языки в P P-полные. Для того, что бы говорить для более слабых вычислениях нам нужны более точные сведения, а именно в данном случае мы используем сведение логарифмическое по памяти и полиномильное по времени.

Def 1.13.6. Язык P-полные, если он принадлежит P и любой другой язык из P сводится к нему за полиномиальное время и логарифмическую память.

Полиномиальное время, в целом, следует из-за логарифмической памяти.

Теорема 1.13.2. Если L — P-полный, то

1. $L \in NC \Leftrightarrow P = NC$(все параллезуется)
2. $L \in DSpace[log] \Leftrightarrow P = DSpace[log]$

▶ 1. То есть хотим доказать,что если L сводится к $L' \in NC$, то $L \in NC$.
Пусть R сведение L к L' логарифмическое по памяти и полиномиальное по времени. Не сложно заметить, что существует логарифмическая по памяти машина тьюринга R', которая принимает вход (x, i) (где i бинарное предствление числа не больше $|R(x)|$), если и только если i-ый бит числа $R(x)$ равен одному. Теперь, если мы решим задачу достижимости в графе конфигураций R' на входе (x, i) мы сможем посчитать i-ый бит в числе R(x). Мы можем решить все эти задачи параллельно за NC_2, как обычно, напрмер, с помощью возведения матрица в степень в графе конфигураций, тогда мы смогли посчитать все биты $R(x)$. А раз у нас есть схема для $R(x)$ мы можем использовать NC схема для языка L', что бы сказать, принадлежит ли x языку L. И это все происходит в NC.
2. Здесь все очевидно, если за логарифм памяти умеем сводить и второй умеем вычислять за логарифм, то более-менее понятно, что $P = DSpace[log]$.

◀

Теорема 1.13.3. P-полный язык: $CIRCUIT_EVAL = \{$(схема C, вход x)| C(x) = 1$\}$.

▶ Очевидно, этот язык лежит в P — берём и вычисляем схему.
Осталось показать, что любой язык $L \in P$ сводится к $CIRCUIT\_EVAL$. Давайте придумаем функцию сведения f. Пусть нам дали на вход слово x. Давайте слово-в-слово повторим доказательство теоремы Кука-Левина (где мы сводили задачи из NP к SAT) и построим многослойную схему, которая эмулирует работу машины Тьюринга, разрешающей язык L. Слои у нас все одинаковые, связи между ними — тоже, логарифма памяти точно хватит (логарифм памяти — это некоторое константное число int'ов, если угодно). Получим на выходе схему, у которой нет входов и которая вычисляется в единицу тогда и только тогда, когда машина принимает слово, что и требовалось. ◀

1.13.3. Примеры параллельных алгоритмов

Для дизъюнктов

Самый простой параллельный алгоритм: $\vee_{i=1}^{n} a_i$

Здесь мы можем нарисовать двоичное дерево логарефмической глубины, где в гейтох будут находится дизъюнкции, в листьях переменные.

Перемножение матриц

Перемножение матриц булевским способом. Вместо сложения у нас дизъюнкция, вместо умножения у нас конъюнкция:
$C = AB$, где $c_{ik} = \vee_{j=1}^{n} a_{ij} \wedge b_{jk}$

Для простоты считаем, что матрицы квадратные.
Каждый элемент матрицы будет считаться своей большой дизъюнкцией.
Большую дизъюнкцию можем писать как дерево, конъюкции можем тоже отдельно привести, но это всего одна операция, поэтому дерево будет всего на единичке больше.
Время $O(\log n)$ и работа $O(n^3)$

Достижимость в графе

Теперь хотим проверим, существование путей между каждыми вершинами.

Для этого можем возвести матрицу смежности в степени n. Через и и или. A^n.
На диагонале нужно изначально добавить единицы.

Понятно, что возводить мы можем за $\log n$ быстрым возведением в степень.

То есть за $O(\log^2 n)$ можем проверить достижимость.

Сложение чисел

Учимся складывать два числа за время $O(\log n)$

Давайте сначала схемой глубины $O(\log n)$ выясним, в каких разрядах будет при сложении возникать перенос. А потом отдельно выясним значение каждого разряда: это просто сумма двух разрядов исходных чисел плюс перенос (который мы уже знаем).

Давайте научимся считать функцию f от двух чисел одинаковой длины k, она будет выдавать два бита:

1. Верно ли, что при сложении чисел возникнет перенос из разряда k?
   $g_i = a_i \wedge b_i$
2. Верно ли, что при сложении этих чисел и еще одной единицы возникнет перенос из разряда k?
   $p_i = a_i \vee b_i$

Определим операцию $\odot$ как:
$(a, b) \odot (a', b') = ((a' \vee (b' \wedge a), b \wedge b')$

Эта операция обладает несколькими свойствами.

1. $(0, 0) \odot (g_1, p_1) \odot (g_2, p_2) \odot \cdots$ Если применить эту операцию i раз, то мы получаем c_i в первой компоненте.
2. Эта операция ассоциативна, из этого сразу будет следавать, что мы сможем построить дерево логарифмической глубины. Где c_i означает итоговое наличие или отсутствие переноса в i-ом разряде.

Можно убедится по индукции, что если мы применим операцию i раз, то получим c_i в первой компоненте.
$(g_i \vee (p_i \wedge c_{i-1}), *)$
Вторая компонента нужна для ассоциативности.

Давай-те теперь поймем, что $g_i \vee (p_i \wedge c_{i-1}) = c_i$. Действительно, как мог возникнуть перенос. Если в этом разрязе уже был перенос, то никуда мы от него не денемся. Или перенос мог возникнуть в случае, если перенос был бы, если мы прибавим 1 и произошел перенос к нам из предыдущего разряда.

Можем уже сейчас все тупа записать для каждого переноса. Но так делать не хочется, так как получится большая работа.

Хорошо бы еще понять, что операция будет ассоциативна. $((a, b) \odot (c, d)) \odot (e, f) = (c \vee (d \wedge a), b \wedge d) \odot (e, f) = (e \vee (f \wedge (c \vee (d \wedge a))), b \wedge d \wedge f)$
$(a, b) \odot ((c, d) \odot (e, f)) = (a, b) \odot (e \vee (f \wedge c), f \wedge d) = ((e \vee (f \wedge c)) \vee ((f \wedge d) \wedge a), b \wedge d \wedge f)$
$(e \vee (f \wedge (c \vee (d \wedge a))) = (e \vee (f \wedge c) \vee (f \wedge d \wedge a)))$

Тогда давай-те разбиваем по парам, запускаем рекурсивно, посчитали все для четных префиксов. Теперь добавим нечетные, получили еще один этаж, тогда работа это $W(n) = W(\frac{n}{2}) + \frac{n}{2}$, а время $t(n) = O(\log(n))$. Работа линия.

Умножение

Что бы умножить два числа в столбик, мы должны просуммировать n чисел. Которые являются либо 0 либо сдвигами числа.

Нужно производить сложение n n битных чисел.

Будем производить сложения $3 - in - 2$. Мы делаем это совсем быстро, так как нам не нужно никуда ничего переносить. Он строится просто: $a = x \oplus y \oplus z$, а b — это маска возникших переносов при суммировании $x + y + z$ (т.е. просто $((x\&y)|(x\&z)|(y\&z)) << 1$).

Два числа можем сложить честно.

1.13.4. $NC^1 < NSpace[logn] < NSpace[logn] < NC^2$

Замечание 1.13.1. В случае недетерминированости по памяти в ленте с подсказкой нельзя бегать туда-сюда, можно только в одну сторону.

Замечание 1.13.2. Что такое вычисление ограниченное по памяти, которое не просто говорит да/нет, а что-то еще выдает. Это значит, что у него есть лента, которая доступна только на запись. Можно считать, что каждый бит можем записать только один раз. Выход может быть длинее ограничения.

Как комбинировать два logspace в одно. (см. практику)

Теорема 1.13.4. $NC^1 \subset DSpace(log) \subset NSpace(log) \subset NC^2$

▶ 1. Второе включение очевидно.

2. Докажем последнее включение, т.е. что $NSPACE(\log) \subseteq NC^2$.
Рассмотрим язык, лежащий в $NSPACE(\log)$, и недетерминированную машину Тьюринга M, которая его принимает.

Def 1.13.7. Псевдоконфигурацией в этой теореме назовем конфигурацию, в которую не включается входная лента.

Рассмотрим граф всех возможных псевдоконфигураций машины M (наличие каждого из ребер в этом графе, очевидно, зависит от входа).
Наша задача — проверить, есть ли (для данного входа) в этом графе путь из начальной псевдоконфигурации в принимающую. Заметим, что размер псевдоконфигурации ограничен $O(\log)$. Таким образом, нам надо решить задачу достижимости в графе, содержащем полиномиальное число вершин (обозначим это число через k).
Покажем, что эта задача лежит в NC^2.
Пусть A — матрица смежности нашего ориентированного графа. Наша задача — вычислить булеву степень[1] A^k схемами глубины $\log^2 k$.
Заметим, что для этого достаточно $\log k$ последовательных умножений пар матриц: A^2, $(A^2)^2$, ...(если получим в итоге чуть бóльшую степень, чем k — а именно, ближайшую к k сверху степень двойки — не страшно: $A^k = A^{k+1} = ...$). Для умножения же одной пары матриц $B \cdot C$ достаточно глубины схем $O(\log k)$: параллельно вычислим все произведения вида $b_{il} \wedge c_{lj}$, затем вычислим их суммы $\bigvee_{l=1}^{k} b_{il} \wedge c_{lj}$ также параллельно, затрачивая на каждую сумму лишь логарифмическое время (глубину) — сначала складываем слагаемые по два, затем полученные частичные сумму — снова по два, и т д.
Итак, мы доказали, что достижимость в графе можно выяснить в NC^2. На вход этой схеме мы должны подать элементы матрицы смежности, т.е. ребра графа (1 = ребро есть, 0 = ребра нет). Понятно, что для любых двух псевдоконфигураций мы можем в NC^2

[1] Булево произведение матриц — это произведение, в котором в качестве операции сложения используется дизъюнкция, а качестве операции умножения — конъюнкция.

выяснить, получена ли одна из них из другой (входом для этой схемы являются биты входной ленты! псевдоконфигурации входами не являются: для каждой интересной нам пары псевдоконфигураций мы строим свою подсхему).

3. Докажем теперь, что $NC^1 \subseteq DSPACE(\log)$. Пусть есть теперь язык $L \in NC^1$. Докажем, что $L \in DSPACE(\log)$. Пусть на вход нам подали x. Надо выяснить, верно ли, что $x \in L$. Мы сделаем это, построив композицию трех функций, вычисляемых с логарифмической памятью.

   Первая функция строит схему для входов длины той же, что у x. Существование соответствующей логарифмической по памяти машины Тьюринга следует из определения NC^1.

   Вторая функция преобразует эту схему в формулу (т.е. делает из ориентированного ациклического графа, представляющего схему, дерево). Сделаем мы это следующим образом: каждый гейт формулы будет обозначаться битовой строчкой, обозначающей путь к некоторому гейту исходной схемы от ее выходного гейта. (В этой строчке 1 будет означать, что мы пошли направо, а 0 — налево.) Понятно, что перечислить эти вершины, их типы $(\vee, \wedge, \neg)$ и ребра между ними мы можем, ограничившись логарифмической памятью, устроив в исходной схеме поиск в глубину (по ребрам, обратным заданным) (правда, чтобы вернуться к предыдущей вершине, придется пройтись еще раз от корня до нее, поскольку мы можем хранить только «куда мы пошли» — налево или направо — но не «откуда пришли»).

   Третья функция вычисляет значение формулы на входе x. Для этого будем обходить дерево и вычислять значение левого поддерева, а, когда надо, и правого. «Когда надо» здесь означает следующее: если вершина помечена «$\vee$», то правое поддерево надо вычислять только в том случае, если значение в левом поддереве 0, а если помечена «$\wedge$», то надо вычислять только в том случае, если значение в левом поддереве 1.

   То, что для этого достаточно логарифмической памяти, очевидно. Вот, впрочем, формальное доказательство.

   Заметим, что нам достаточно хранить только битовую строчку, обозначающую текущий гейт c и переменную b, принимающую одно из пяти значений:

   $b = 1$, если мы приступили к обработке данной вершины,

   $b = 2$, если уже обработали левое поддерево с результатом 0,

   $b = 3$, если уже обработали левое поддерево с результатом 1,

   $b = 4$, если уже обработали правое поддерево с результом 0,

   $b = 5$, если уже обработали правое поддерево с результом 1.

   Начинаем мы с пустой строки c (соответствующей выходному гейту) и $b = 1$. Опишем работу алгоритма на гейте, вычисляющем «$\wedge$» (для «$\vee$» — аналогично).

   Если $b = 1$, то $c := c\,0$, $b := 1$;

   если $b = 2$, то «обрезаем» последний бит c, и

   если этот бит был 0, то $b := 2$, а

   если этот бит был 1, то $b := 4$;

   если $b = 3$, то $c := c1, b := 1$;

   если $b = 4$, то поступаем аналогично случаю $b = 2$;

   если $b = 5$, то «обрезаем» последний бит c и

   если этот бит был 0, то $b := 3$, а

   если этот бит был 1, то $b := 5$.

   Понятно, что памяти мы затратили снова лишь $O(\log)$ на хранение c и b.

◀

Следствие 1.13.4.1. $PSpace = NPSpace$

Замечание 1.13.3. Часть конспекта взята из: http://logic.pdmi.ras.ru/h̃irsch/students/complexity1/

Недетерменизм по памяти это не страшно, за недетерменизм по памяти мы платим только квадрат. Если нам дали ориентированный граф и спрашивают, если путь из одной вершины в другую. Как это сделать за логарифм.

Если есть путь значит, есть какая-та вершина по дороге, до которой есть путь половинной длины и от нее есть путь половинной длины. У нас есть задание на вершине стека если мы взяли ABK. Нужно перебирая все промежуточные вершины обработать задания $AC(K/2)$ и $CB(K/2)$. Размер этого стека никогда не будет привышать логарифм.

Теперь мы посмотрим на граф состояний и посмотрим на наличие путей. То есть мы поняли, что если речь идем про полиномиальную память, то ничего в недетерменизме не меняется.

1.14. Вероятностные алгоритмы

Вероятностный алгоритм — это алгоритм, который может использовать случайные числа. Случайные числа со входом никак не свзяны и у алгоритма есть отдельный источник.

Источник выдает равновероятно 0 и 1, хотя, в целом, можно использовать и не равновероятные варианты, но мы этим заниматься не будем.

Случайные биты можно понимать как дополнительную ленту Машины Тьюринга или как подсказку.

Алгоритмы бывают с одностороней ошибкой, двустороней ошибкой и без ошибки вовсе. Если алгоритм работает без ошибки, это значит, что случайные биты только помогают сэкономить время. Одностороняя ошибка, это когда один ответ дается с полной ясностью, а второй ответ можеть быть не верным.

1.14.1. Определение классов

RP

Сделаем определение класса RP из определения класса NP.

Def 1.14.1. $L \in NP$, если имеется п.о. п.п R, такая, что $\forall x \in \{0,1\}^*$

1. $x \not\in L \Rightarrow \forall w (x,w) \not\in R$
2. $x \in L \Rightarrow \exists w (x,w) \in R$

Класс RP отличается тем, что он требует, что бы хороших ответов было много.

Def 1.14.2. $L \in RP$, если имеется п.о. п.п R, такая, что $\forall x \in \{0,1\}^*$

1. $x \not\in L \Rightarrow \forall w (x,w) \not\in R$
2. $x \in L \Rightarrow \frac{|\{w|(x,w)\in R\}|}{\{\text{всех } w\}} > \frac{1}{2}$

В целом, мы всегда можем уменьшить вероятность ошибки. Если мы ошибались с вероятносью $\frac{1}{2}$, то запустим алгоритм k раз и вероятность ошибки уже будет $\frac{1}{2^k}$. То есть, если все запуски алгоритма вернули 0, то выдать 0, а если хоть какой-то алгоритм выдал 1, то сразу с увереностью выдать 1.

На самом деле любой вероятностный алгоритм легко сделать с ошибкой с совсем маленкой вероятностью.

BPP

Def 1.14.3. $L \in BPP$, если п.о п.п R, такая, что $\forall x \in \{0,1\}^*$

1. $x \not\in L \Rightarrow \frac{|\{w|(x,w)\in R\}|}{\{\text{всех } w\}} < \frac{1}{3}$
2. $x \in L \Rightarrow \frac{|\{w|(x,w)\in R\}|}{\{\text{всех } w\}} > \frac{2}{3}$

$\frac{1}{3}$ и $\frac{2}{3}$ это произвольные константы. Можем взять любые две отличимые. Здесь уменьшить вероятность ошибку чуть-чуть сложнее.

Повтрим алгоритм много раз. Он нам выдаст кучу разных ответов, нужно взять по большинству и это выдать. И что бы понять, с какой вероятностью большинство будет неправильным нам потребуется неравенство Чернова.

Случаная величина в данном случае это вероятность выдать правильный ответ. После k запусков матожидание правильных ответов $\frac{2}{3}k$. Нас интересует вероятность того, что количетсво неправльных ответов окажется больше половины.

Def 1.14.4. Неравенство Чернова

$PrX > (1+\varepsilon)pk < (\frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}})^{pk} \leqslant e^{-\frac{pk\varepsilon^2}{4}}$

Где $X = \sum_{i=1}^{k} x_i$, а x_i — независимые случайные велечины, принимающие 1 с вероятностью p и 0 с вероятностью $(1-p)$.

$Pr\{\text{ошибок более}\frac{k}{2}\} \leqslant 2^{-\omega(k)}$

Теперь в определения мы можем поставить сколь угодно малую константу и 1-сколь угодно малую константу.

Так же, мы можем сделать еще лучше, и запустить алгоритм полином раз.

PP

Теперь определим бесполезный класс.

Def 1.14.5. $L \in PP$, если имеется п.о. п.п. R, такая, что $\forall x \in \{0,1\}^*$

$x \not\in L \Rightarrow \frac{|\{w|(x,w)\in R\}|}{\{\text{всех } w\}} < \frac{1}{2}$

$x \in L \Rightarrow \frac{|\{w|(x,w)\in R\}|}{\{\text{всех } w\}} > \frac{1}{2}$

Здесь мы уже не можем уменьшить вероятность ошибки.

Здесь должна быть красивая картинка с тем, как классы вкаладываются друг в друга.

RP частный случай NP, поэтому RP живет в NP.

ZPP

Def 1.14.6. $ZPP = RP \cap co-RP$

Теперь, почему это нулевая вероятность ошибки. В данном случае получается, что у нас язык такой, что для него есть два алгоритма. Из RP и $co-RP$. Алгоритм из RP нам точно отвечает для слов не из языка. А $co-RP$ ошибается, но в другую сторону. Соответственно, нужно запустить оба алгоритма. Если алгоритм из RP сказал 1, то все, значит точно 1, а если алгоритм из $co-RP$ дал 0, значит точно ноль. Соответственно, нужно подождать, пока один из них не даст ответ, в котором он уверен.

Как же долго будет происходить этот процесс? Теоретически, он может, конечно, вечно работать. Но можем оценить матожидание времени работы.

Матожидание количества запусков, соответственно, константа. $\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \cdots$

Теперь наоборот, пусть у нас есть алгоритм, матожидание времени работы которого полиномиально, тогда мы хотим убедиться, что есть два алгоритма, один из RP, другой из $co - RP$.

Запускаем наш алгоритм и ждем, когда он отработает и ставим будильник. Даем ему работать время kEt. Ошибиться он может только, если не успел отработать. Вероятность этого мы можем оценить с помощью неравенства Маркова.

$NP \subset PP$

Теорема 1.14.1. $NP \subset PP$

▶ Приведем программу q с ограничениями класса PP, которая решает $L \in NP$. Пусть функция infair_coin() моделирует нечестную монетку, которая возвращает ноль с вероятностью $\frac{1}{2} + \varepsilon$, а 1 с вероятностью $\frac{1}{2} - \varepsilon$.

Далее программа работает следующим образом. Генерирует случаный сертификат, если он подошел возвращает 1, иначе то, что вернула функция infair_coin().

Теперь посмотрим на вероятности, если $x \notin L$, то сертификат не пдошел и мы вернем правильный ответ с вероятностью $\frac{1}{2} + \varepsilon$.

Если $x \in L$, то есть хотя бы один сертификат, значит вероятность в него попасть хотя бы $\frac{1}{2^{p(n)}}$.

Значит, вероятность выдать правильный ответ

$\frac{1}{2^{p(n)}} + (1 - \frac{1}{2^{p(n)}})(\frac{1}{2} - \varepsilon) > \frac{1}{2}$

$\frac{1}{2^{p(n)}} + (\frac{1}{2} - \varepsilon) - \frac{1}{2^{p(n)}}(\frac{1}{2} - \varepsilon) > \frac{1}{2}$

$\frac{1}{2^{p(n)}} - \varepsilon - \frac{1}{2^{p(n)}}(\frac{1}{2} - \varepsilon) > 0$

$\frac{1}{2^{p(n)}} - \frac{1}{2^{p(n)}}\frac{1}{2} > (1 - \frac{1}{2^{p(n)}})\varepsilon$

$\varepsilon < ...$ Нам подойдет.

Таким образом, построили алгоритм из PP. ◀

1.14.2. Лемма Шварца-Циппеля

Приведем пример задачи, которую мы не умеем быстро решать за полином детерминировано, зато хорошо умеем решать вероятностно.

А именно. Дан многочлен, нужно проверить, что он нулевой.

Для этого нам достатчно подставить в многочлен от n переменных n точек, с хорошей вероятностью результат будет не нулевой.

Лемма 1.14.1. Sehwartr-Zippel

$S \subset F, \deg Q \leqslant d$

$\Pr\{Q(r_1, \cdots, r_n) = 0\} \leqslant \frac{d}{|S|}$

$r_1, \cdots, r_n \in S$

▶ Докажем по индукции по числу переменных:

По предположению многочлен не нулевой. $Q(x_1, \cdots, x_n) = x_1^k q_k(x_2, \cdots, x_n) + x_1^{k-1} q_{k-1}(x_2, \cdots, x_n) + \cdots + q_0(x_2, \cdots, x_n)$

С какой вероятностью здесь написан не ноль? $\Pr\{Q(r_1, \cdots, r_n) = 0\} \leqslant \Pr\{q_k(x_2, \cdots, x_n) = 0 \vee Q[x_2, \cdots, x_n](x_1) = 0\} =$

$\Pr\{q_k(x_2, \cdots, x_n) = 0\} + \Pr\{Q[x_2, \cdots, x_n](x_1) \wedge q_k(x_2, \cdots, x_n) \neq 0\}$

$q_k(x_2, \cdots, x_n)$ не ноль с вероятностью $\leqslant \frac{d-k}{|S|}$ по предположению индукции. Теперь, мы получили многочлен от одной переменной. Подставили что-то в нее. Многочлен станет ноль с вероятностью $\frac{k}{|S|}$.

Что бы был ноль, нужно что бы случилось хоты бы одно из этих двух событий, тогда итоговая вероятность $\frac{d}{|S|}$. ◀

Эту лемму надо запомнить надолго, она у нас еще появится.

1.14.3. Связь BPP и P/poly

Теорема 1.14.2. *$BPP \subset P/poly$*

▶ В BPP много хороших подсказок.

Что бы доказать, что язык принадлежит $P/poly$, нам достаточно для всех языков данной длины выбрать одну хорошую подсказку и зашить ее в схему.

Для каждого входо негодных подсказок $\frac{1}{2^{poly}}$. Возьмем в качестве этого poly просто 2n, тогда для каждого входа плохих подсказок $\frac{1}{4^n}$. При этом всего входов данной длины 2^n. Значит суммарно плохих подсказок, которые хоть где-то врут, меньше, чем $\frac{1}{4^n}2^n << 1$.

Значит куча подсказок являются годными для всех. Эту подсказку мы и возьмем и зашъем в схему. ◀

1.14.4. BPP относительно плиномиальное иерархии

Теорема 1.14.3. *$BPP \subset \Sigma_2 \cap \Pi_2$*

▶ Класс BPP замкнут относительно дополнения, поэтому, если BPP лежит в Σ_2, то он сразу лежит и в Π_2.

Давай-те рисовать формулу с двумя кванторами для языка из BPP. Мы опять будем каким-нибудь образом открывать множество всех подсказок, но чуть по-другому.

Хотим показать, что множество подсказок, которые приводят к ответу 1 велико, не правильному ответу, а именно один, если это так, то мы знаем из определения, что правильный ответ 1. Что бы показать, что оно велико, то мы наделаем его копии и покажем, что мы смогли покрыть все множество подсказок. Умерино размножить, множетсво подсказок.

Размножать мы будем с помощью xor.

Возьмем какую-нибудь строчку t_i и ее приксоим к каждой строчке-подсказке, тем самым мы получим новое множество подсказок.

Вот такими копиями мы хотим покрыть все пространство подсказок.

$r \in A_x \oplus t_i \Leftrightarrow R(x, r \oplus t_i) = 1$

$r \not\in A_x \oplus t_i \Leftrightarrow R(x, r \oplus t_i) = 0$

$\exists \{t_i\}_{i=1}^k \forall r \in U \vee_{i=1}^k (r \in A_x \oplus t_i)$

Надо теперь как-то понять, что вот именно такие копии нам подойдет.

Возьмем t_i случайным образом и посчитаем вероятность неприятности. $Pr(\exists r \in U \wedge_{i=1}^k (r \not\in A_x \oplus t_i)) \leqslant$

$\sum_{r \in U} Pr(\wedge_{i=1}^k r \not\in A_x \oplus t_i) = \sum_{r \in U} \prod_{i=1}^k Pr\{r \not\in A_x \oplus t_i\}$

$\leqslant \frac{1}{2^{nk}} 2^{p(n)} < 1$

Если вероятность меньше одного, то из этого следует, что существует набор подсказок, для которого для каждого r найдется соответствующее множество. ◀

1.15. Интерактивные доказательства

1.15.1. Оперделения классов MA и IP

У нас будут интерактивные протоколы такого рода. У нас будет Prover и Verifier. Prover будет что-то там доказывать, а Verifier будет проверять. Иногда их называют Мерлин и Артур.

Они оба видят вход. Prover всесильный, он может даже вычислять не вычислимые задачи, а verifier он скромный, он ограничен полиномиальным временем и он использует случайные числа.

Случайные числа могут быть секретные, а могут быть публичные. Секретные, это те, которые prover не видет. А публичные видят оба. Соответственно public coin(MA), private coin(PV).

Проводится сколько-то раундов MAM, AM...

В конце раундов A должен сказать правда это или нет. И мы хотим, что бы он говорил, что эта правда с достаточно хорошей вероятностью.

Def 1.15.1. Язык $L \in MA$, если имеются такие полиномы p и q и полиномиальная ДМТ A, что $\forall x \in \{0,1\}^*$

1. $x \in L \Rightarrow \exists y \in \{0,1\}^{p|x|}$: $Pr_{Z \in \{0,1\}^{q(|x|)}}\{A(x,y,z)=1\} > \frac{3}{4}$

2. $x \not\in L \Rightarrow \forall y \in \{0,1\}^{p|x|}$: $Pr_{Z \in \{0,1\}^{q(|x|)}}\{A(x,y,z) < \frac{1}{4}\} = 1$

Def 1.15.2. Язык $L \in AM$, если имеются такие полиномы p и q и полиномиальная ДМТ A, что $\forall x \in \{0,1\}^*$

1. $x \in L \Rightarrow Pr_{Z \in \{0,1\}^{q(|x|)}}\{\exists y \in \{0,1\}^{p|x|}$: $\{A(x,y,z)=1\} > \frac{3}{4}$

2. $x \in L \Rightarrow Pr_{Z \in \{0,1\}^{q(|x|)}}\{\exists y \in \{0,1\}^{p|x|}$: $\{A(x,y,z)=1\} < \frac{1}{4}$

Def 1.15.3. Язык $L \in IP$, если имеются prover(функция) P и verifier (полиномиальная вероятностная MT)V, такая, что $\forall x \in \{0,1\}^*$

1. $x \in L \Rightarrow Pr\{V^P(x)=1\} = 1$

2. $x \not\in L \Rightarrow \forall P' Pr\{V^{P'}(x)=1\} < \frac{1}{4}$

1.15.2. Неизоморфизм графов

Мерлин доказывает, что $G_0 \not\sim G_1$.

Артур берет или первый либо второй и применяет к нему случаную перестановку. Если эти два графа неизоморфны, то Мерлин правильно ответит, он всесильный.

А теперь, если изоморфны, и он не может их различить и угадает ответ с вероятностью $\frac{1}{2}$.

Замечание 1.15.1. Правда жизни состоит в том, что если у нас константное число раундов, то мы всегда можем сократить их до двух, но в порядке сначала Арутур, потом Мерлин. И добавив два лишних раунда можем избавится от приватности случайных чисел.

$NP \subset MA \subset AM$. Они чуть-чуть выше NP.

Если обменов не константа, а полином, то мы сразу получаем $PSPACE$.

1.15.3. Класс функций $\#P$

Сегодня мы поговорим об интерактиынх протоколах. Прежде чем говорить, об интерактивных протоколах я хочу вас научить еще одному классу. Он, может быть, вам и не понадбится, но в общем полезно про него знать.

Это класс не языков, это класс функций, который зовется $\#P$.

Это, в некотором смысле обобщения многих классов, что мы изучали. Почти классы, которые мы изучали формулируются в виде недетерменированных машин тьюринга:

1. NP — есть хотя бы одна единица.
2. $co-NP$ — наоборот.
3. RP — единиц много.
4. BPP — принимающий $2/3$ или отвергающий $2/3$

Все эти классы опередеряются в терминах количества принимающих состояний в Машине Тьюринга.

Def 1.15.4. $\#P = \{f | \exists NMTM : \forall x f(x) \#\{acceptingpathM(x)\}\}$

Если есть оракул, который говорит, сколько в Машине Тьюринга принимающих состояний,то мы научились решать $NP, RP, BPP, PP \cdots$.

В частности, мы научились решать любой класс полиномиальной иерархии, но это сложная теорема, ее мы не будем доказывать.

1.15.4. Перманент матрицы

PERM - вычисление пермонента матрицы является $\#P$-полная задача.

То что эта задача $\#P$-полная мы доказывать не будем, потому что это скучно. И это даже не достойно упражнения.

Сведения между функциями это сведения по тьюрингу, то есть с помощью оракулов.

Def 1.15.5. Пермонент это как определитель, но не знакопеременный. $PermA = \sum_{\sigma \in S_n} a_{1\sigma(1)} \cdots a_{n\sigma(n)}$.

▶ Для этой задачи построим IP протокол.

Сейчас у нас в протоколе случайные биты скрытые. Прувер всесильный, а варифаер полиномиальный и в конце концов он должен сказать да или нет. Оба знают вход. Конечно, мы сказали, что хотим вычислить функцию, а не да нет. Но давай-те у нас просто первым входом прувер скажет ответ, а дальше просто быдет это доказывать.

А мы скажем, правильно он сказал или нет.

То есть с помощью протокола хотим доказать, что $permA = c$, где c - константа.

Все это будет происходить в некотором конечном поле, но не велечины два, а побольше.

Теперь как мы это будем делать. Самостоятельно вычислить пермонент мы не можем. Нет у нас времени. А вот для определителя у нас есть, например, метод Гауса.

Мы сведем сначала задачу вычисления пермонента к нескольким задачам поменьше. Мы можем разложить пермонент по какой-нибудь строчке.

Раскладываем матрицу по строке. Точнее просим это сделать прувера.

Просим прувера отправить пермоненты соответствующих подматриц.
$permA = a_{11} permA_1 + a_{12} permA_2 + \cdots + a_{1n} permA_n = c$

$permA_i = c_i$

Мы сразу проверем, что сумма сошлась.

Теперь можно проверять n тождест на единичку меньше.

Если мы дальше будем продолжать таким же образом, количество задач, которые нужно проверять, станет экспоненциально много.

Давай-те теперь научимся из двух задачек делать одну такого же размера.
Вот пусть у нас есть две задачи: $permB = b$
$permD = d$

Составим символьную матрицу $perm(Bx + D(1-x)) = p(x)$
Результат будет многочлен. Мы можем подставлять разные x получать пермоненты разных матриц.

Проверим, что результат совпадает с тем, в чем он хочет нас убедить. То есть подставим 0 и подставим 1.

Нужно проверить, что то, что прислал прувер является пермонентом. Для этого выбирем случайное значения для x. Получиться некоторая конкретная матрица и нужно проверить, что она хорошая.
$R = Br + D(1-r)$
$p(r) = \rho$

Проверим, что $permR = \rho$

Теперь весь алгоритм вместе.

Нам дали матрицу и дали скаляр и просят это дело проверить. Мы раскладываем по первой строчке и получаем кучу задачек. n штук. Потом потихоньку начинаем собирать их вместе. Взяли первые две задачи попросили у прувера многочлен. ткнули в случайную точку. Получили одну.

Потом повторили.

И того у нас будет n таких этапов и на каждом этапе нам нужно собрать меньше n матриц вместе.

Какие шансы у прувера нас обмануть? Ну это конечно зависит от размера поля. Мы тут пользуемся тем, что у ненулегого многочлена достаточно мало корней. Поэтому поле будет достаточно большого размера. В частности, если нам нужно вычислять в поле поменьше, то нужно взять расширения. С самого начала будем считать, что вычисление ведется над полем размера $\geqslant n^4$

Первый шаг мы разбиваем матрицу на много и проверяем, что сумма такая, тут он вообще не может нас обмануть. Если исходное было не верно, то хотя бы одно из них тоже будет не верно.

Он может нам прислать коэффициенты неправильного многочлена. Может оказаться, что многочлен не соответствует символьной матрице. Большая беда, если многочлены разные, но мы попали в совпадающую точку.
Шансы на это $\frac{deg(p(x)-perm(Bx-D(1-x)))}{n^4}$

Если тот многочлен прислал, то совсем все хорошо. То есть, все, что может плохого случиться, это пришлют не тот многочлен. Всего таких событий не больше n^2 за время всего протокола.
Значит вероятность ошибится не больше, чем $\frac{1}{n}$. ◀

1.15.5. $PSPACE = IP$

Теорема 1.15.1. *AdiShammir* $PSPACE \subset IP$

▶ Задача о булевых формулах с кванторами является PSPACE-полной. То есть, если мы предъявим интерактивный протокол, для задачи QBF, то мы докажем $PSPACE \subset IP$.

$IP \subset PSPACE$ Сначала докажем простое обратное включение.

По большому счету у нас есть два игрока, у одного ход существования, у другого ход вероятностный. И нам нужно самостоятельно промоделировать игру этих игроков. Переборным образом. И понять, верно ли что вероятность принять равна 1.
У нас есть полиномиальная память, которую мы будем использовать.
Перебираем все ходы verifier. Дальше перебираем все возможные случайные числа, потом все шаги prover и так далее.
В конце моделируем работу verifier. Дальше, рекурсивно раскручиваемся и считаем вероятность выграть.

$PSPACE \subset IP$ Закадируем булеву формулу в виде арифметического выражения.

1. $False \to 0$
2. $True \to 1$
3. $a \wedge b \to a \cdot b$
4. $\not a \to 1 - a$
5. $a \vee b \to 1 - (1-a)(1-b) = a \odot b$

То есть мы перешли из булевых формул в арифметические.
Если формула была истеной, то мы получим арифметически 1, если формула была ложной, то получим арифметичекси 0.
Даже это верно в любом кольце.
Теперь нам нужно что-то сделать с квантарами. Это мы делаем с помощью операция над формулами.
Мы не будем это делать явно, иначе у нас будет слишком много скобочек, но всякий раз, когда мы видем $\forall x$ мы заменяем его на $A_x P$, $\exists x$ заменяем на $E_x P$ и имеем в веду, что

1. $(A_x P)(...) = P(0, \cdots) \cdot P(1, \cdot)$
2. $(E_x P)(...) = P(0, \cdots) \odot P(1, \cdot)$

Новые арифметические операторы. С понятным смыслом, но это не значит, что мы это прям сразу будем раскрывать.
Вводим оператор линеаризации, то есть убиваем степени. $(L_x P)(x, ...) = Pmod(x^2 - x)$. Это значает что у x все степени превращаются в степень один.
Со значаниями многочлена ничего не происходит. Если мы срезали степень у нолика и единички, то значание многочлена вообще никак не поменялось.
То есть такой оператор мы можем применить, когда хотим сколько угодно раз.
Перепишем формулу следующим образом, напихае везде, где только можно операторов линеаризации по всему.

$q^{(1)}_{x_1} L_{x_1} q^{(2)}_{x_2} L_{x_1} L_{x_2} \cdots P(x_1, \cdots, x_n) = 1.$

Таким способом записана некоторая арифметическая формула. Ее значание в булевых точках совпадает со значаниями исходной булевой формулы.
Таким образом, нам достаточно, что бы прувер доказал нам, что эта формула кодирует константу 1.
Теперь мы будем подставлять не всегда 0 и 1. Мы требуем от прувера многочлен, который соответствует формуле без первого квантора, потом к этому многочлену применится оператор A и все.
$A_{x_1} q' \cdots q''' P(x_1, \cdots, x_n)$
$R(x_1, \cdots, x_n) = q' \cdots q''' P(x_1, x_2, \cdots, x_n)$

То есть получили многочлен от x_1, назовем его $s(x_1)$. Степень этого многочлена не велика, поэтому результат этого многочлена мы можем сами вычислить.

1. $q = A$
   можем проверить, что $s(0)s(1) = c$
   Далее выбираем случайные точки r_1 и проверяем рекурсивно, что $R(r_1, \cdots, x_n) = s(r_1)$

   Свели задачу к задаче с меньшем количеством кванторов. Когда кванторы закончатся у нас останется только сам многочлен и значения для всех переменых.
2. $q = E$
   Аналогично с проверкой $s(0) \odot s(1) = c$.
3. $q = L$
   Тут есть особенность в том, что кванторы удаляют перменные, а этот оператор перменные не удаляют и значение для этой переменной уже есть.
   Просим коэффициенты многочлена R после подстановки всех переменных, кроме x_i. Он присылает нам какой-то многочлен от x_i. Нужно проверить, что это правильный многочлен и нужно проверить что мнгочлен правильно линеаризуется $s(0) + (s(1) - s(0))_{r_i} = c$.
   Ответ прувера соответствует тому, что утверждал прувер раньше.
   Теперь нужно проверить, что многочлен правильный. Для этого нужно ткнуть в новую случайную точку и нужно проверить, что новый многочлен правильный.

Степень многочлена будет маленькая, поэтому попасть в корень маленькая вероятность. Шагов у нас столько, сколько кванторов, значит вероятность ошибиться на одном шаге $\leqslant \frac{d}{\text{размер поля}}$

Значит если размер поля больше d^4, то вероятность ошибки $\leqslant \frac{1}{d}$

Самая большая степень может быть, когда мы только в первый раз линеаризовали, она может быть досаточно боьшой, вплоть до размера формулы. Но больше она уже точно никогда не будет, как раз благодаря операции линеаризации.

◀

1.16. Хеширование

Сегодняшняя лекция будет про хеширование, которое применяется в разных местах теории сложности.

Нам потребуется определить семейство попарно независимых хешфункций. Мы рассматриваем функции $\{0,1\}^n \to \{0,1\}^k$, которы по n битов сделают k битов и хочется, что бы эти функции

разбрасывали строчки достаточно равномерно, при этом не хочется, что бы этих функций было сверх много, но при этом много. Нужно, что бы при взятии хешфункции случайнм образом склеивающиеся пары были равномерно распределены.

Def 1.16.1. Хочется иметь такое семейство функций H такое, что если я буду брать случайным образом из этого семейства хешфункцию($h \to random(H)$), то $\forall x, y P\{h(x) = y\} = \frac{1}{2^k}$.
Вероятность, что мы отправим конкретную строчку в конкретную другую строчку она одинаковая.

$\forall x, y, x', y', x \neq x' P\{h(x) = y, h(x') = y'\} = \frac{1}{2^{2k}}$
А события что мы две разные строчки куда-то отправили должны быть независимые.

Такое семейство называется семейством попарно независимых хешфункций.

Пусть $k = n$, тогда давай-те построим такое семейство, а из него мы семейство для всех остальных k сделаем запросто.

Семейство у нас будет проиндексировано двумя индексами. Эти индексы будут строчками длин n из поля с 2^n элементами.
$a, b \in GF(2^n)$, $\{0,1\}^n \leftrightarrow GF(2^n)$

Теорема 1.16.1. Хешфункция будет выглядить очень банальным образом, а именно: $h_{a,b}(x) = ax + b$

Мы только что привели пример семейства попрно независимых хешфункций.

▶ Давай-те посмотрим, что у нас происходит со вторым условием. Путсь $x \neq x'$ и функция x отправила в y, а x' в y'.
Тогда имеет место система:

$$\begin{cases} ax + b = y \\ ax' + b = y' \end{cases}$$

Из этой системы мы можем однозначно востановить $a = \frac{y-y'}{x-x'}$ и b

Из этого мы понимаем, что если случилась конъюнкция таких событий, то мы точно знаем a и b.
Тогда вероятность этого события какая надо.

$$P = \begin{cases} ax + b = y \\ ax' + b = y' \end{cases} = \frac{1}{2^{2n}}$$

Поскольку вероятность выбрать такое $a = \frac{1}{2^n}$, вероятность выбрать такое $b = \frac{1}{2^n}$ и выбираем мы a и b независимо.
Теперь вернемся к первому пункту. Эта функция не биекция. Нам вообще не это нужно. Зафиксировали y и x, ищем подходящую хешфункцию.
$ax + b = y$

Какое бы мы не выбрали a, $b = y - ax$, то есть b выбирается однозначно, значит, вероятность угадать b: $\frac{1}{2^n}$
◀

Теорема 1.16.2. В случе $k < n$ можно просто образать результат хешфункции до k бит и при этом нужные нам условия будут выполняться.

▶ Если $k < n$, то обрежим последние биты у хешфункции [...k..n|...]. Каждой хешфункции соответсвует 2^{n-k} прежних хешфункции и собрав по этим значениям целое, мы будем получать все тоже самое.

h(x) = yt

Функция h переводит из x в y с хвостиком t. Функция h' этот хвстик обрубает.

1. $P\{h'(x) = y\} \leqslant \sum_{|t|=n-k} P\{h(x) = yt\} = 2^{n-k}\frac{1}{2^n} = \frac{1}{2^k}$
   Вероятность явно не больше, чем сумма вероятностей по всем хвостикам, даже если события зависимы, при этом если мы знаем, что вероятность у каждой функции не больше, чем $\frac{1}{2^k}$, а сумма 1, то вероятность ровно $\frac{1}{2^k}$.
2. $P\{h'(x) = y \wedge h'(x') = y'\} = P\{\vee_{t,t'} h(x) = yt \wedge h(x') = y't'\}$
   $= \sum_{t,t'} P\{h(x) = yt \wedge h(x') = y't'\} = 2^{2(n-k)}\frac{1}{2^{2n}} = \frac{1}{2^{2k}}$

   Мы воспользовались тем, что если мы берем две разные строчки, то эти события дизъюнктны. И поэтому дизъюнкция в точности равна сумме.

◀

1.16.1. Лемма Вэлианта-Вазирани

Знаем мы про задачу выполнимости булевой формулы. Знаем, что она NP-полная и мы ее не умеем решать.

То есть мы по произвольной формуле не умеем находить выполняющий набор. $x \colon F[x] = 1$

Но вдруг, если у формулы ровно один выполняющий набор, то его найти проще. Сейчас, мы докажем, что не проще.

Теорема 1.16.3. $SAT \to Unique - SAT$
Решение задачи USAT не проще, чем решение задачи SAT.

▶ А точнее мы построим следующее вероятностное сведение: Возьмем формулу и чего-нибудь к ней допишем. Так, что если формула была невыполнимой то она и останется невыполнимой, ну просто потому что мы только больше ограничений к ней дописали.

А если формула была выполнимой, то останится ровно один выполняющий набор с большой вероятностью. А допишим мы, что случайная хешфункция из нашего универсального семейства равна 0^k. Как выбрать k еще не знаем. $F \wedge (h(x) = 0 \cdots 0)$

То есть сведение выглядит следующим образом, взяли формулу F, которая дана нам на вход, берем случайную хешфункцию h и записываем $F \wedge (h(x) = 0 \cdots 0)$, где x это переменные в формуле. Если их записать подряд и присвоить 0 и 1, то получится обычная строчка.

Понятно, что выполняющих наборов у этой формулы меньше, чем у F, хочется доказать, что их с большой вероятностью 1.

S — множество выполняющих наборов для формулы F Давай-те выбирем k так, что бы выполнялись неравенства $2^{k-2} < |S| \leqslant 2^{k-1}$.

k возьмем случайным образом, тогде с вероятностью $\frac{1}{n+1}$ мы нужное k угадаем.

Утверждается, что если мы угалали k, то с вероятностью $\frac{1}{8}$ хотя бы, если формула была выполнимая, то в новой один выполняющий набор.

Нужно оценить вероятность того, что мы не убъем все выполняющие наборы, но тем неменее убъем почти все. $P\{\text{остается ровно 1}\} \geqslant P\{\text{хотя бы 1}\} - P\{>1\}$

Давай-те оценим вероятнось каждой из этих штук. Как оценить, что остался хотя бы один выполняющий набор. Вероятность, что выжило больше одного не больше, чем чем сумма по всем парам, что оба выжили. $P\{>1\} \leqslant \sum_{x,x' \in S} P\{h(x)=0 \wedge h(x')=0\} = \binom{|S|}{2} \frac{1}{2^{2k}}$

Эту вероятнусть нужно оценить сверху. Оценим п формуле включения-исключения. $P\{\text{хотя бы 1}\} = \sum_{x\in S} P_{x\in S}\{h(x)=0\} - sum_{x,x'\in S} P\{h(x)=0 \wedge h(x')=0\} \geqslant |S|\frac{1}{2^k} - \binom{|S|}{2}\frac{1}{2^{2k}}$
$= \frac{|S|}{2^k} - |S|(|S|-1)\frac{1}{2^{2k}} \geqslant \frac{|S|}{2^k}(1-\frac{|S|-1}{2^k}) = \frac{1}{8}$
И того, наша процедура выдает формулу с одним выполняющим набором хотя бы с вероятностью $\frac{1}{8(n+1)}$.

Но можно сделать другое сведение, которое по произвольной формуле генерирует n + 1 другую формулу $F_1 \cdots F_{n+1}$, тогда с вероятностью хотя бы $\frac{1}{8}$ хотя бы одна из них имеет ровно один выполняющий набор. Это будут F для разных значений k. То есть здесь мы просто перебрали все возможные значения k.

Если где-то заведется оракул, который умеет решать USAT, тогда мы вероятностно сможем решить SAT.

Много раз повторим и вероятность ошибки потихоньку станет константной. $(1-\frac{1}{8(n+1)})^{8(n+1} \to \frac{1}{e}$

◀

1.16.2. Протокол Гольдвассер-Сипсера

При помощи хеширования оценим размер множества.

Теорема 1.16.4. Существует протокол из AM для оценки размера множества. Нас пытаются убедить в том, что мощность множества какая-та определенная. $2^{k-2} \leqslant |S| \leqslant 2^{k-1}$

▶ Артут захочит это множество захешировать в строчку длины k
У нас есть множетсво $S \subset \{0,1\}^n$ артур хочет захишировать в строчки длины k

Далее Артур будет брать случаную строчку в образе и будет говорить Мерлину, в придъявика мне прообраз это точки из множества S.

Протокол такой:

1. Артур выберает случайным публичным образом хешфункцию и Артур выбирает случайным образом y.
2. Мерлин должен найти такое x, что $x \in S$ и $h(x) = y$.
3. Артур потом детерменированно проверяет, что все верно. Артут умеет проверять, что $x \in S$.

Это основано на том, что если S гораздо меньше чем нас убеждают, то у Мерлина будут трудности, потому что можем с большой вероятностью ткнуть в такую точку y, что у нее-то никакого праобраза в S то и нет.

Лемма 1.16.1. $p = \frac{|S|}{2^k}$
Пусть $|S| \leqslant \frac{2^k}{2}$, тогда

1. $Pr_{h,y}\{\exists x \in S\colon h(x) = y\} \leqslant p$

2. $Pr_{h,y}\{\exists x \in S\colon h(x) = y\} \geqslant \frac{3p}{4}$

▶ 1. Этот пункт должен быть более-менее очевидным. Взяли множество, его с помощью функции куда-то отправили. Если множество совсем маленькое, то оно не может занимать много место среди строчек размера 2^k.

Если множество размера $p \cdot 2^k$, то после того, как мы его захешировали сюда, то получилось не более, чем $p \cdot 2^k$ строчек, а всего строчек 2^k, то есть это банальность.

Первый пункт дает нам, что если множество сильно меньше, то у Мерлина очень мало шансов найти нужный x.

2. Хочется посмотреть, что у нас происходит с другой стороны.

Временно зафиксируем y, h все еще случайно, давай-те смотреть на события $h(x) = y$. Будем так же это оценивать по формуле включения исключения. $Pr_{h,y}\{\exists x \in S\colon h(x) = y\} \geqslant \sum_{x\in S} P_h\{h(x) = y\} - \sum_{x,x',x<x'\in S} P_h\{h(x) = y, h(x') = y\}$
$\geqslant |S|\frac{1}{2^k} - \binom{|S|}{2}\frac{1}{2^{2k}} = \frac{|S|}{2^k}(1 - \frac{|S|-1}{2\cdot 2^k}) \geqslant \frac{3p}{4}$

Это значит, у Мерлина есть покрайней мере такой шанс на успех.

◀

Теперь Мерлин может убедить нас, что множество достаточно большое. Вероятность того, что он нас обманет, она p.

Честный Мерлин дает нам k, такое, что $2^{k-2} \leqslant |S| \leqslant 2^{k-1}$

Хороший Мерлин с вероятностью $\frac{3}{4}p \geqslant \frac{3}{16}$ нас убедит.

Если Мерлин пытается нас сильно обмануть, то есть $|S| \leqslant 2^{k-3}$, то вероятность того, что он нас убедит меньше, чем $\frac{1}{8}$.

Если сделать этот эксперемент много раз, то мы сможем принять правильное решение с большой вероятностью.

Если размер множество слишком большой, то мы уже решили, что в этом нас убеждать не будут. Мерлин доказывает нижнюю оценку для размера множества. ◀

Теперь как это применить на пользу народного хозяйства? Если у нас нет приватных случайных бит, а только публичные, то мы можем протакол, который пользуется приватными переделать в публичные.

Пусть У нас V придумывает секретную случайную строчку r, P дает доказательство w, V после этого проверяет v(x, r, w) = 1.

Как переделать в публичный случайные биты. Вместо P пришел M и теперь убеждает Артура с публичными битами, что V с большой вероятностью получил бы 1 в игре с P. То есть, что множество случайных строк приводящих к успеху достоточно большое.

Теперь $S = \{r\colon \exists w\colon v(x, r, w) = 1\}$. Теперь Мерлин принимает не только r, но и w. И артур проверяет, что все верно.

1.17. PCP-теорема

1.17.1. Вероятностно проверяемые доказательства

Что такое класс PCP("Probabilistically Checkable Proof")? Наше определение класса NP можно записать как существует полиномиальная машина Тьюринга V(verifier), x - вход и далее выполняются следующие свойства:

1. $x \in L \Rightarrow \exists \pi\colon V^{\pi}(x) = 1$

2. $x \not\in L \Rightarrow \forall \pi \colon V^{\pi}(x) = 0$

Где π — это сертификат.

В случае класса PCP у нас меняется несколько вещей, во-первых verifier становится вероятностным, а во-вторных мы можем обращаться к любому биту доказательству, не просматривая остальные. В случае PCP обращение к битам доказательства - ограничено. Так же стоит учитывать, что адрес ячейки имеет логарифмический размер от всего доказательства, значит в данном случае полиномиальному V может потребовать экспоненциальное доказательство.

Verifier может быть адаптивным и неадаптивным. Неадаптивный выбирает следующие место куда посмотреть основываясь только на входе и случайных битах, а адаптивный так же еще учитывает ранее считанные биты доказательства.

Мы будем далее считать, что V неадаптивный.

Def 1.17.1. Пусть у нас есть две функции $q, r \colon \mathbb{N} \to \mathbb{N}$. Мы говорим, что язык L принадлежит $PCP(r(n), q(n))$, если существует такая полиномиальная вероятностная машина V, что

Эффективность: Для каждого входа $x \in \{0,1\}^n$ и данного доказательства π. V использует на более $O(r(n))$ случайных бит и делает не более $O(q(n))$ запросов к доказательству.

Полнота: Если $x \in L$ тогда существует доказательство $\pi \in \{0,1\}^n$, такое что $P[V^{\pi}(x) = 1] = 1$. Мы будем называть π корректным доказательством.

Проность: Если $x \not\in L$, тогда для любого доказательства $\pi \in \{0,1\}^*, Pr[V^{\pi}(x) = 1] \leqslant \frac{1}{2}$

Теорема 1.17.1. $NP = PCP(O(\log n), O(1))$

1.17.2. Связь с неаппроксимируемостью

PCP теоремя часто связана со многими NP оптимизационными задачами. И хорошо работает, в случа, когда надо выдавать ответ близкий к оптимальному. Прувер рисует доказательство. [.....] В каких местах мы хотим посмотреть.

Рассмотрим на примере задачи MAX-3SAT.

Def 1.17.2. MAX-3SAT — по данной формуле надо узнать максимальное количество клозов, которые могут быть одновременно выполнены.

Задача, естественно, NP-трудная.

▶

Def 1.17.3. Опеределим $val(\varphi)$ — как доля клозов, которые могут быть выполнены. То есть для выполнимой формулы $val(\varphi) = 1$.

Def 1.17.4. Пусть $\rho \leqslant 1$. Алгоритм A является ρ-апроксимируемым если для любой формула в 3КНФ с m клозами результат работы алгоритма A не меньше, чем $\rho val(\varphi)m$.

Давай-те построим $\frac{1}{2}$ приблеженный алгоритм для $MAX - 3SAT$.

В данном алгоритме будем присваивать значения переменным поочереди. Присваимваем значение и выбирае тот вариант переменной, когда выполнилось больше клозов. После чего окончательно присваиваем это значение и забываем про эту переменную и клозы, в которых она участовала. Более-менее понятно, что при таком подходе хотя бы половина от максимального числа выполнится.

В целом, походими метадами можно добиться $\frac{7}{8}$ приближенный алгоритм. ◀

Так же существует теорема, что если есть $\frac{7}{8} + \varepsilon$ приближенное решение, для любого $\varepsilon > 0$, для $MAX - 3SAT$, тогда $P = NP$.

PCP теорема часто может показывать для многих важных задач, что известный алгоритм апроксимации является оптимальным.

1.17.3. Доказательство простой версии PCP-теоремы

Код Walsh-Hadamard

Хочется предствлять строчку в некотором другом более длинном, но зато более удобном виде.

Вместо того что бы рассматривать битовою строчку мы будем рассматривать табличку ее произведений на все другие битовые строчки.

Вместо строчку x P пишет длинную длинную строчку, в которой написаны все скалярные произведения x на все возможные строчки анологичной длины.

То есть длина этого кода для строчки длины n есть 2^n.

Чем хорош этот код, так это то, что это код исправляющий ошибки. А именно. Любые два кодовых слова находятся на расстояние по крайней мере $\frac{1}{2}$ от общего числа битов, в данном случае 2^{n-1}. Иначе говоря, если мы возьмем две разных строчки $x \neq x'$ и ткнем в случаное место кода, то с вероятность хотя бы $\frac{1}{2}$ там окажутся разные биты.

Как в этом убедится? Вот у нас есть две разные строчки.
$x \neq x'$

Мы записали два скалярных произведения, сразу перенесли все в одну час, + потому что по модулю два и хотим понять какая вероятность, что данное скалярное произведенеи окажется равно 0: $\sum(x_i + x'_i)r_i? = 0$

Мы знаем, что среди $x_i + x'_i$ есть хотя бы одна 1, поскольку строчки не равны. Давай-те тогда эту 1 вынесем и напишем отдельно. Не умоляя общности первую.

$r_1 + \sum_{i \geqslant 2}(x_i + x'_i)r_i? = 0$

Давай-те выберем сначала все остальные r_i и получили какое-то конкретное значение второй части. После этого эта штука окажется 0, если мы умудрились тыкнуть r_1 ровно в это значение, а вероятность это сделать $\frac{1}{2}$.

Нам P дал код Волше-Адомара, нам нужно убеждаться что это что-то, похожее на линейную функции и значене этой линейной функции научиться считать.

Нам дали некоторую функцию $s[i]$, которая должна была быть линейной. Потому что она должна была быть кодом Волше-Адомара некоторого вектора x. Нам должны были дать функцию $f(y) =< x, y >$. Мы не знаем какое x, поэтому не можем убедится, что прям такое неравенство, но мы хатим хотя бы убедится, что для какого-то x так, ну или $f-$ линейная. Линейная же функция так и выглядит, есть какие-то коэффициенты.

То есть мы хотим понять, что есть такая линейная функция f^*, которая похожа на функцию f. То есть расстояние между двумя функциями небольшое $\delta(f, f') \leqslant \cdots$. Расстояние между двумя функциями — это вероятность того, что мы бурем случаную точки и значение функции там будет разное.

Сначала, хотим убедится, что такая f^* есть, а второе мы хотим научиться считать эту f^* с какой-то разумной вероятностью. $f^*(y) = ?$

Вероятность того, что мы правильно вычисли в значение y зависит от наших случайных бит, а нет от того, что написано в конкретной ячейке. Тыкать в те места, в которые хотим вычислить нельзя, нас могли именно там и обмануть.

И мы знаем, что если нам дали функцию, похожую на линейную, то мы не станем вычислять другую функцию, поскольку две разных линейных функции много где отличаются.

Теперь, как мы это делаем.

Будем брать две случайные точки y и z и проверять в них линейность $f(y) + f(z) = f(y + z)$. То есть мы посмотрели в три места нашей таблицы.

Нужно доказать, что это хорошоший тест на линейность, то есть если он прошел, то наша функция близка к линейной. Если f далека от линейной, то есть находится на расстояние больше, чем δ, то тест не будет пройден с вероятностью хотя бы $\frac{\delta}{2}$. Эту лемму мы докажем чуть больше.

После этого мы этот тест много раз повторим и после этого мы будем с очень хорошей вероятнтью верить, что то, что у нас есть очень хорошая линейная функция.

Вторая вещь - это вычислить f^*, мы будем вместо того, что бы вычислять в у будем вычислять: $f^*(y) = f(r) + f(y+r)$ вместо того, что бы брать $f(y)$. Иногда, конечно, может быть ошибка, но не если f линейна. r - берем случайным образом.

Что мы так хорошо вычислим f^* мы тоже докажем отдельно.

Само доказательство

Теорема 1.17.2. $NP \subset PCP(poly(n), 1)$

▶ В нашем доказательстве мы решим NP-полную задачу с помощью $PCP(poly(n), 1)$.

Язык, который мы будем использовать, это язык $QUADEQ$ — язык систем квадратных уравнений над полем 0,1.

Пример 1.17.1. Пример слова из языка $QUADEQ$:
$u_1u_2 + u_3u_4 + u_1u_5 = 1$
$u_2u_3 + u_1u_4 = 0$
$u_1u_4 + u_3u_5 + u_3u_4 = 1$

В данном примере решением являются все 1.

Теперь приступим к конструированию доказательства используя код Волше-Адомара. Прувер пытается нас убедить, что у этой системы есть некоторое решение U.

Что бы доказать, что $QUADEQ$ NP-полный проще всего свести к этой задачи задачу о трираскраске графа.

Просили прислать код Уолше-Адомара для решения u_i и для вектора из u_iu_j.

Первым делом мы проверяем ту и другую табличку на линейность, тестом, который у нас был. Сколько раз мы должны будем ткнуть в табличку? Тест на линейность требует ткнуть в табличку 3 раза. Ну и нужно надо будет константное число раз повторить.

Надо проверить, что они к друг другу имееют отношения. Для этого надо взять случайные точки. То есть взять случайные две точки из первой таблички, составить тензорное произведенеи и ткнуть в соответствующую ячейку второй таблички.

Хотим в будующем понять, что вероятностью $\frac{1}{4}$ мы прувера за руку поймаем.

После проверки на линейность, мы обращаемся не к табличке а к линейной функции, которую получили.

Теперь проверим, что решение, которо нам дали удовлетворяет системе. Мы можем легко вычислеть значение одной строчки системы. $< (u_iu_j), (a_{ij}) >$ можем взять сколярное произведение квадратичной формы на строчку составленную из u_iu_j. Это будет $< a_{ij}, u_iu_j >= \sum a_{ij}u_iu_j$ значение квадротичной формы. Проблема в том, что мы не можем такое сделать с каждой строчкой нашей системы.

Поэтому мы сделаем следущую хитрость, мы из нашей систему составим одно уравнение случайным образом. Как это делается. Мы вычисляем сумму каких-то подстрочек нашей матрицы. То есть каждую строчку берем или не берем с вероятностью $\frac{1}{2}$. И то же самое со свободным столбцом.

В результате получаем ровно одну квадратичную форму и ровно одно для него значение. И эту форму мы уже сможем это проверить. Но все равно, мы к таблице обратимся константу

раз. Это был последний шаг, проверить, что все подходит.

Теперь вернемся к предыдущему шагу, проверке, что две строчки связаны между собой. Мы утверждаем, что поймаем с вероятноть $\frac{1}{4}$. Что происходит, когда мы делаем этот тест. Мы проверяем, что $f(r)f(r')? = g(r \otimes r')$. Оказалось, что g что-то не то. Мы хотим это поймать. Это точно линейные функции.

Если мы две разные матрицы умножили на случайною строчку, то с вероятностью $\frac{1}{2}$ мы получим разные вектора. Потом мы эти матрицы умножаем на случайный вектор справа, то получим разное с вероятностью по карйней мере $\frac{1}{2}$. В итоге результат будет разный с вероятностью хотя бы $\frac{1}{4}$.
$\sum_{i,j} u_i u_j r_i r'_j = (\sum_i u_i r_i)(\sum_j u_j r'_j) = \sum_{i,j} r_i r'_j$

Что бы улучшить вероятность нужно опять повторить все много раз.

В итоге мы в итоге все равно обращались константное число раз.

Все это в нашем рассуждение так, если не случилось не одной ошибки в вычисление нашей функции. Но мы изначально сделаем так, что бы вероятность ошибки была маленькая.

Если хотите проконтроитровать все числа, то это хорошее упражнение на экзамен :) С точностью, до нескольких лемм доказали. ◀

QUADEQ — NP-полная задача

Еще из прошлой теоремы осталось два момента. Первый, почему задача, которую мы сводим NP-полна.

Лемма 1.17.1. QUADEQ — NP-полная задача.

▶ Сведем Circuit SAT к QUADEQ. Давай-те нарисуем схему. Ее можно переписать в виде системы квадратичных уравнений.

Как? Вот есть гид, которые из x и y делает z. Понятно, что если там конъюнкция, то это произведение. Дизъюнкцию можно переписать через конъюнкцию и отричание, то есть каждый гид можно записать, как какое-то квадратичное уравнение. Тут только есть одирн маленький момент. Потому что у нас были чисто квадратичные уравнения, там не было первях степеней. Поэтому, как только видете первую степень можете сразу вместо нее записать квадрат.
$z^2 = xy$ ◀

Еще про связь с неаппроксимируемостью

Вторая вещь, оставленная безсознательная. Я вам сказал, что мотивация PCP теоремы, доказывать, что для каких-то задач нельзя построить приблеженных алгоритмов. Но я вам не сказал почему.

Лемма 1.17.2. PCP теорема, часто является доказательством не существования хорошо приближенных решений.

▶ Вот почему. Смотрите. PCP теорема позволяет сделать вот что, она позволяет из формулы сделать другую формулу. Такую, что, если формула была выполнима, то она и останется выполнимой, а если формула была невыполнимой, то она стент очень сильно невыполнимой. То есть в ней выполнить не только 100% клозов, но и даже некоторую константную долю даже нельзя.
$F_{\text{вып}} \to F'_{\text{вып}}$
$F_{\text{невып}} \to F'_{\text{сильно невыполнимо}}$

То есть некотрую фиксированную долю мы точно никогда не сможем выполнить. Некоторую конструкциями можно получить $\frac{1}{8}$. А мы умеем только совсем маленькую долю можем получить, но тем неменее фиксированная.

Теперь поймем, что этого достаточно, что бы оправдать не существование хорошего приближенного алгоритма. В самом деле если есть алгоритм, со степенью приближения больше, чем 1-доля, то мы сможем точно решать выполнимость, а именно нам дали на вход формулу, мы ее раздули до сильно невыполнимой, нам дали этот чудо алгоритм и этот чудо приближенный алгоритм выдаст больше, чем $1-\alpha$ клозов, а для невыполнимых не выдаст, потому что там столько нет.

Теперь откуда берется это сведение. Для этого и нужна PCP теорема. $PCP(O(logn), O(1)) = NP$. Нам нужно log случайных чисел и константа просмотров. И мы говорили, что это неадоптивно.

Теперь как из этого сделать другую формулу. Эта формулу будет моделировать все возможные запуски прувера в PCP. То есть перебираем все случайные биты. Всех вариантов у нас полином.

В каждом куске в каждом блоке мы пишем те дизъюнкции, которые говорят нам, в каком случае мы примем доказательство. Переменными в этой формуле будут те биты в доказательстве, которые мы будем запрашивать.

Вот мы запросили три бита доказательства, и после этого мы должны принять решения брать или не брать. Это какая-та функция. Эту функцию мы можем записать в виде КНФ от этих переменных. Она будет по прежнему константного размера.

$(\underline{(a_1 \vee a_2 \vee)} \wedge \cdots O(1)) \wedge ()$

И так, у нас теперь для каждых выпавших случайных битов $2^{c \log n}$ у нас будет константный кусок формулы. Мы получили новую формулу, и если существует выполняющий набор для нее, это значит что V у нас ее в PCP принимал, значит и ранее она была выполнимой .

Теперь, PCP теорема нам говорит, что если формула невыполнима, то она будет отвергнута с некоторой константной вероятностью. Что это значит? Это значит, что среди наших блоков константная доля будет ложна. Блок ложен, если хотя бы одна из дизъюнкций ложна, но дизъюнкций там константное число. Для примера их 100 штук, а вероятность отвергнуть $\frac{1}{2}$ Это значит, что доля ложных клозов у нас больше, чем $\frac{1}{2}\frac{1}{100}$.

◀

Оценка вероятности ошибки в проверке на линейность

Лемма 1.17.3. Если функция отличается от линейной хотя бы в доле δ точек, то мы ее улечим в этом с вероятностью $\frac{\delta}{2}$

▶ У нас две функции отличаются хотя бы в δ точках. Если мы ткнем в случайное место, то с вероятность δ будет расхождение в ней и в ближайшей к ней линейной. То есть есть f и лучшая к ней линейная f^*.

Иначе говоря, хотим доказать, если тест проходит с хорошей вероятностью, то f - делта близка к линейной.

Для $\delta < \frac{1}{3}$, если $Pr_{y,z}\{f(y+z) = f(y) + f(z) \geqslant 1 - \frac{\delta}{2}\}$, то f является δ-близкой к линейной.

Давай-те эту f^* найдем и продемонстрируем, что она δ-близка.

$$f^* = maj_r(f(x+r) + f(r))$$

Берем случайную точку r и берем результат как сумму. Естественно результаты могут разные для разных r, получаться, мы берем доминирущее значение.

Нам остается убедится в двух вещах, что f близка с f^* и что f^* действительно линейная.

Пусть $p_x = Pr_r\{f^*(x) = f(x+r) + f(r)\}$. Давай-те посмотрим на вероятность того, что мы попадаем в это самое maj.

$p_x \geqslant \frac{1}{2}$, потому что мы знаем, что там maj берется, то есть мы же знаем, что там большинство. Мы хотим показать, что она по крайней мере $1 - \delta$.

▶ В нашем распоряжение есть тест на линейность, про который мы знаем, что он с большой вероятностью проходит.

$Pr_{r,s}\{f(x+r)+f(s) \neq f(x+r+s)\} \leqslant \frac{\delta}{2}$

В этот тест на линейность можно подставлять совершенно любые случаные точки. Важно понимать, что если вы берете x и добавляете к ней случаную сточку r, то вы получаете случайную равномерно распередленную точку.

И так, проделаем следующие трюки. Посмотрим на одни и теже точки с двух сторон:

$Pr_{r,s}\{f(x+r)+f(s) \neq f(x+r+s)\} \leqslant \frac{\delta}{2}$

$Pr_{r,s}\{f(r)+f(s+x) \neq f(x+r+s)\} \leqslant \frac{\delta}{2}$

В обоих случаях тест на линейность проходится независимо с вероятностью $\frac{\delta}{2}$

Теперь, что происходит, если эти оба теста одновременно не сломались. Вероятность того, что они оба не сломались $= 1 - \delta$.

Если они оба не сломались, значит $f(x+r)+f(s) = f(x+s)+f(r)$. $1-\delta \leqslant Pr_{f(x+r)+f(s)=f(x+s)+f(r)}$

Есть две равных между собой сущности. Эти сущности - это один бит. То есть это либо ноль, либо 1. Рассмотрим эти два варианта, когда этот бит 0 и когда этот бит 1.

$= \sum_{b=0}^{1}(Pr_r(f(x+r)+f(s)) = b)^2 = p_x^2 + (1-p_x)^2 \leqslant p_x^2 + p_x(1-p_x) = p_x$

когда она 0, то это вероятность того, что $f(x+r)+f(r) = 0$ и одновременно $f(x+s)+f(r) = 0$, поскольку, r и s независимые, то нужно эти вероятности перемножить, но поскольку это просто условное обозначение переменных, то можем эту вероятность возвести в квадрате.

Одна из этих вероятностей, мы не знаем какая равне p_x. Поскольку одна из них совпадает с $f^*(x)$, а когда на равно $1 - p_x$. Остается это дело оценить, с учетом того, что p_x хотя бы $\frac{1}{2}$.

Тем самым мы доказали, что $p_x \geqslant 1 - \delta$ ◀

Значит, $Pr_r\{f^*(x) \not= f(x+r) + f(r)\} < \delta$.

Теперь нужно доказать, что функцию, которую мы получили — линейная.

У нас есть три события. $Pr_r(f^*(x) + f^*(y) + f(r)) \neq f(x+r) + f^*(y)\} < \delta$ В первом событие написана вероятность, что $f^*(x) = f(r)+f(x+r)$. Это известная нам вероятность, она больше чем $1 - \delta$

$Pr_r(f(x+r) + f^*(y) \neq f(x+y+r)) < \delta$

Здесь написано про $f^*(y)$. Взяли $f^*(y)$ и добавили к ней случаную точку, но только случаная точка выглядит как $x + r$. И это снова тоже самое, что p_y.

$Pr_r(f(x+y+r)) \neq f^*(x+y) + f(r) < \delta$ Здесь мы взяли точку $(x+y)$.

Все эти три события случаются с маленькой вероятностью. С вероятностью $1 - 3\delta$ там равенства. А если есть три равенства, то $f^*(x) + f^*(y) = f^*(x+y) > 1 - 3\delta$. Но тут уже мы ничего случайно не выбираем. А если про какое-то не вероятностное событие сказано, что его вероятность больше 0, значит оно просто верное.

Значит функция является линейной.

Теперь обсудим δ-близость. Пусть $Pr_x\{f(x) \neq f^*(x)\} > \delta$. По построение $Pr_r\{f^*(x) = f(x + r) + f(r)\} \geqslant \frac{1}{2}$, значит $Pr_{x,r}\{f(x) \neq f(x+r) + f(r)\} > \frac{\delta}{2}$. Противоречие. ◀