Лекция 2

Оптимальный алгоритм для $\widetilde{\mathbf{NP}}$-задачи. Не NP-полные задачи в $\mathbf{NP} \setminus \mathbf{P}$. Унарные и редкие языки.

(Конспект: В. Моргенштерн)

2.1 Почти оптимальный алгоритм для задачи из $\widetilde{\mathbf{NP}}$.

Так как решение любой задачи из $\widetilde{\mathbf{NP}}$ может быть проверено за полиномиальное время, то каждая задача из $\widetilde{\mathbf{NP}}$ решается некоторым алгоритмом. В худшем случае этот алгоритм просто проверяет все возможные ответы, ограниченные по длине тем полиномом, который фиксирован для данной задачи. Алгоритм обязательно остановится и выдаст правильный ответ через конечное время. Можно, однако, построить *в явном виде* некоторый “оптимальный” алгоритм, который будет решать задачу из $\widetilde{\mathbf{NP}}$ “почти столь же быстро”, как и любой другой. (В частности, если $\widetilde{\mathbf{P}} = \widetilde{\mathbf{NP}}$, то наш алгоритм будет полиномиальным.) Именно, для каждого алгоритма M наш “оптимальный” алгоритм будет тратить на входе (достаточно большой) длины n не более $Ct_M(n) + p(n)$ шагов, где $t_M(n)$ — время работы алгоритма M, константа C зависит только от задачи и алгоритма M (но не от n), а p — полином, фиксированный для всех задач и алгоритмов M сразу.

Получить в точности этот результат мы сейчас не сможем, но получим весьма близкий к нему, ограничившись в качестве модели вычисления машинами Тьюринга.

Каждую машину Тьюринга можно закодировать. Значит, их все можно пронумеровать. Каждую машину при этом слегка модифицируем: вместо того, чтобы останавливаться, машина проверяет, что найденное решение действительно удовлетворяет условию, и останавливается только если решение правильное (по определению $\widetilde{\mathbf{NP}}$ это можно сделать за полиномиальное время). Итак, у нас есть последовательность (проверяющих свой результат) машин Тьюринга $M_1, \ldots, M_l, \ldots$. Далее, пусть наша универсальная машина M на шаге с номером $i = 2^l(1+2k)$ моделирует k-ый шаг l-ой машины Тьюринга в этом списке. Если машина заканчивает работу (стало быть, она нашла правильное решение), мы тоже заканчиваем работу. Если нет, то продолжаем моделировать работу оставшихся машин. Легко проверить, что $(l,k) \neq (l',k') \Rightarrow 2^l(1+2k) \neq 2^{l'}(1+2k')$. Поэтому алгоритм работает корректно и тратит на решение количество шагов, отличающееся от времени оптимального алгоритма лишь в полином раз[1].

Упражнение 2.1. Однако, чтобы уточнить время работы, необходимо зафиксировать модель вычислений (причем желательно такую же, как и у моделируемого устройства!): заметим, что нам, например, необходимо одновременно поддерживать память каждой из моделируемых машин Тьюринга. $\square$

Замечание 2.1.

- В случае, когда решений нет, алгоритм зацикливается! Поэтому оптимален от только на входах, имеющих решение.
- Можно подумать, что из этого алгоритма можно сделать алгоритм, оптимальный на входах соответствующей задачи распознавания. Но это не так, поскольку это требует применения самосводимости (F выполнима $\iff$ $F[x]$ выполнима или $F[\neg x]$ выполнима), которая заменяет вход другим.

2.2 Задачи из NP, которые не являются NP-полными, но и не лежат в P.

Мы уже знаем, что в классе **NP** есть самые трудные — **NP**-полные, и самые простые — решаемые за полиномиальное время — задачи. Возникает резонный вопрос, есть ли в **NP** промежуточные по сложности

[1] Достичь заявленной выше оценки нам мешает необходимость передвигать головку между текущим состоянием памяти для различных моделируемых нами машин; для RAM-машин удалось бы от этого избавиться.

задачи (те, что с одной стороны не лежат в $\mathbf{P}$, а с другой — не являются $\mathbf{NP}$-полными). Ответ на этот вопрос дает следующая теорема.

Теорема 2.1. *Если* $\mathbf{P} \neq \mathbf{NP}$*, то в* $\mathbf{NP} \setminus \mathbf{P}$ *имеются задачи, не являющиеся* $\mathbf{NP}$*-полными.*

Доказательство. Мы уже знаем, что задача SAT является $\mathbf{NP}$-полной. Сейчас мы немного изменим эту задачу так, чтобы она больше не была $\mathbf{NP}$-полной, но и не попала в $\mathbf{P}$.

Для начала, каким-либо способом перенумеруем[2] все машины Тьюринга, снабженные «полиномиальным будильником»: считающие свои шаги и останавливающиеся после $p(n)$ шагов (для каждой машины полином p — свой). Получим последовательность $M_1, M_2, \ldots$. Теперь перенумеруем все полиномиальные сведения по Карпу (тоже снабженные таким «будильником») — это тоже машины, только оракульные. (Неформально, нас будут интересовать сведения SAT к нашему языку.) Получим другую последовательность $R_1, R_2, \ldots$.

Далее, пусть строка x кодирует некоторую булеву формулу. Пусть S — конкретная машина, которая принимает только выполнимые формулы (перебирая все наборы значений переменных — т.е. работающая экспоненциально долго). $S(x)$ обозначает результат работы этой машины на формуле x. Искомый язык определим так:

$$\mathcal{K} = \{x \mid S(x) = 1 \ \wedge \ f(x) \vdots 2 \},$$

где функция $f(n)$ вычисляется алгоритмом, описанным ниже. Это очень медленно растущая (но $\underset{n\to\infty}{\longrightarrow} \infty$) функция. Она будет пытаться найти полиномиальный алгоритм для SAT либо сводимость SAT $\to \mathcal{K}$, и, как только появится аргумент, при котором ей удастся найти это, с этого момента окажется константой, что будет противоречить условию теоремы (но именно это и будет вытекать из гипотезы $\mathcal{K} \in \mathbf{P}$, равно как и из гипотезы, что $\mathcal{K}$ — $\mathbf{NP}$-полный).

Обозначим K детерминированную машину, выясняющую принадлежность $\mathcal{K}$ (она легко определяется через машину S и машину, вычисляющую функцию f, и работает экспоненциально долго).

На вход машине Тьюринга, вычисляющей f, подается число n в «палочковой» системе счисления: 1^n. Эта машина делает $2n$ шагов. За первые n шагов она вычисляет последовательно $f(0), f(1), \ldots, f(i)$ — столько значений, сколько успеет (да, это рекурсивное задание функции f!).

[2]Очевидно, перечислить машины, которые всегда "случайно" останавливаются столь быстро, как нам надо, мы не сможем. Однако чтобы перечислить для каждого языка из $\mathbf{P}$ хотя бы одну принимающую его машину, достаточно перечислять машины с «полиномиальным будильником».

(Каждый шаг наша машина сдвигает указатель на ленте, где у нее записано 1^n, на одну позицию вправо — как только вход закончится, эта фаза прекращается.) Последнее вычисленное значение обозначим за k.

Вторая фаза вычислений также занимает n шагов, и будет зависеть от k.

1. Если k — четное, то алгоритм запускает машину $M_{k/2}(z)$ последовательно на всех входах в течении n шагов. Если хотя бы для одного из этих входов $M_{k/2}(z) \neq K(z)$, то возвращаем $k+1$, иначе возвращаем k.

2. Если k — нечетное, то алгоритм запускает машину $K(R_{(k-1)/2}(z))$ последовательно на всех входах в течении n шагов. Если хотя бы для одного из этих входов $K(R_{(k-1)/2}(z)) \neq S(z)$, то возвращаем $k+1$, иначе возвращаем k.

Заметим, что все рекурсивные определения корректны, так как за n шагов мы не успеваем добраться до такого z, что $K(z)$ еще не определено.

Ясно, что $\mathcal{K} \in \mathbf{NP}$. Функция f (по определению) вычисляется за полиномиальное время, а получив выполняющий набор для формулы x в качестве подсказки, мы с легкостью проверим результат.

Покажем, что $\mathcal{K} \notin \mathbf{P}$. Допустим противное. Тогда машина K совпадает с одной из машин нашего списка, т.е. $K = M_k$ для некоторого k. Тогда существует константа c и номер N_0, такой, что для любого $n \geq N_0$, $f(n) = c \leq 2k$. Действительно, функция f монотонно возрастает, а достигнув значения $2k$, алгоритм вычисления f всегда будет попадать в первый случай, получать равенство на всех значениях z и снова выдавать $2k$ на выход. Поэтому язык $\mathcal{K}$ совпадает с языком выполнимых формул, везде, кроме разве что конечного числа слов. Это означает, что задача выполнимости разрешима за полиномиальное время, т.е. $\mathbf{P} = \mathbf{NP}$. Противоречие.

Аналогичные рассуждения в предположении, что $\mathcal{K}$ — не $\mathbf{NP}$-полон, показывают, что, начиная с некоторого места, f — нечетная константа. Это, в свою очередь, означает, что язык $\mathcal{K}$ конечен. Но в таком случае, конечно, он распознаваем за полиномиальное время и $\mathbf{P} = \mathbf{NP}$. Противоречие. $\square$

2.3 Унарные и редкие языки.

Теперь мы докажем, что некоторые слишком простые типы языков не могут быть $\mathbf{NP}$-полными. Конечно, мы предполагаем, что $\mathbf{P} \neq \mathbf{NP}$.

Определение 2.1. Язык называется унарным, если все его слова состоят из одного и того же символа. (Например, язык $\{\underbrace{0\dots0}_{p} \,|\, p \in \mathbb{P}\}$.)

Теорема 2.2. *Если* $\mathbf{P} \neq \mathbf{NP}$, *то никакой унарный язык не может быть* $\mathbf{NP}$-*трудным по Карпу.*

Доказательство. Пусть U — $\mathbf{NP}$-трудный унарный язык. Докажем тогда, что задача выполнимости может быть решена за полиномиальное время.

Заметим, что для любой переменной x формула f выполнима тогда и только тогда, когда выполнима хотя бы одна из формул $f[x := true]$ или $f[x := false]$ (при присваивании дизъюнкции, содержавшие подставленное значение $true$ — оно же $\neg false$, удаляются, а из остальных удаляется подставленное значение $false$ — оно же $\neg true$). Поэтому выполняющий набор можно найти, построив дерево поиска. Каждый узел этого дерева содержит две ветви с двумя возможными подстановками очередной переменной.

Объем поиска можно уменьшить, если при поиске в ширину в этом дереве на каждом уровне приводить формулы к каноническому виду и убирать совпадающие формулы. Раз U — $\mathbf{NP}$-полна по Карпу, то существует полиномиальное сведение $g : \mathtt{SAT} \to U$. При этом если для формул $f,\ f'$ выполняется $g(f) = g(f')$, то $\mathtt{SAT}(f) = \mathtt{SAT}(f')$ и одну из этих формул можно без риска убрать из дерева поиска. Но U — унарный язык (пусть $\subseteq \{0^n | n \in \mathbb{N} \cup \{0\}\}$). Поэтому полиномиальное сведение g порождает лишь полиномиальное число различных «разумных» образов[3] подформул формулы f. Значит, начиная с некоторого места, все уровни дерева поиска содержат число формул, ограниченное этим полиномом. Такое дерево можно обойти за полиномиальное число шагов, что означает, что $\mathbf{P} = \mathbf{NP}$. Противоречие. □

Определение 2.2. Язык L называется редким, если для некоторого полинома p, $|\{x \in L \mid |x| \leq n\}| \leq p(n)$.

Теорема 2.3. *Если* $\mathbf{P} \neq \mathbf{NP}$, *то никакой редкий язык не может быть* $\mathbf{NP}$-*трудным по Карпу.*

Доказательство этой теоремы будет приведено на спец. семинаре.

Замечание 2.2. Для сводимости по Куку все гораздо сложнее. В частности, требуется более сильное предположение, чем $\mathbf{P} \neq \mathbf{NP}$.

[3] Мы можем не обращать внимание на те образы, что состоят не только из нулей — это образы невыполнимых формул!

Определение 2.3. Для любого класса $\mathcal{C}$, $\mathbf{co}\text{-}\mathcal{C} = \{L \mid \overline{L} \in \mathcal{C}\}$, где $\overline{L}$ содержит в точности те строки в данном алфавите, которые L не содержит.

Теорема 2.4. *Если $\mathbf{P} \neq \mathbf{NP}$, то никакой редкий язык не может быть $\mathbf{co}\text{-}\mathbf{NP}$-трудным по Карпу.*

Доказательство. Доказательство похоже на доказательство теоремы 2.2. Трудность будет заключаться в том, что теперь сведение может выдавать строки, состоящие не только из нулей. Тем не менее, мы знаем, что количество «разумных» образов[4] ограничено полиномом от длины входа. Следовательно, как только образов станет «слишком много», мы можем смело говорить, что какая-то из полученных (тем самым, и исходная) формула — выполнима. □

[4]На сей раз — тех, что являются образами невыполнимых формул — мы ведь сводили $\overline{\texttt{SAT}}$.