

Д.В. Мусатов

# СЛОЖНОСТЬ ВЫЧИСЛЕНИЙ

Конспект лекций

МФТИ

2017



# Оглавление

<b>1</b>	<b>Модели вычислений</b>	<b>9</b>
1.1	Виды вычислительных задач и вычислительных ресурсов . . . . .	9
1.2	Измерение сложности задачи . . . . .	13
1.2.1	Меры сложности алгоритма на данном входе . . . . .	13
1.2.2	Сложность конкретного алгоритма . . . . .	14
1.2.3	Асимптотические обозначения . . . . .	14
1.2.4	От сложности алгоритма к сложности задачи . . . . .	15
1.3	Базовая модель: многоленточная машина Тьюринга . . . . .	15
1.4	Распознавание языков . . . . .	17
1.5	Исторические замечания и рекомендации по литературе . . . . .	19
1.6	Задачи и упражнения . . . . .	19
<b>2</b>	<b>Классы P, NP и coNP</b>	<b>21</b>
2.1	Что называть быстрым алгоритмом? . . . . .	21
2.2	Детерминированные сложностные классы . . . . .	22
2.2.1	Класс P . . . . .	22
2.2.2	Более высокие классы . . . . .	23
2.2.3	Неконструктивные оценки сложности . . . . .	24
2.3	Класс NP . . . . .	25
2.3.1	Недетерминированные машины Тьюринга . . . . .	25
2.3.2	Два определения класса NP и их эквивалентность . . . . .	27
2.3.3	Примеры задач из NP . . . . .	28
2.4	Проблема перебора ( $P \stackrel{?}{=} NP$ ) . . . . .	29
2.5	Класс coNP . . . . .	30
2.6	Класс NEXP . . . . .	32
2.7	Исторические замечания и рекомендации по литературе . . . . .	32
2.8	Задачи и упражнения . . . . .	32
<b>3</b>	<b>NP-полнота</b>	<b>35</b>
3.1	Полиномиальная сводимость по Карпу . . . . .	35
3.2	NP-трудность и NP-полнота . . . . .	36
3.3	Выполнимость булевых формул . . . . .	37
3.3.1	Сводимость к 3-КНФ . . . . .	37
3.3.2	Сводимость раскраски к выполнимости . . . . .	38
3.3.3	Теорема Кука–Левина . . . . .	38
3.4	Примеры NP-полных задач . . . . .	39
3.4.1	Клика, независимое множество, вершинное покрытие . . . . .	39
3.4.2	Раскраски . . . . .	40
3.4.3	Пути и циклы в графе . . . . .	43
3.4.4	Покрывание множествами и задача о рюкзаке . . . . .	46
3.4.5	Вариации с выполнимостью . . . . .	47

3.4.6	Линейное и квадратичное программирование . . . . .	48
3.5	Математические доказательства . . . . .	51
3.6	Задачи поиска . . . . .	52
3.6.1	Сводимость по Левину . . . . .	53
3.6.2	Самосводимость в <b>NP</b> -полных задачах . . . . .	54
3.7	Как решать <b>NP</b> -полные задачи на практике . . . . .	55
3.8	Исторические замечания и рекомендации по литературе . . . . .	56
3.9	Задачи и упражнения . . . . .	56
<b>4</b>	<b>Техника диагонализации</b> . . . . .	<b>57</b>
4.1	Иерархия по времени . . . . .	57
4.2	Существование <b>NP</b> -промежуточных задач . . . . .	59
4.3	Релятивизация утверждения <b>P = NP</b> . . . . .	61
4.3.1	Вычисления с оракулом . . . . .	61
4.3.2	Теорема Бейкера–Джилла–Соловья . . . . .	62
4.3.3	Случайный оракул . . . . .	63
4.3.4	Нерелятивизируемость теоремы Кука–Левина . . . . .	65
4.4	Исторические замечания и рекомендации по литературе . . . . .	65
4.5	Задачи и упражнения . . . . .	65
<b>5</b>	<b>Пространственная сложность</b> . . . . .	<b>67</b>
5.1	Сложностные классы, связанные с памятью . . . . .	67
5.2	Теорема Сэвича . . . . .	70
5.3	<b>PSPACE</b> -полнота . . . . .	71
5.3.1	Общая теория . . . . .	71
5.3.2	Первые примеры полных задач . . . . .	71
5.3.3	Булевы формулы с кванторами . . . . .	72
5.3.4	Обобщённая игра в города . . . . .	73
5.4	Вычисления на логарифмической памяти . . . . .	75
5.4.1	Примеры языков из <b>L</b> . . . . .	75
5.4.2	<b>NL</b> -полнота . . . . .	78
5.4.3	Сертификатное определение <b>NL</b> . . . . .	80
5.4.4	<b>NL</b> -полнота задачи <b>PATH</b> . . . . .	80
5.4.5	<b>NL = coNL</b> . . . . .	81
5.4.6	Другие <b>NL</b> -полные задачи . . . . .	82
5.5	Исторические замечания и рекомендации по литературе . . . . .	84
5.6	Задачи и упражнения . . . . .	84
<b>6</b>	<b>Полиномиальная иерархия</b> . . . . .	<b>85</b>
6.1	Уровни полиномиальной иерархии . . . . .	85
6.2	Отделимость классов иерархии друг от друга . . . . .	88
6.2.1	Коллапсированные иерархии . . . . .	88
6.2.2	Полные задачи на уровнях иерархии . . . . .	88
6.3	Альтернирующие машины Тьюринга . . . . .	89
6.4	Определение иерархии при помощи оракулов . . . . .	91
6.5	Другие иерархии: обзор . . . . .	92
6.6	Исторические замечания и рекомендации по литературе . . . . .	93
6.7	Задачи и упражнения . . . . .	93

<b>7</b>	<b>Схемная сложность</b>	<b>95</b>
7.1	Определения	95
7.1.1	Схемы как графы	95
7.1.2	Схемы как программы	97
7.2	Класс $P_{poly}$ и машины с подсказками	98
7.3	Соотношения $P_{poly}$ с другими классами	100
7.3.1	Теорема Карпа–Липтона	100
7.3.2	Теорема Мейера	101
7.4	NC-иерархия	101
7.4.1	Определения классов $NC^d$ и $AC^d$	101
7.4.2	Семейства схем для сложения и умножения	103
7.4.3	Связь с классами $L$ и $NL$	104
7.5	Обзор других классов	106
7.6	Исторические замечания и рекомендации по литературе	106
7.7	Задачи и упражнения	106
<b>8</b>	<b>Вероятностная сложность</b>	<b>107</b>
8.1	Что такое вероятностные вычисления	107
8.2	Примеры вероятностных алгоритмов	108
8.2.1	Проверка двух многочленов на равенство	108
8.2.2	Проверка числа на простоту	109
8.3	Вероятностные сложностные классы	110
8.3.1	Классы $BPP$ , $RP$ , $coRP$ и $ZPP$	110
8.3.2	Амплификация (уменьшение ошибки)	112
8.3.3	Роль вероятностного распределения	114
8.3.4	Связь со схемами из функциональных элементов	116
8.3.5	Связь с полиномиальной иерархией	117
8.3.6	$BPP$ -полные задачи?	118
8.4	Логарифмическая память	118
8.4.1	Определения и простые включения	118
8.4.2	Случайные блуждания в графах	120
8.5	О числе затраченных случайных битов	123
8.5.1	Использование попарно независимых наборов случайных битов	123
8.6	Исторические замечания и рекомендации по литературе	124
8.7	Задачи и упражнения	124
<b>9</b>	<b>Сложность задач подсчёта</b>	<b>127</b>
9.1	Класс $\#P$	127
9.2	Полнота в классе $\#P$	128
9.2.1	Определение и простые примеры	129
9.2.2	Класс $RP$	130
9.2.3	Задача о подсчёте перманента	132
9.3	Языки с единственным сертификатом	136
9.3.1	Семейства попарно независимых хеш-функций	136
9.3.2	Теорема Вэлианта–Вазирани	137
9.4	Класс $\oplus P$	139
9.5	Теорема Тоды	140
9.5.1	Построение вероятностного алгоритма	140
9.5.2	Дерандомизация	143
9.6	Подсчёт числа изоморфизмов в графе	144

9.7	Иерархия подсчёта . . . . .	146
9.8	Исторические замечания и рекомендации по литературе . . . . .	147
9.9	Задачи и упражнения . . . . .	147
<b>10</b>	<b>Сложность в среднем</b>	<b>149</b>
10.1	Роль распределения входов . . . . .	149
10.2	Полиномиальность в среднем и класс <b>distP</b> . . . . .	151
10.3	Класс <b>distNP</b> . . . . .	152
10.3.1	Сводимость в <b>distNP</b> . . . . .	153
10.3.2	Полные задачи в <b>distNP</b> . . . . .	154
10.4	Задачи с квазиравномерными распределениями . . . . .	155
10.5	Пять миров Импальяццо . . . . .	156
10.6	Исторические замечания и рекомендации по литературе . . . . .	156
10.7	Задачи и упражнения . . . . .	156
<b>11</b>	<b>Сложность задач поиска</b>	<b>157</b>
11.1	Что такое задача поиска . . . . .	157
11.2	Класс <b>TFNP</b> и его подклассы . . . . .	158
11.2.1	Класс <b>PLS</b> . . . . .	159
11.2.2	Класс <b>PPA</b> . . . . .	160
11.2.3	Классы <b>PPAD</b> и <b>PPADS</b> . . . . .	164
11.2.4	Класс <b>PPP</b> . . . . .	165
11.3	Полнота в классе <b>PPAD</b> . . . . .	165
11.3.1	Лемма Шпернера . . . . .	166
11.3.2	Теорема Брауэра . . . . .	166
11.3.3	Равновесие Нэша . . . . .	166
11.3.4	Рыночные равновесия . . . . .	166
11.4	Исторические замечания и рекомендации по литературе . . . . .	166
11.5	Задачи и упражнения . . . . .	166
<b>12</b>	<b>Интерактивные доказательства</b>	<b>169</b>
12.1	Интерактивные системы доказательств . . . . .	169
12.2	Класс <b>IP</b> . . . . .	171
12.2.1	Метафора . . . . .	171
12.2.2	Задача о неизоморфизме графов . . . . .	172
12.2.3	Вариации с точностью и полнотой . . . . .	172
12.3	Игры Артура–Мерлина . . . . .	173
12.3.1	Классы <b>MA</b> и <b>AM</b> . . . . .	174
12.3.2	Уменьшение числа раундов . . . . .	175
12.3.3	Идеальная полнота . . . . .	176
12.3.4	Связь с другими сложностными классами . . . . .	176
12.3.5	Неизоморфизм графов с общей случайностью . . . . .	177
12.3.6	Моделирование частных случайных битов при помощи об- щих . . . . .	180
12.3.7	Может ли задача GI быть NP-полной? . . . . .	182
12.4	<b>IP = PSPACE</b> . . . . .	183
12.4.1	Идея арифметизации . . . . .	183
12.4.2	Интерактивные доказательства для тавтологий . . . . .	184
12.4.3	Интерактивные доказательства для булевых формул с кван- торами . . . . .	186
12.5	Исторические замечания и рекомендации по литературе . . . . .	188

12.6 Задачи и упражнения . . . . .	189
<b>13 Доказательства с нулевым разглашением . . . . .</b>	<b>191</b>
13.1 Идея нулевого разглашения . . . . .	191
13.1.1 Передача знания или передача информации? . . . . .	192
13.2 Совершенно нулевое разглашение . . . . .	193
13.2.1 «Сезам, откройся!» . . . . .	193
13.2.2 Протокол для задачи об изоморфизме графов . . . . .	194
13.2.3 Формальное определение . . . . .	194
13.2.4 Корректность протокола . . . . .	195
13.3 Статистически нулевое разглашение . . . . .	196
13.4 Вычислительно нулевое разглашение . . . . .	197
13.4.1 Неформальный протокол для задачи о 3-раскраске . . . . .	197
13.5 Амплификация . . . . .	197
13.6 Исторические замечания и рекомендации по литературе . . . . .	197
13.7 Задачи и упражнения . . . . .	198
<b>14 Вероятностно проверяемые доказательства . . . . .</b>	<b>199</b>
14.1 Понятие вероятностно проверяемого доказательства . . . . .	199
14.2 Приближённое решение <b>NP</b> -трудных задач . . . . .	200
14.3 Эквивалентность двух подходов . . . . .	200
14.4 Экспоненциальная <b>RSP</b> -теорема . . . . .	200
14.5 Исторические замечания и рекомендации по литературе . . . . .	200
14.6 Задачи и упражнения . . . . .	200
<b>15 Рациональные интерактивные доказательства . . . . .</b>	<b>201</b>
15.1 Однораундовые доказательства . . . . .	201
15.2 Многораундовые доказательства . . . . .	201
15.3 Исторические замечания и рекомендации по литературе . . . . .	201
15.4 Задачи и упражнения . . . . .	201
<b>A Математические сведения . . . . .</b>	<b>203</b>
A.1 Множества и логика . . . . .	203
A.2 Комбинаторика . . . . .	203
A.3 Теория графов . . . . .	203
A.3.1 Вершины и рёбра . . . . .	203
A.3.2 Изоморфизмы . . . . .	204
A.3.3 Маршруты в графах . . . . .	204
A.4 Теория вероятностей . . . . .	204
A.5 Алгебра и теория чисел . . . . .	204
<b>B Ответы, указания, решения . . . . .</b>	<b>205</b>
Глава 1 . . . . .	205
Глава 2 . . . . .	206
Глава 3 . . . . .	207
Глава 4 . . . . .	207
Глава 5 . . . . .	207
Глава 6 . . . . .	207
Глава 7 . . . . .	207
Глава 8 . . . . .	207
Глава 9 . . . . .	208
Глава 10 . . . . .	208

Глава 11 . . . . .	208
Глава 12 . . . . .	209
Глава 13 . . . . .	210
Глава 14 . . . . .	211
<b>Список литературы</b>	<b>213</b>



# Глава 1

## Модели вычислений

Чистая математика изучает какие угодно объекты и функции и ставит любые вопросы о них. Важную роль играют бесконечные объекты: действительные и комплексные числа, топологические пространства, ординалы и т.п. Ответ на поставленный вопрос часто может быть неконструктивен: например, существование объекта доказывается без конкретной процедуры его поиска.

Классическая теория вычислимости накладывает определённые рамки. Во-первых, изучаются лишь конечные объекты, то есть такие, которые можно закодировать словом в некотором алфавите или натуральным числом. Это могут быть сами числа, конечные их наборы, матрицы, графы и другие подобные структуры. Во-вторых, из всех функций над такими объектами рассматриваются лишь вычислимые. Это естественное ограничение: ведь и у функции (бесконечного объекта) тоже должно быть конечное описание. В-третьих, решаемыми задачами считаются не те, где решение есть в принципе, а лишь те, где решение можно найти при помощи алгоритма. Для построения связной теории важно, что есть общепринятое понятие алгоритма, не зависящее от низкоуровневых деталей реализации. Сами реализации могут быть самыми разными: машины Тьюринга и их вариации, адресные машины, рекурсивные функции, лямбда-комбинаторы и т.д.

Теория сложности вычислений идёт ещё на шаг дальше: решаемыми задачами считаются лишь те, у которых есть эффективное решение, то есть разрешающий алгоритм, работающий не слишком долго.<sup>1</sup> Основной (и нерешённый) вопрос состоит в том, как отличить решаемую задачу от нерешаемой. Для начала его нужно чётко поставить. Этим мы и займёмся в этой главе.

### 1.1 Виды вычислительных задач и вычислительных ресурсов

Вначале мы очертим круг задач, о которых идёт речь. Во-первых, мы будем изучать только задачи про конечные объекты.<sup>2</sup> Под конечными объектами мы

---

<sup>1</sup>В английском языке есть два разных слова, означающие эффективность: *effectiveness* и *efficiency*. Первое из них означает, что задача в принципе решается, т.е. речь идёт о разрешимом множестве или вычислимой функции. Второе означает эффективность в вышеуказанном смысле.

<sup>2</sup>Вообще говоря, можно изучать сложность вычислительных задач и для бесконечных объектов. Так, ещё до создания современных компьютеров строились *аналоговые* вычислительные машины. Если мышление человека рассматривать как вычислительный процесс, то он именно аналоговый. Многие численные методы, например, метод Ньютона, естественно рассматривать как алгоритмы над действительными числами. Есть и специальная область тео-

будем понимать объекты, которые можно «эффективно» закодировать словами в некотором алфавите (без ограничения общности, в двоичном). Эффективность кодирования означает, что по коду объекта можно вычислить все ключевые характеристики. Например, по коду графа можно вычислить, сколько в нём вершин и какие из них соединены рёбрами, по коду матрицы можно вычислить её размер и число в любой ячейке, и т.д. Как правило, кодирование не представляет особых проблем: с обычными строками, описывающими объекты, можно работать алгоритмически. Например, натуральные числа представляются своей двоичной (или десятичной) записью, вектора и матрицы представляются как цепочки записей чисел с разделителями, граф можно представить в виде списка рёбер или в виде матрицы смежности, и т.д. Во всех примерах мы будем считать, что все объекты уже закодированы как слова в некотором алфавите, и не будем отдельно обозначать код объекта. Разумеется, исключением будут бесконечные объекты, такие как действительные числа или геометрические фигуры. Такие объекты мы рассматривать не будем, хотя содержательная теория возможна и для них.

Мы будем изучать массовые задачи, т.е. бесконечные семейства индивидуальных задач для всевозможных конкретных значений параметров. Алгоритм должен действовать по одной и той же инструкции для всех экземпляров. У каждого алгоритма должен быть вход и выход. В зависимости от того, как они выглядят, мы будем различать несколько основных видов вычислительных задач:

- *Задачи поиска.* Наверное, это первая постановка, которая приходит на ум при слове «задача»: дано условие, нужно найти решение. Например, требуется решить уравнение, или найти кратчайший маршрут в графе, или составить расписание согласно требованиям, или придумать, как уместить все детали на один лист фанеры. Формально задано отношение  $R \subset \Sigma^* \times \Sigma^*$  и слово  $x$ , а найти нужно слово  $y$ , такое что  $xRy$ , если такое слово есть. Иными словами, алгоритм на любом  $x$  должен вернуть либо  $y$ , для которого  $xRy$ , либо  $\perp$ , если таких  $y$  нет.
- *Задача распознавания языка.* Может быть так, что итоговый ответ бинарный: да или нет. Например, является ли число простым, подходит ли пароль к сохранённому хеш-значению, легитимна ли транзакция в платёжной системе, горит ли зелёный на светофоре. В общем случае по слову  $x$  нужно выяснить, лежит ли оно в языке  $L \subset \Sigma^*$ . Иными словами, алгоритм должен вернуть 1 на любом  $x \in L$  и вернуть 0 на любом  $x \notin L$ . Можно также сказать, что алгоритм должен проверить, выполнено ли для слова  $x$  данное свойство.
- *Задача вычисления функции.* Если в задаче поиска всегда ровно один ответ, то удобно воспринимать его как функцию от входа. Это может быть отсортированный массив, значение арифметического выражения, разложение на простые множители. В общем случае задана некоторая функция  $f: \Sigma^* \rightarrow \Sigma^*$ , а алгоритм должен на любом  $x$  вернуть  $f(x)$ . Как правило, мы будем считать, что  $f$  всюду определена.
- *Задача подсчёта.* Бывает так, что важно посчитать количество решений: например, сколько может быть различных меню в столовой, удовлетворя-

---

рии — алгебраические вычисления. Тем не менее, все эти аспекты выходят за рамки нашего повествования.

ющих всем требованиям и возможностям, чтобы каждый день было новое меню. Либо в результате тонкого анализа оказывается, что значение функции есть число решений некоторой задачи. В общем случае для фиксированного отношения  $R \subset \Sigma^* \times \Sigma^*$  по слову  $x$  нужно найти количество слов  $y$ , таких что  $xRy$ .

- *Задача оптимизации.* Это ещё один вид задач, часто встречающийся на практике. Люди и фирмы всё время что-то оптимизируют: минимизируют время выполнения задания или затраченные деньги, максимизируют прибыль или удовольствие. Вообще говоря, задача оптимизация есть частный случай задачи поиска: по слову  $x$  нужно найти такой  $y$ , что значение  $f(x, y)$  максимально (или минимально), где  $f: \Sigma^* \times \Sigma^* \rightarrow \mathbb{Q}$  — некоторая числовая функция. Альтернативно, может потребоваться найти лишь само оптимальное значение.
- *Задача аппроксимации.* На практике не требуется найти настоящий оптимум, приближённое решение тоже хорошо подойдёт. В теории задача ставится так: по слову  $x$  нужно найти такой  $y$ , что значение  $f(x, y)$  составляет хотя бы  $(1 - \varepsilon)$  долю от максимального (или не более чем в  $(1 + \varepsilon)$  раз больше минимального). Вновь может потребоваться найти лишь само приближённо оптимальное значение.

В основном мы будем работать с задачами распознавания языка. Исключениями будут глава 9, в которой изучаются задачи подсчёта, и глава 11, в которой изучаются задачи поиска, а также отдельные разделы в других главах.

Для алгоритмического решения задач требуются различные вычислительные ресурсы. Некоторые из них сразу приходят на ум, другие доступны только в теории.

- *Время работы.* Как правило, задачу нужно решить не в принципе, а к определённому сроку. Заведомой теоретической верхней границей является предполагаемый срок дальнейшего существования Вселенной, но обычно дедлайн куда ближе. Поэтому время работы — важнейшая характеристика программы. Поскольку мы изучаем алгоритмы, выполняемые пошагово, то время удобно мерить в этих шагах. В том или ином виде про время работы будет идти речь во всех последующих главах.
- *Память.* Алгоритму необходимо рабочее пространство для промежуточных вычислений. Обычно в это пространство не включают вход программы: например, при поиске информации среди всех файлов диска разумно подсчитывать только оперативную память, выделенную под этот процесс, и временные файлы, а не общий объём диска. Пространственной сложности посвящена глава 5.
- *Случайность.* Некоторые задачи (например, проверка числа на простоту или проверка двух арифметических выражений на равенство) при современном уровне знаний решаются гораздо быстрее, если применить вероятностный алгоритм вместо детерминированного. Как правило, расплатой за это ускорение будет маленькая, но положительная вероятность ошибки. Помимо самого факта использования случайных битов, важными характеристиками являются «количество» и «качество» использованной случайности. Под количеством понимается число случайных битов, под качеством — близость их распределения к равномерному, в том числе малые

корреляции между значениями разных битов. Вероятностным алгоритмам посвящена глава 8.

- *Подсказка.* Если машина получает небольшую подсказку, зависящую только от длины входа, она может существенно ускориться. Более того, с подсказкой можно решить даже некоторые невычислимые функции. Использование подсказки моделирует ситуацию, в которой длина входа заранее известна и можно сконструировать микросхему ровно для этой длины. Например, функционирование криптовалюты Bitcoin основано на вычислении хеш-функции SHA-256, где 256 и есть длина входа. Если на стадии формирования системы она работала на базе обычных персональных компьютеров, затем на базе мощных видеокарт, то сейчас основную роль играют специально сконструированные «фермы», предназначенные ровно для вычисления SHA-256 и ни для чего другого. Схемной сложности посвящена глава 7.
- *Параллелизм.* Некоторые задачи «хорошо параллелятся», то есть разбиваются на подзадачи, выполнение которых не зависит друг от друга. Если можно запускать подзадачи на разных машинах, то общее время работы существенно уменьшится. Поэтому изобретение алгоритмов для параллельных вычислений или доказательство, что таковых нет, весьма важно для практики. Именно за счёт более успешного параллелизма растёт производительность современных процессоров.
- *Длина программы.* Бывают ситуации, когда программа с более длинным текстом будет работать быстрее. Поэтому ограничения на длину программы могут быть важны с точки зрения расхода остальных ресурсов. Изучению этого фактора посвящена теория колмогоровской сложности, в том числе с ограничением на ресурсы. Её рассмотрение выходит за рамки данной книги, а подробное изложение можно найти в книгах [147] и [91].
- *Интерактивность.* В современном мире всё большее распространение получают облачные вычисления и другие подходы с делегированием вычислительных мощностей. За счёт подобной интерактивности, с одной стороны, можно получить доступ к большим вычислительным ресурсам, с другой, приходится доверяться удалённому вычислителю. Оказывается, часто проверка результата может быть значительно проще его получения, поэтому можно использовать интерактивность без опасений. Интересно, что интерактивность гораздо эффективнее работает в сочетании со случайностью. На этом основана теория интерактивных доказательств, рассматриваемая в главе 12. Мерой интерактивности можно назвать общее число переданных битов, называемой коммуникационной сложностью протокола. Изучение этой темы также выходит за рамки данной книги, подробное изложение можно найти в книгах [kush-nisan] и [10, гл. 13].
- *Специальные модели вычислений.* Существуют модели вычислений, которые пока что не умеют симулировать на стандартных компьютерах без значительного роста вычислительных ресурсов. Прежде всего, это недетерминированные, альтернирующие и квантовые вычисления. Некоторые задачи на данный момент решаются в этих моделях гораздо быстрее, чем в классической. Недетерминированным вычислениям посвящены главы 2

и 3, упоминаются они и в других главах. Про альтернирующие вычисления говорит глава 6. Квантовые вычисления остаются за рамками нашего рассмотрения, про них можно почитать в книгах [154] и [10, гл. 9].

- *Оракул.* Вычисления с оракулом позволяют про любое слово за один шаг узнавать, принадлежит ли это слово некоторому фиксированному множеству. Как известно, это даже расширяет класс алгоритмически решаемых задач. Также это расширяет класс задач, решаемых эффективно, а количество запросов к оракулу можно само по себе воспринимать как меру эффективности алгоритма. Вычисления с оракулом отличаются и от вычислений с подсказкой: при вычислениях на разных словах запросы к оракулу могут быть разными, — и от интерактивных: ответ оракула на повторно заданный вопрос будет таким же, как в первый раз. О вычислениях с оракулом пойдёт речь в разделе 4.3, а также в нескольких других местах.

Возможно, в будущем будут изучаться и другие важные на практике ресурсы, такие как электроэнергия, рабочее время программиста или общие финансовые затраты.<sup>3</sup> Пока таких теорий не создано.

## 1.2 Измерение сложности задачи

### 1.2.1 Меры сложности алгоритма на данном входе

Поговорим о том, в каких единицах измерять сложность алгоритма. Мы уже перечислили некоторое количество вычислительных ресурсов. Во многих случаях единицы измерения очевидны из определения: число использованных случайных битов, число параллельно работающих машин, длина подсказки, число запросов к оракулу и т.д. В других случаях выбор бинарный, поэтому никаких единиц нет. Например, квантовые вычисления, интерактивность или возможность адаптивных запросов к оракулу либо доступны, либо нет. А вот к определению времени работы и памяти возможны разные подходы.

Как правило, в вычислительных моделях есть понятие элементарного шага и элементарной ячейки памяти. Соответственно, именно эти элементарные единицы используются для подсчёта времени работы и памяти. Этот подход называется битовой сложностью. В некоторых моделях возможны более сложные подходы. Например, может подсчитываться количество алгебраических операций над натуральными числами (отдельно аддитивных и мультипликативных, этот подход называется алгебраической сложностью), или необходимое число регистров, в которых также хранятся натуральные числа. Ярким примером алгебраического подхода является задача о быстром умножении матриц. Вычисление произведения двух матриц  $n \times n$  по определению требует  $O(n^3)$  мультипликативных операций: в каждой из  $n^2$  ячеек нужно перемножить  $n$  пар чисел и потом сложить. Алгоритм Штрассена снижает это число до  $O(n^{\log_2 7}) \approx O(n^{2.807})$ . Аддитивных операций при этом становится больше, но асимптотически этим можно пренебречь, т.к. сложение быстрее умножения. Алгоритм Копперсмита–Винограда снижает число мультипликативных операций до  $O(n^{2.375})$ , недавние

---

<sup>3</sup>Эти ресурсы важны не столько для отдельных алгоритмов, сколько для распределённых систем, таких как криптовалюты. Например, Bitcoin отличается крайне низкой средней энергетической эффективностью: по состоянию на сентябрь 2017 года одна транзакция требует около 100 кВт·ч. При этом нельзя сказать, что такие затраты объективно нужны для решаемых задач, они возникают из-за особенностей конкретной реализации.

достижения ещё чуть-чуть улучшили эту оценку. Алгебраический подход позволяет выделить самое важное: во-первых, мультипликативные операции производятся значительно дольше аддитивных, и сокращение числа умножений за счёт увеличения числа сложений оправдано. Во-вторых, мы отделяем количество операций с элементами матрицы от сложности осуществления отдельной такой операции, которую на практике можно считать константой, а в теории она даст лишние полилогарифмические множители.

### 1.2.2 Сложность конкретного алгоритма

Пусть фиксированы конкретный алгоритм и конкретная мера его сложности, например, число шагов машины Тьюринга. Как из сложности работы на каждом входе получить сложность всего алгоритма? Во-первых, для слов одинаковой длины можно использовать два подхода: подсчитывать сложность в худшем случае, измеряемую как максимум по всем входам данной длины, и сложность в среднем, для которой эти величины так или иначе усредняются. Конкретный выбор зависит от приложений: если мы интересуемся, есть ли универсальный эффективный алгоритм для всех экземпляров задачи, то нужно считать сложность в худшем случае. А если мы хотим использовать сложность задачи, например, для построения стойкого шифра, то нужно считать сложность в среднем. Возможно также изучение сложности в «типичном» случае: на входах, которые «обычно» встречаются на практике. Но такой подход трудно формализовать. Во-вторых, разных длин тоже бесконечно много, поэтому мы будем изучать *асимптотическое* поведение сложностной меры при длине, стремящейся к бесконечности. У такого подхода есть недостатки, ведь асимптотическое поведение может быть достигнуто только при нереальных значениях длины входа. Таких примеров немало, но цельную теорию трудно построить иначе.

### 1.2.3 Асимптотические обозначения

Итак, мы анализируем только асимптотическое поведение потраченных ресурсов. Специальные обозначения для асимптотик использовались в теории чисел и математическом анализе с конца XIX века. Точный смысл этих обозначений для информатики был введён Дональдом Кнутом в 1976 году [83]:

**Определение 1.1.** Пусть функции  $f$  и  $g$  отображают натуральные числа в натуральные ненулевые числа. Тогда:

- $f(n) = O(g(n))$ , если  $\exists C \exists N \forall n > N \ f(n) < Cg(n)$ ;
- $f(n) = \Omega(g(n))$ , если  $\exists c > 0 \exists N \forall n > N \ f(n) > cg(n)$ ;
- $f(n) = \Theta(g(n))$ , если  $\exists c > 0 \exists C \exists N \forall n > N \ cg(n) < f(n) < Cg(n)$ ;
- $f(n) = o(g(n))$ , если  $\forall c > 0 \exists N \forall n > N \ f(n) < cg(n)$ ;
- $f(n) = \omega(g(n))$ , если  $\forall C \exists N \forall n > N \ f(n) > Cg(n)$ .

Иными словами,  $o$  обозначает асимптотически меньшую скорость роста,  $O$  — асимптотически не большую,  $\Theta$  — ровно такую же,  $\Omega$  — не меньшую,  $\omega$  — строго большую. Одну и ту же функцию в разных ситуациях можно назвать и  $O(g(n))$ , и  $\Omega(g(n))$ , в зависимости от того, верхняя или нижняя оценка важна для контекста.

Помимо базовых, используются такие синтетические обозначения:



- $f(n) = \text{poly}(g(n))$ , если для некоторой константы  $c$  выполнено  $f(n) = O((g(n))^c)$ ;
- $f(n) = \text{polylog}(g(n))$ , если для некоторой константы  $c$  выполнено  $f(n) = O((\log g(n))^c)$ ;
- $f(n) = \tilde{O}(g(n))$ , если  $f(n) = O(g(n) \text{polylog}(g(n)))$ .

#### 1.2.4 От сложности алгоритма к сложности задачи

Одну и ту же задачу можно решать самыми разными алгоритмами. У каждого из этих алгоритмов есть своя сложность. Естественно ожидать, что для каждой задачи есть оптимальный алгоритм и именно его сложность является сложностью задачи. К сожалению, оптимального алгоритма может не быть: небольшие дополнительные улучшения могут приводить ко всё меньшим и меньшим сложностям, а предельная сложность не будет достигаться никаким конкретным алгоритмом. Теорема Блума [22] показывает, что какую меру сложности ни выбрать, можно подобрать задачу, выглядящую именно так.

Поэтому о сложности задачи всегда говорят в терминах верхних или нижних оценок. Как правило, для задачи есть какой-то алгоритм сложности  $O(g(n))$ , тогда говорят, что задача лежит в классе с такой сложностью. В гораздо более редких случаях удаётся получить нетривиальную нижнюю оценку: доказать, что любой алгоритм, решающий задачу, имеет сложность  $\Omega(h(n))$ . Например, для умножения матриц есть тривиальная нижняя оценка  $\Omega(n^2)$  и оценка Раца в модели арифметических схем с ограниченными коэффициентами  $\Omega(n^2 \log n)$ .

### 1.3 Базовая модель: многоленточная машина Тьюринга

Базовой вычислительной моделью для нашего курса служит многоленточная машина Тьюринга. В некоторых приложениях удобнее использовать другие модели, например адресные машины или алгебраическую сложность, но для теории машина Тьюринга наиболее удобна. Мы определим машину, решающую задачу распознавания языка.

**Определение 1.2.** *Детерминированной машиной Тьюринга с  $k$  лентами называется кортеж  $\langle \Sigma, \Gamma, Q, q_1, q_a, q_r, k, \delta \rangle$ , где  $\Sigma$ ,  $\Gamma$  и  $Q$  суть конечные непустые множества, причём  $\Sigma \subset \Gamma$  и  $\Gamma \cap Q = \emptyset$ ,  $q_1, q_a, q_r \in Q$  попарно различны, а  $\delta$  есть функция из  $(Q \setminus \{q_a, q_r\}) \times \Gamma^k$  в  $Q \times \Gamma^k \times \{L, N, R\}^k$ . Множество  $\Sigma$  называется входным алфавитом,  $\Gamma$  — ленточным алфавитом,  $Q$  — множеством состояний,  $q_1$ ,  $q_a$  и  $q_r$  — начальным, принимающим и отвергающим состояниями соответственно, а  $\delta$  — функцией перехода. Среди элементов  $\Gamma$  выделяют специальный символ  $\#$  (бланк, пробел, пустой символ, обозначается также как  $\_$ ,  $\square$ ,  $\flat$ ), не входящий в множество  $\Sigma$ .*

Неформально говоря, машина состоит из  $k$  бесконечных в обе стороны лент, разделённых на ячейки, и управляющего блока с указателями на каждую из  $k$  лент. За один такт машина считывает символы со всех  $k$  ячеек, на которые указывает, и в зависимости от внутреннего состояния и прочтённых символов переходит в новое состояние, записывает новые символы и сдвигает каждый указатель влево или вправо или оставляет его на месте. В начале работы на

первой ленте написано слово  $x$ , все остальные ленты пусты, машина находится в состоянии  $q_1$ , а указатель на первой ленте указывает на первый символ слова  $x$ . Если через некоторое количество тактов машина приходит в состояние  $q_a$ , то говорят, что машина *принимает* слово  $x$ , если машина приходит в состояние  $q_r$ , то *отвергает*. Дадим формальное определение вычисления на машине Тьюринга.

**Определение 1.3.** Конфигурацией  $k$ -ленточной машины Тьюринга  $M$  называется кортеж

$$(a_1, \dots, a_k; b_1, \dots, b_k; q),$$

где  $a_i \in \Gamma^+$ ,  $b_i \in \Gamma^+$  и  $q \in Q$ .

Смысл определения: машина находится в состоянии  $q$ , для каждого  $i \in \{1, \dots, k\}$  на  $i$ -й ленте написано слово  $a_i b_i$  (остальная часть ленты заполнена бланками), причём указатель находится на первом символе слова  $b_i$ . Если слева от указателя всё заполнено бланками, то один из них выделяется в качестве  $a_i$ , аналогично для правой части и  $b_i$ . Такое соглашение заключается для непустоты всех упомянутых слов.

**Определение 1.4.** Пусть  $C = (a_1 \sigma_1, \dots, a_k \sigma_k; \tau_1 b_1, \dots, \tau_k b_k; q)$  — некоторая конфигурация, где  $q \neq q_a$  и  $q \neq q_r$ . (Поскольку все исходные слова  $a_i$  и  $b_i$  непусты, можно выделить их последние и первые символы, соответственно, а оставшиеся части переобозначить вновь за  $a_i$  и  $b_i$ ). Пусть  $\delta(q, \tau_1, \dots, \tau_k) = (s, \rho_1, \dots, \rho_k, D_1, \dots, D_k)$ . Тогда следующей за  $C$  конфигурацией будет  $C' = (a'_1, \dots, a'_k; b'_1, \dots, b'_k; s)$ , где:

- Если  $D_i = L$ , то  $a'_i = a_i$  (или  $a'_i = \#$ , если  $a_i = \varepsilon$ ),  $b'_i = \sigma_i \rho_i b_i$ ;
- Если  $D_i = N$ , то  $a'_i = a_i \sigma_i$ ,  $b'_i = \rho_i b_i$ ;
- Если  $D_i = R$ , то  $a'_i = a_i \sigma_i \rho_i$ ,  $b'_i = b_i$  (или  $b'_i = \#$ , если  $b_i = \varepsilon$ ).

Таким образом, машина меняет все  $\tau_i$  на  $\rho_i$  и сдвигается в предписанных ей направлениях на каждой ленте. Очевидно, что за одной конфигурацией следует ровно одна конфигурация. А вот предшествующих конфигураций может быть несколько.<sup>4</sup>

**Определение 1.5.** Вычислением называется последовательность конфигураций  $(C_1, \dots, C_t)$ , такая что для всех  $i = 1, \dots, t-1$  конфигурация  $C_{i+1}$  следует за  $C_i$ .

**Определение 1.6.** Машина *принимает* слово  $x$ , если существует вычисление, начинающееся с конфигурации  $(\underbrace{\#, \dots, \#}_{k \text{ раз}}; x, \underbrace{\#, \dots, \#}_{k-1 \text{ раз}}; q_0)$  и заканчивающаяся в конфигурации, содержащей  $q_a$ . Аналогично машина *отвергает* слово  $x$ , если существует вычисление, начинающееся с той же конфигурации и заканчивающееся конфигурацией, содержащей  $q_r$ .

**Замечание 1.7.** Нетрудно заметить, что вычисление полностью определяется начальным состоянием. Поэтому для каждого  $x$  верно одно из трёх: либо машина принимает его, либо отвергает, либо не останавливается.

Многоленточная машина Тьюринга — очень гибкая модель. Она позволяет моделировать такие вещи как:

<sup>4</sup>Есть специальная теория *обратимых вычислений*, где предшествующая конфигурация тоже всегда одна.



- Входная лента только для чтения и выходная лента только для записи. Это важно для подсчёта памяти именно для промежуточных вычислений.
- Лента со случайными битами.
- Лента для подсказки.
- Лента для общения с оракулом.
- Лента для интерактивного взаимодействия с другими алгоритмами.
- Адресная лента для реализации произвольного доступа.
- Недетерминированные и альтернирующие вычисления.

Более подробно эти вопросы будут рассмотрены в соответствующих разделах. Выбор многоленточной модели вместо одноленточной будет обоснован в следующем разделе.

## 1.4 Распознавание языков

В этом разделе мы определим, когда данная машина решает данную задачу распознавания за данное время.

**Определение 1.8.** Машина распознаёт язык  $A$  за время  $T(n)$ , если она принимает все слова, лежащие в  $A$ , отвергает все слова, не лежащие в  $A$ , и на каждом слове  $x$  работает не больше  $T(|x|)$  шагов.

Здесь измеряется сложность *в худшем случае*: нужно, чтобы максимальное время работы на словах длины  $n$  не превысило  $T(n)$ .

**Определение 1.9.** Классом  $\mathbf{DTIME}(T(n))$  называется класс языков, которые распознаются за время  $O(T(n))$ . Иными словами, время работы машины на любом слове длины  $n$  не превосходит некоторой константы, умноженной на  $T(n)$ .

Выбор  $O(\cdot)$ -обозначения вместо точного значения времени работы связан с тем, что за счёт увеличения числа состояний и числа символов в ленточном алфавите можно уменьшить время работы в некоторое константное число раз: например, в качестве элементарных операций можно считать операции не с битами, а с байтами. Если же изменить и модель вычисления, то время работы может измениться ещё сильнее. Например, при переходе от многоленточной машины к одноленточной время работы может возвестись в квадрат.

**Утверждение 1.10.** *Любой язык, который можно распознать за время  $O(T(n))$  на многоленточной машине Тьюринга, можно распознать за время  $O(T(n)^2)$  на одноленточной машине.*

*Доказательство.* Будем моделировать машину с  $k$  лентами при помощи одноленточной. На ленте новой машины будем хранить содержимое всех лент исходной, а также положения всех указателей. Поскольку исходная машина работает не дольше  $O(T(n))$ , то на каждой из лент непустыми могут быть не больше  $O(T(n))$  ячеек. Таким образом, новая машина займёт не больше  $k \cdot O(T(n))$  ячеек, т.е. также  $O(T(n))$ : число лент для разных машин может быть сколь

угодно большим, но точно не зависит от длины входа. Для моделирования одного шага исходной машины нужно изменить содержимое всех лент и передвинуть указатели. Если исходной машине требуется больше места, то нужно будет освободить ячейки, сдвинув всё содержимое на одну ячейку в сторону. В любом случае моделирование одного шага потребует  $O(T(n))$  шагов, таким образом моделирование всей работы потребует  $O(T(n)) \cdot O(T(n)) = O(T(n)^2)$  шагов.  $\square$

Можно показать (см. задачу 1.9), что язык палиндромов  $\text{PAL} = \{x \mid x = x^R\}$  можно распознать двухленточной машиной за время  $O(n)$ , но нельзя распознать одноленточной машиной за время  $o(n^2)$ . Вместе с тем машину с любым числом лент, работающую за время  $T$ , можно смоделировать на двухленточной со временем работы  $O(T \log T)$ . (Константа в  $O(\cdot)$ -обозначении зависит от исходного числа лент, но не от длины входа и вида функции  $T$ ).

Похожие теоремы можно доказать и для других вариаций вычислительной модели: машин с двумерной лентой, произвольным доступом и т.д. В каждом из этих случаев вычисления можно будет смоделировать на более простой машине с полиномиальным замедлением. Известен *тезис Чёрча–Тьюринга в сильной форме*, или *тезис Кобхэма–Эдмондса*: любое вычисление на реальном устройстве можно смоделировать на одноленточной машине Тьюринга с полиномиальным замедлением. Иногда его также формулируют чуть иначе: любой эффективный алгоритм для реального устройства можно смоделировать как полиномиальный алгоритм для машины Тьюринга. Однако этот тезис может быть неверен, если выполнены два условия. Во-первых, должны существовать физические устройства, способные проводить недетерминированные или квантовые вычисления (а с квантовыми такое вполне возможно). Во-вторых, такие вычисления должно быть невозможно смоделировать с полиномиальным замедлением на обычном компьютере (во что также верит большинство исследователей).

Как правило, класс  $\mathbf{DTIME}(T(n))$  рассматривают не для всех функций  $T$ , а только для «хороших». Во-первых, естественно потребовать  $T(n) \geq n$ , иначе машина не успеет даже прочесть свой вход. (Хотя в моделях параллельных вычислений это требование может быть ослаблено). Во-вторых, естественно потребовать монотонности  $T(n)$ : чем длиннее вход, тем больше время работы. В-третьих, функция должна быть конструируемой по времени: вычислить  $T(n)$  должно быть возможно за время  $T(n)$ .

**Определение 1.11.** Функция  $T: \mathbb{N} \rightarrow \mathbb{N}$  называется *конструируемой по времени*, если существует алгоритм, который за время  $O(T(n))$  по  $1^n$  (т.е. числу  $n$  в унарной записи) получает  $T(n)$  (в бинарной записи).

Унарная запись аргумента введена лишь для единообразия: тогда время вычисления  $T(n)$  измеряется как функция от длины аргумента, т.е.  $n$ , и это время не должно быть больше  $O(T(n))$ . Все «обычные» функции конструируемы по времени:  $n^c$ ,  $2^n$ ,  $n \log n$  и т.д. Более того, для построения функции, не конструируемой по времени, нужно применить специальную нетривиальную конструкцию. С другой стороны, рассмотрение не конструируемых по времени функций приводит к парадоксальным результатам, например, теорема об иерархии (теорема 4.1) будет неверна.

## 1.5 Исторические замечания и рекомендации по литературе

Машина Тьюринга как вычислительная модель появилась в 1931 году. Сложность работы на ней активно изучалась в 1960-х годах.

## 1.6 Задачи и упражнения

**1.1.** . Придумайте монотонные  $f(n)$  и  $g(n)$ , такие что  $f(n) = O(g(n))$ , но  $f(n) \neq \Theta(g(n))$  и  $f(n) \neq o(g(n))$ .

**1.2. Сложение асимптотик.** Пусть  $f(n) = \xi(t(n))$ , а  $g(n) = \eta(t(n))$ , где  $\xi$  и  $\eta$  обозначают какие-то из символов  $O, \Omega, \Theta, o, \omega$ . Докажите, что  $f(n) + g(n) = \max\{\xi, \eta\}(t(n))$ , где максимум берётся по порядку  $o \prec O \prec \Theta \prec \Omega \prec \omega$ .

**1.3. Умножение асимптотик.** Пусть  $f(n) = \xi(t(n))$ , а  $g(n) = \eta(s(n))$ , где  $\xi$  и  $\eta$  обозначают какие-то из символов  $O, \Omega, \Theta, o, \omega$ . Постройте «таблицу умножения»:  $f(n) \cdot g(n) = \zeta(t(n) \cdot s(n))$ , где  $\zeta$  либо также означает один из этих символов, либо равно ?, если ответ неоднозначен.

**1.4. Функция под асимптотикой.** Пусть  $f$  — возрастающая функция, а  $g(n) = \xi(t(n))$ , где  $\xi \in \{O, \Omega, \Theta, o, \omega\}$ . Верно ли, что  $f(g(n)) = \xi(f(t(n)))$ ? Если нет, то можно ли что-нибудь гарантировать для  $f(g(n))$ ?

**1.5. Композиция асимптотик.** Пусть  $f(n) = \xi(t(n))$ , а  $g(n) = \eta(f(n))$ , где  $\xi$  и  $\eta$  обозначают какие-то из символов  $\Omega, \Theta, \omega$ . Докажите, что  $g(n) = \max\{\xi, \eta\}(t(n))$ . Что будет, если  $\xi$  и  $\eta$  обозначают какие-то из символов  $O, \Theta, o$ ? Любые из 5 символов?

**1.6.** Докажите, что функции  $n$ ,  $n \log n$ ,  $n^2$ ,  $2^n$  являются конструируемыми по времени.

**1.7. Клетчатая машина Тьюринга.** Пусть каждая из лент машины Тьюринга представляет из себя клетчатую плоскость, по которой машина может сдвигаться в любом из четырёх направлений. Докажите, что язык, распознающийся на такой машине за  $T(n)$ , лежит в  $\mathbf{DTIME}((T(n))^2)$ .

**1.8. Машина Тьюринга с произвольным доступом.** Рассмотрим машину с произвольным доступом (random access). У такой машины в паре с каждой рабочей лентой есть адресная; также есть специальное состояние  $q_{access}$  и два спецсимвола  $\mathbf{W}$  и  $\mathbf{R}$  в ленточном алфавите. Если машина оказывается в состоянии  $q_{access}$  в конфигурации  $\text{bin}(i)q_{access}\mathbf{W}\sigma$  на одной из адресных лент ( $\text{bin}(i)$  — двоичная запись числа  $i$ ), то символ  $\sigma$  записывается в ячейку с номером  $i$  на соответствующей рабочей ленте. Если машина оказывается в состоянии  $q_{access}$  в конфигурации  $\text{bin}(i)q_{access}\mathbf{R}$  на адресной ленте, то в следующую за содержащей  $\mathbf{R}$  ячейку записывается символ из ячейки с номером  $i$  на соответствующей рабочей ленте. Докажите, что язык, распознаваемый на такой машине за  $T(n)$ , лежит в  $\mathbf{DTIME}((T(n))^2)$ .

**1.9. Палиндромы.** Пусть  $x^{\mathbf{R}}$  есть слово  $x$ , записанное в обратном порядке. Докажите, что язык  $\text{PAL} = \{x \in \{0, 1\}^* \mid x^{\mathbf{R}} = x\}$ , т.е. множество палиндромов,

лежит в классе **DTIME**( $n$ ), но не распознаётся за время  $o(n^2)$  на одноленточной машине Тьюринга.

## Глава 2

# Классы P, NP и coNP

### 2.1 Что называть быстрым алгоритмом?

В прошлой главе мы условились измерять время работы программы как асимптотику числа её шагов в худшем случае. Но какое время считать хорошим, доступным для реализации на настоящих машинах? Точный ответ зависит от приложений. Так, программа, анализирующая граф всего интернета, может позволить себе фактически только линейное время работы. Возможно, подойдёт  $O(n \log n)$ , но никак не  $O(n^2)$ : такая программа будет попросту не успевать за изменением графа. С другой стороны, не слишком большие системы линейных уравнений успешно решаются алгоритмами кубической сложности. Для общей теории хотелось бы иметь фундамент, не зависящий от деталей задачи и тонкостей реализации. В частности, класс быстрых алгоритмов не должен зависеть от точной спецификации вычислительной модели. В качестве такого класса был выбран класс алгоритмов с полиномиальным временем работы. Искомая независимость от модели достигается за счёт тезиса Кобхэма–Эдмондса: при переходе к другой модели время работы может увеличиться в полиномиальное число раз, но полином от полинома остаётся полиномом. Кроме того, желательно, чтобы композиция эффективных алгоритмов была эффективным алгоритмом. Класс полиномов — минимальный класс, содержащий линейные функции и замкнутый относительно композиции. Конечно, алгоритм со временем работы  $O(n^{100})$  или с очень большой константой в  $O(\cdot)$ -обозначении на практике не реализуем, но настолько большие степени полинома крайне редко встречаются в реальных алгоритмах. Более того, если открыт алгоритм с большим полиномиальным временем работы, то обычно в последующих работах степень уменьшается до приемлемых значений. (Например, знаменитый AKS-алгоритм проверки простоты изначально имел сложность  $\tilde{O}(n^{12})$  для  $n$ -значного числа, а затем был улучшен до  $\tilde{O}(n^6)$ ). Однако есть и следующие ступени (не)доступности. Так, для нескольких задач неизвестно полиномиального алгоритма, но известен квазиполиномиальный, работающий за время  $O(n^{\text{polylog } n})$ . В этой связи стоит отметить недавнее достижение Ласло Бабаи, построившего квазиполиномиальный алгоритм для проверки изоморфности двух графов [13].<sup>1</sup>

---

<sup>1</sup>4 января 2017 года Бабаи отозвал своё утверждение о квазиполиномиальности, заменив на более скромную субэкспоненциальную оценку. Однако уже 9 января квазиполиномиальность была восстановлена.

## 2.2 Детерминированные сложностные классы

### 2.2.1 Класс P

Классом **P** называется множество языков, распознающихся за полиномиальное время. Формально он определяется так:

**Определение 2.1.**  $P = \bigcup_{c=1}^{\infty} DTIME(n^c)$ .

Можно упрощённо записать это определение как  $P = DTIME(\text{poly}(n))$ . Примерами языков из **P** являются такие множества:

- $GCD = \{(a, b, d) \mid \text{число } d \text{ является наибольшим общим делителем чисел } a \text{ и } b\}$ . Действительно, наибольший общий делитель можно найти алгоритмом Евклида и затем сравнить результат с  $d$ .
- $PATH = \{(G, s, t) \mid \text{в графе } G \text{ есть путь из } s \text{ в } t\}$ . Наличие пути в графе проверяется быстро, например обходом в ширину.
- $CONNECTED = \{G \mid G \text{ — связный граф}\}$ . Например, можно проверить граф на связность, проверив наличие пути из некоторой фиксированной вершины во все остальные, или использовать обход графа одновременно.
- $EULERCYCLE = \{G \mid \text{в графе } G \text{ есть эйлеров цикл}\}$ . По критерию существования эйлерова цикла достаточно проверить связность, а также посчитать степени всех вершин и проверить, что все они чётные.
- $BIPARTITE = \{G \mid \text{граф } G \text{ двудольный}\}$ . Иными словами, вершины графа  $G$  можно покрасить в 2 цвета, так чтобы вершины одного цвета не были соединены ребром. В данном случае подойдёт «жадный» алгоритм: покрасим одну вершину произвольно, дальше будем красить соседние с уже покрашенными согласно условию, пока не придём к противоречию или не покрасим всю компоненту связности. В последнем случае продолжим тем же алгоритмом, пока все компоненты не кончатся. Если в какой-то момент случилось противоречие, значит граф содержит нечётный цикл, а тогда он не является двудольным.
- $MINCUT = \{(G, k) \mid \text{вершины графа } G \text{ можно разбить на два множества } S \text{ и } T, \text{ так что между частями проведено не более } k \text{ рёбер}\}$ . Эта задача решается широко известным алгоритмом Форда–Фалкерсона с использованием теоремы о максимальном потоке и минимальном разрезе.

Обратите внимание, что если в постановку задачи входит числовой параметр  $n$ , то полином считается не от этого параметра, а от его логарифма, т.к. на запись числа  $n$  требуется  $\log n$  битов. Если алгоритм работает полиномиальное время от самого  $n$ , то такой алгоритм называется *псевдополиномиальным*. Существуют задачи, для которых не известно полиномиального алгоритма, но известен псевдополиномиальный. Например, такими задачами являются задача о рюкзаке

$$KNAPSACK = \{(n_1, \dots, n_k, m_1, \dots, m_k, N, M) \mid \exists \alpha \in \{0, 1\}^k \sum \alpha_i n_i \leq N \text{ и } \sum \alpha_i m_i \geq M\}$$

и её частный случай

$$SUBSET-SUM = \{(n_1, \dots, n_k, N) \mid \exists \alpha \in \{0, 1\}^k \sum \alpha_i n_i = N\}.$$

Интерпретация задачи о рюкзаке такая: имеются предметы с весами  $n_i$  и стоимостями  $m_i$ . Вопрос: можно ли взять в рюкзак несколько предметов, так чтобы их суммарный вес не превысил  $N$  (иначе рюкзак порвётся), а суммарная стоимость была не меньше  $M$ . В задаче SUBSET-SUM остаётся один параметр: ставится вопрос о том, можно ли заполнить рюкзак, не оставляя свободного места.

Нетрудно заметить следующие соотношения для класса  $\mathbf{P}$ :

**Теорема 2.2.** Если  $L \in \mathbf{P}$  и  $M \in \mathbf{P}$ , то  $\bar{L}$ ,  $L \cup M$ ,  $L \cap M$  также лежат в  $\mathbf{P}$ .

Иначе говоря, класс  $\mathbf{P}$  замкнут относительно операций дополнения, объединения, пересечения.

*Доказательство.* Действительно, имея ответ на вопрос, принадлежит ли слово  $x$  языкам  $L$  и  $M$ , простейшими логическими операциями (отрицание, дизъюнкция, конъюнкция) можно получить ответы на вопросы о принадлежности слова  $x$  языкам  $\bar{L}$ ,  $L \cup M$ ,  $L \cap M$  соответственно. Эти операции займут совсем немного времени, поэтому время работы останется полиномиальным. Более точно, если  $L \in \mathbf{DTIME}(T(n))$ , то  $\bar{L} \in \mathbf{DTIME}(T(n))$ , а если  $L \in \mathbf{DTIME}(T(n))$  и  $M \in \mathbf{DTIME}(S(n))$ , то  $L \cup M$  и  $L \cap M$  лежат в  $\mathbf{DTIME}(\max\{T(n), S(n)\})$ .  $\square$

### 2.2.2 Более высокие классы

Помимо класса  $\mathbf{P}$  изучают другие классы, определяемые порядком роста времени работы на машине Тьюринга:

- $\mathbf{QP} = \mathbf{DTIME}(2^{\text{polylog}(n)}) = \bigcup_{c=1}^{\infty} \mathbf{DTIME}(2^{\log^c n})$  (квазиполиномиальное время);
- $\mathbf{SUBEXP} = \mathbf{DTIME}(2^{n^{o(1)}}) = \bigcap_{\varepsilon > 0} \mathbf{DTIME}(2^{n^\varepsilon})$  (субэкспоненциальное время);
- $\mathbf{E} = \mathbf{DTIME}(2^{O(n)}) = \bigcup_{c=1}^{\infty} \mathbf{DTIME}(2^{cn})$  (линейно-экспоненциальное время);
- $\mathbf{EXP} = \mathbf{DTIME}(2^{\text{poly}(n)}) = \bigcup_{c=1}^{\infty} \mathbf{DTIME}(2^{n^c})$  (экспоненциальное время, иногда его также называют  $\mathbf{EXPTIME}$ );
- $\mathbf{EE} = \mathbf{DTIME}(2^{2^{O(n)}}) = \bigcup_{c=1}^{\infty} \mathbf{DTIME}(2^{2^{cn}})$  (дважды линейно-экспоненциальное время);
- $\mathbf{EEXP} = \mathbf{DTIME}(2^{2^{\text{poly}(n)}}) = \bigcup_{c=1}^{\infty} \mathbf{DTIME}(2^{2^{n^c}})$  (дважды экспоненциальное время);
- и т.д.

Очевидно, что  $\mathbf{P} \subset \mathbf{QP} \subset \mathbf{SUBEXP} \subset \mathbf{E} \subset \mathbf{EXP} \subset \mathbf{EE} \subset \mathbf{EEXP}$ . Из теоремы об иерархии следует, что все эти вложения строгие. Доказательство этой теоремы мы пока отложим. Стоит также обратить внимание на то, что  $\mathbf{SUBEXP}$  в отличие от остальных классов определяется не через объединение, а через пересечение различных классов. Это приводит к потенциальной неоднородности: может не быть единого алгоритма, время работы которого растёт медленнее любой функции  $2^{n^\varepsilon}$ , но для каждого  $\varepsilon$  найдётся свой такой алгоритм.

### 2.2.3 Неконструктивные оценки сложности

*Неконструктивные доказательства существования извещают мир о том, что сокровище существует, не указывая при этом его местонахождение, т.е. не позволяя это сокровище использовать. Такие доказательства не могут заменить построение — подмена конструктивного доказательства неконструктивным влечёт утрату смысла и значения самого понятия «доказательства».*

Герман Вейль, *Континуум*

Как правило, включение задачи в тот или иной класс, например,  $P$ , доказывается предъявлением алгоритма соответствующей сложности. Может показаться, что никакого другого способа и быть не может. Тем удивительнее, что бывает иначе. Есть целый класс задач, лежащих в  $P$ , для которых неясно, как построить конкретный алгоритм! Они связаны с топологическими свойствами графов.

Известен критерий Понтрягина–Куратовского планарности графов: граф планарен тогда и только тогда, когда в нём нет подграфа, гомеоморфного  $K_5$  или  $K_{3,3}$  (т.е. полному подграфу из 5 вершин или полному двудольному подграфу с двумя долями по 3 вершины). Наличие таких подграфов проверяется полиномиальным перебором: например, для  $K_5$  нужно перебрать все наборы из 5 вершин и проверить наличие всех путей между всеми 10 парами. Алгоритм будет полиномиален, поскольку нужно перебрать  $C_n^5 = O(n^5)$  вариантов, и проверка на наличие пути также полиномиальна. Ясно, что аналогично можно проверить вложенность любого конкретного подграфа. Заметим, что для проверки планарности есть и более эффективные алгоритмы, например, линейный алгоритм Хопкрофта–Тарджана (см. [75, 85], а также [151, с. 175–183]).

Планарность графа не теряется, если из него удалить ребро или если ребро стянуть. На этом основан критерий планарности Вагнера: граф планарен тогда и только тогда, когда в нём нет *миноров*  $K_5$  или  $K_{3,3}$ . Минором как раз называется граф, полученный последовательностью удалений и стягиваний рёбер. Легко понять, что если у графа есть подграф, гомеоморфный  $H$ , то у него есть и минор  $H$ . Обратное не всегда верно, тем не менее в критерии Вагнера и критерии Понтрягина–Куратовского набор запрещённых графов одинаков.

Многие другие топологические свойства также сохраняются при переходе к минорам: вложимости графа без самопересечений на какую-нибудь двумерную поверхность, размещение его в пространстве, так чтобы все циклы были не заузлены и/или не зацеплены друг с другом, и т.д. Гипотеза Вагнера заключалась в том, что для любого такого свойства существует конечное число запрещённых миноров. В огромной серии из 23 статей, выходившей с 1983 [112] по 2012 [115] годы,<sup>2</sup> Нил Робертсон и Пол Сеймур разработали теорию графовых миноров, где среди прочего построили полиномиальный алгоритм проверки, есть ли в графе данный минор [113] и доказали гипотезу Вагнера [114]. Тем самым, для всех таких топологических свойств существуют полиномиальные алгоритмы. Вот только теорема не даёт явного списка запрещённых миноров, и потому непонятно, как именно устроен алгоритм и какая там степень полинома. Даже для вложимости в тор полный список запрещённых миноров пока нет. Некоторые из них нарисованы в [143].

В 2013 году Берт Джерардз, Джим Джилин и Джефф Уиттл заявили, что доказали аналогичную теорему в более общем случае, а именно для замкнутых

<sup>2</sup>Статья [115] имеет номер 22, но 23-я статья вышла раньше.



относительно взятия миноров классов матроидов, представимых над конечным полем. Здесь также есть лишь конечное число запрещённых миноров, и вхождение каждого из них проверяется за полиномиальное время. Стоит отметить, что теорема о конечном числе запрещённых миноров есть и для самого свойства представимости над данным полем — в этом заключалась гипотеза Роты. Но в этом случае нет теоремы о полиномиальном алгоритме; более того, при некоторой спецификации модели его точно нет. На текущий момент (сентябрь 2017 года) полные доказательства ещё не опубликованы. Базовые идеи и точные формулировки написаны в [55], см. также популярное изложение [156].<sup>3</sup>

## 2.3 Класс NP

### 2.3.1 Недетерминированные машины Тьюринга

Недетерминированные вычисления — важная теоретическая концепция, хотя и не реализуемая в виде физического устройства.<sup>4</sup> Неформально говоря, недетерминизм позволяет проводить перебор экспоненциальных семейств слов за один шаг, сразу «угадывая» нужное слово. Формально недетерминированные вычисления проводятся на недетерминированной машине Тьюринга.

**Определение 2.3.** *Недетерминированной машиной Тьюринга с  $k$  лентами называется кортеж  $\langle \Sigma, \Gamma, Q, q_0, q_a, q_r, k, \delta \rangle$ , где  $\Sigma$ ,  $\Gamma$  и  $Q$  суть конечные непустые множества, причём  $\Sigma \subset \Gamma$  и  $\Gamma \cap Q = \emptyset$ ,  $q_0, q_a, q_r \in Q$  и попарно различны, а  $\delta: (Q \setminus \{q_a, q_r\}) \times \Gamma^k \rightrightarrows Q \times \Gamma^k \times \{L, N, R\}^k$  — всюду определённая многозначная функция.*

Таким образом, единственным отличием от детерминированной машины является многозначность функции перехода  $\delta$ . Гораздо сильнее отличие в определении того, что такое вычисление и что означает распознавание языка. Суть состоит в том, что из каждой конфигурации машина может перейти не в какую-то одну следующую, а в любую совместимую с функцией перехода. Формально это определяется так:

**Определение 2.4.** Пусть  $C = (a_1\sigma_1, \dots, a_k\sigma_k; \tau_1b_1, \dots, \tau_kb_k; q)$  и  $C' = (a'_1, \dots, a'_k; b'_1, \dots, b'_k; r)$  — некоторые конфигурации, где  $q \neq q_a$  и  $q \neq q_r$ . Тогда переход от  $C$  к  $C'$  *допустим*, если среди значений  $\delta(q, \tau_1, \dots, \tau_k)$  найдётся  $(r, \rho_1, \dots, \rho_k, D_1, \dots, D_k)$ , такое что выполнено одно из трёх:

- $D_i = L$ ,  $a'_i = a_i$  (или  $a'_i = \#$ , если  $a_i = \varepsilon$ ) и  $b'_i = \sigma_i\rho_ib_i$ ;
- $D_i = N$ ,  $a'_i = a_i\sigma_i$  и  $b'_i = \rho_ib_i$ ;
- $D_i = R$ ,  $a'_i = a_i\sigma_i\rho_i$  и  $b'_i = b_i$  (или  $b'_i = \#$ , если  $b_i = \varepsilon$ ).

<sup>3</sup>Насколько известно автору, утверждения о полиномиальных алгоритмах не опубликованы ни в каком виде, о них Джим Джилин рассказывал в личных сообщениях.

<sup>4</sup>Некоторые полагают, что недетерминизм можно моделировать на квантовом компьютере. Это распространённое заблуждение, сродни тому что «в геометрии Лобачевского параллельные прямые пересекаются». К сожалению, этому заблуждению порой подвержены и серьёзные учёные. Так, Дэвид Дойч ошибочно использовал такой довод в пользу своей теории множественности миров: «Если параллельных миров нет, то откуда же квантовые компьютеры черпают свою силу?» Здесь подразумевается, что в параллельных мирах производится экспоненциальный перебор, но квантовые компьютеры устроены не так и проводить перебор не умеют.

Не следует думать, что машина выбирает следующую конфигурацию случайно. Лучше представлять, что машина каким-то образом разделяется на несколько копий, каждая из которых выбирает свой переход. На следующем шаге каждая из копий ещё подразделяется, и так далее. В результате каждая копия производит своё вычисление.

**Определение 2.5.** Цепочка конфигураций  $(C_1, \dots, C_n)$  называется *вычислением*, если все переходы от  $C_i$  к  $C_{i+1}$  допустимы.

Удобно считать, что все вычисления собираются в одно укоренённое дерево: корнем будет начальная конфигурация, а непосредственными потомками каждой вершины — те конфигурации, переходы в которые допустимы. Как обычно, будем считать, что машина, получившая на вход слово  $x$ , начинает вычисление в конфигурации  $(\underbrace{\#, \dots, \#}_{k \text{ раз}}; x, \underbrace{\#, \dots, \#}_{k-1 \text{ раз}}; q_0)$ . Ниже мы будем просто писать, что вычисление начинается с  $x$ . Теперь надо определить, как машина собирает исходы всех вычислений в один результат своей работы. Неформально говоря, если одна из копий пришла в принимающая состояние, то она сразу сигнализирует об этом в управляющий центр. Если центр получил хотя бы один такой сигнал за отведённое время, то он принимает вход, иначе отвергает. Формально определение следующее.

**Определение 2.6.** Недетерминированная машина  $M$  *распознаёт* язык  $L$  за время  $T(n)$ , если при всех  $x \in \{0, 1\}^*$  верно следующее:

- Любое вычисление, начинающееся с  $x$ , не длиннее  $T(|x|)$  (иначе говоря, машина в любом случае останавливается не более чем за  $T(|x|)$  шагов);
- Если  $x \in L$ , то найдётся вычисление, начинающееся с  $x$  и заканчивающееся в состоянии  $q_a$ ;
- Если  $x \notin L$ , то любое вычисление, начинающееся с  $x$ , заканчивается в состоянии  $q_r$ .

Недетерминированные вычисления позволяют определить соответствующий сложностной класс:

**Определение 2.7.** Классом  $\text{NTIME}(T(n))$  называется множество языков, распознаваемых на недетерминированной машине Тьюринга за время  $O(T(n))$ .

**Замечание 2.8.** Некоторые авторы говорят не о многозначной, а о двузначной функции перехода  $\delta$ , или о двух функциях  $\delta_0$  и  $\delta_1$ . От такой вариации время работы изменится в константу раз, поскольку множество значений функции  $\delta$  имеет константный (т.е. зависящий не от длины входа, а только от параметров машины) размер, поэтому выбор нужного значения можно смоделировать как константное число последовательных бинарных выборов. Точный размер множества значений —  $|Q| \cdot |\Gamma|^k \cdot 3^k$ .

Легко установить следующую связь между детерминированными и недетерминированными вычислениями:

**Теорема 2.9.**  $\text{DTIME}(T(n)) \subset \text{NTIME}(T(n)) \subset \text{DTIME}(2^{O(T(n))})$ .

Как и раньше, запись в правой части толкуется как  $\bigcup_{c=1}^{\infty} \text{DTIME}(2^{cT(n)})$ . В дальнейшем мы также будем пользоваться подобными сокращениями.

*Доказательство.* Первое вложение следует из того, что обычная машина Тьюринга — частный случай недетерминированной. Для второго вложения вспомним, что у функции перехода может быть не больше некоторой константы  $K$  значений. Значит, общее количество конфигураций во всех вычислениях, начинающихся с данного входа и имеющих длину не больше  $T$ , не превышает  $K^T = 2^{O(T)}$ . За такое же время можно все эти конфигурации перебрать (например, обходя дерево) и проверить, встречается ли среди них состояние  $q_a$ . Если встречается, то данный вход лежит в языке, иначе не лежит.  $\square$

### 2.3.2 Два определения класса NP и их эквивалентность

Класс NP важен как для теории, так и для практики. У него есть два эквивалентных определения: через недетерминированные машины и через сертификаты.

**Определение 2.10.**  $\text{NP} = \bigcup_{c=1}^{\infty} \text{NTIME}(n^c)$ .

Это определение объясняет название NP: N — недетерминированные вычисления, P — полиномиальное время.<sup>5</sup>

**Определение 2.11.** Классом NP называется множество языков  $L$ , для которых существует функция  $V(x, s)$  с булевыми значениями, вычисляемая за полиномиальное время от длины первого аргумента, такая что:

- Если  $x \in L$ , то  $\exists s V(x, s) = 1$ ;
- Если  $x \notin L$ , то  $\forall s V(x, s) = 0$ .

Второй вход  $s$  часто называют *сертификатом*, а функцию  $V$  — *верификатором*. Таким образом, сертификат удостоверяет, что  $x \in L$ , а верификатор проверяет верность сертификата. Для слов из языка подходящий сертификат должен существовать, а для слов не из языка все сертификаты должны отвергаться. Иначе говоря, класс NP — это класс языков, принадлежность к которым можно быстро доказать, а P — класс языков, принадлежность к которым можно быстро выяснить.

**Замечание 2.12.** Оговорка о том, что время работы  $V$  полиномиально именно от длины  $x$ , важна, иначе в сертификат можно было бы включить слишком много информации и сделать любое время работы полиномиальным. Альтернативно можно потребовать, чтобы алгоритм  $V$  был полиномиальным от своего входа целиком, но вот длина сертификата была бы ограничена полиномом от длины  $x$ .

**Теорема 2.13.** Определения 2.10 и 2.11 эквивалентны, т.е. задают один и тот же класс языков.

*Доказательство.* Пусть  $L$  распознаётся недетерминированной машиной  $M$ , которая работает  $O(n^c)$  шагов. Тогда в качестве сертификата можно взять последовательность значений функции перехода, а верификатор будет моделировать работу машины  $M$ , используя данные сертификата для выбора одной из ветвей алгоритма. Можно действовать и по-другому: в качестве сертификата взять всё вычисление, а верификатором проверять его соответствие программе.

<sup>5</sup>Тут тоже встречается популярная ошибка: люди считают, что раз P означает «полиномиальное время», то NP — «неполиномиальное». Это, конечно, не так, ведь  $P \subset NP$ .

Пусть, напротив, для  $L$  верно сертификатное определение. Тогда сертификат не может быть длиннее, чем время работы  $V$ , т.е. чем некоторый полином  $p(|x|)$ . Недетерминированная машина будет работать следующим образом: сначала она недетерминированно напишет сертификат  $s$  длины не больше  $p(|x|)$ , а затем запустит  $V(x, s)$ .  $\square$

### 2.3.3 Примеры задач из NP

Примерами языков из **NP** служат такие задачи:

- **SAT** =  $\{\varphi \mid \varphi \text{ — выполнимая булева формула}\}$ . (Выполнимость означает, что формула равна 1 на некотором наборе значений). В данном случае сертификатом будет выполняющий набор, а верификатор проверит, что значение формулы на этом наборе действительно равно 1. Важным частным случаем является задача **3SAT**, в которой про формулу  $\varphi$  дополнительно сказано, что она представлена в виде 3-КНФ, т.е. КНФ, в которой каждый дизъюнкт включает в себя 3 литерала.
- **CLIQUE** =  $\{(G, k) \mid \text{в графе } G \text{ найдётся полный подграф хотя бы на } k \text{ вершинах}\}$ . Сертификатом будет сама клика.
- **3COL** =  $\{G \mid \text{вершины графа } G \text{ можно правильно раскрасить в 3 цвета}\}$ . Сертификатом будет раскраска.
- **HAMCYCLE** =  $\{G \mid \text{в графе } G \text{ существует гамильтонов цикл}\}$ . (Гамильтонов цикл проходит ровно 1 раз через каждую вершину). Сертификатом будет сам цикл.
- **KNAPSACK, SUBSET-SUM** (см. выше). Сертификатом будет набор  $\alpha$ .
- **FACTORING** =  $\{(N, a, b) \mid \text{у числа } N \text{ существует простой делитель на отрезке } [a, b]\}$ . Сертификатом будет этот делитель.
- **MAXCUT** =  $\{(G, k) \mid \text{вершины графа } G \text{ можно разбить на два множества } S \text{ и } T, \text{ так что между частями проведено не менее } k \text{ рёбер}\}$ . Сертификатом будет это разбиение.

Задачи из класса **NP** встречаются очень часто в дискретной математике, алгебре, логике, дискретной оптимизации и других разделах «конечной математики».

Как и класс **P**, класс **NP** замкнут относительно объединения и пересечения языков (но не дополнения!). Более того, можно доказать такую общую теорему:

**Теорема 2.14.** Если  $L \in \mathbf{NTIME}(T(n))$ , а  $M \in \mathbf{NTIME}(S(n))$ , то  $L \cup M$  и  $L \cap M$  лежат в  $\mathbf{NTIME}(\max\{T(n), S(n)\})$ .

*Доказательство.* Недетерминированная машина должна сначала провести вычисление для  $L$ , затем для  $M$ , а затем взять дизъюнкцию или конъюнкцию результатов для  $L \cup M$  и  $L \cap M$  соответственно. В случае с дизъюнкцией для элементов  $L \cup M$  хотя бы на одной ветви хотя бы одного из этапов ответ будет положительным, поэтому на соответствующей ветви общего алгоритма ответ будет положительный, а для слов не из  $L \cup M$  на всех ветвях обоих этапов будет отрицательный ответ, поэтому и итоговый ответ будет отрицательным. В случае с конъюнкцией для элементов  $L \cap M$  на какой-то ветви на каждом этапе ответ будет положительным, поэтому на составной ветви ответ также будет

положительным. Для слов не из  $L \cap M$  на каком-то из этапов ответы на всех ветвях будут отрицательными, поэтому и итоговый ответ будет отрицательным.

Время работы будет примерно равно сумме  $O(T(n))$  и  $O(S(n))$ , что также представимо в виде  $O(\max\{T(n), S(n)\})$ .  $\square$

## 2.4 Проблема перебора ( $P \stackrel{?}{=} NP$ )

Мы уже выяснили, что  $DTIME(T(n)) \subset NTIME(T(n))$ , откуда  $P \subset NP$ . Также из теоремы 2.9 следует, что  $NP \subset EXP$ . Вскоре мы докажем, что хотя бы одно из этих вложений строгое, т.к.  $P \neq EXP$ . Однако вопрос о том, являются ли строгими оба вложения, открыт. Наибольшее внимание привлекает вопрос о равенстве классов  $P$  и  $NP$ . Он также называется проблемой перебора, потому что может быть сформулирован так: возможно ли смоделировать экспоненциальный перебор возможных сертификатов за полиномиальное время? Существенная часть теории сложности вычислений и подавляющая часть теоретической криптографии разработаны в предположении  $P \neq NP$ . Однако это не единственное недоказанное предположение. Существуют и другие: неколлапсирование полиномиальной иерархии, существование односторонней функции и др. Исходный вопрос поставлен в начале 1970-х Стивеном Куком и Леонидом Левиным, однако до сих пор все существенные продвижения лишь дополнительно показывали, что вопрос сложный:

- В оригинальных работах Кук [35] и Левин [162] независимо друг от друга открыли класс  $NP$ -полных задач, или универсальных задач перебора, «самых сложных» в классе  $NP$ . В настоящее время известны тысячи таких задач, с вариациями — десятки тысяч. Нахождение полиномиального алгоритма для любой из них привело бы к тому, что  $P = NP$ , однако многолетние усилия огромного числа учёных по поиску такого алгоритма успехом не увенчались. Подробно про  $NP$ -полные задачи мы поговорим в следующей главе.
- В 1975 году Бейкер, Джилл и Соловей [15] доказали, что проблема  $P \stackrel{?}{=} NP$  не релятивизируется, т.е. существует оракул  $A$ , такой что  $P^A = NP^A$ , и существует оракул  $B$ , такой что  $P^B \neq NP^B$ . (Подробнее о вычислениях с оракулом и этой теореме мы поговорим в разделе 4.3). Это значит, что любое правильное доказательство должно быть нерелятивизуемым, а примеров таких доказательств не так много.
- В 1994 году Разборов и Рудич [109] ввели понятие естественного доказательства и доказали следующий факт: если верно некоторое усиление  $P \neq NP$ , то само утверждение  $P \neq NP$  невозможно доказать при помощи естественных доказательств. На совсем неформальном уровне идея такова: утверждение  $P \neq NP$  говорит как раз о том, что поиск доказательства гораздо сложнее, чем его проверка. Но это соображение можно применить и к самому утверждению  $P \neq NP$ ! Именно поэтому мы не можем найти доказательства этого утверждения. Разборов и Рудич уточняют эту идею: если некоторая техника позволяет доказать, что некоторая задача сложная, то эту же технику можно развернуть для доказательства, что некоторая другая задача простая. А простота этой задачи и будет противоречить упомянутому усилению  $P \neq NP$ .

- В 2008 году Ааронсон и Вигдерсон [5] доказали, что гипотетическое доказательство  $P \neq NP$  не может использовать «алгебраические техники», что дополнительно исключило многие известные методы. Формально они доказали аналог результата Бейкера–Джилла–Соловея для оракулов специального вида, названных алгебраическими.

Помимо вариантов совпадения и несовпадения классов  $P$  и  $NP$  есть и экзотический вариант: утверждение  $P = NP$  не зависит от остальных аксиом, подобно аксиоме выбора или континуум-гипотезе. С одной стороны, имеется несколько результатов (например, [38]), показывающих независимость утверждения  $P = NP$  от некоторых слабых аксиоматических теорий. С другой стороны, в работе [20] показано, что если  $P = NP$  не зависит от арифметики Пеано, то  $P$  и  $NP$  «почти равны». А именно,  $NP$  вложено в  $DTIME(n^{\log^* n})$ . (Функция  $\log^* n$  равна количеству итераций логарифмирования, которое нужно применить к  $n$ , чтобы получить число меньше 2. Для всех реальных значений  $n$  эта функция не превосходит 5, хотя асимптотически и стремится к бесконечности.) Более подробный обзор и предварительные сведения из логики можно найти в статье Ааронсона [1]. Теоретически возможны и другие «странные» варианты, связанные с тем, что  $NP \subset SUBEXP$ .

При этом из-за того, что за решение проблемы перебора Институтом Клэя назначена премия в миллион долларов, вопрос привлекает внимание многих непрофессионалов, в результате несколько раз в год появляются заявления о том, что проблема решена в ту или иную сторону. Как правило, найти ошибку в рассуждениях не составляет большого труда, но случаются и более изощрённые попытки, например, работа Виная Деолаликара 2010 года, вызвавшая переполох в среде теоретиков. Сообщество быстро самоорганизовалось, пристально изучило стостраничную статью и нашло «дырку», которая пока не заделана.

Также многие обыватели, завороженные словосочетанием «квантовый компьютер», ошибочно полагают, что изобретение квантового компьютера позволит решать все задачи из  $NP$  за полиномиальное время. Однако на современном уровне знаний это не так: про задачу о разложении числа на множители (которую быстро решает квантовый компьютер) не доказана  $NP$ -полнота, а для  $NP$ -полных задач не известно квантовых алгоритмов.

Вопрос о равенстве  $P$  и  $NP$  настолько важен, что в 1989 году его увековечили в камне на стене факультета компьютерных наук принстонского университета (см. рис. 2.1). Углубления в стене можно прочесть как « $P=NP?$ » в 7-битной ASCII-кодировке. Такая кодировка давно не используется, а теоретический вопрос всё так же стоит.

## 2.5 Класс coNP

В отличие от класса  $P$ , для класса  $NP$  скорее всего не верна замкнутость относительно дополнения. Действительно, непонятно, как можно быстро доказать отсутствие подходящего сертификата. Дополнения языков из  $NP$  объединяют в отдельный класс  $coNP$ :

**Определение 2.15.**  $coNP = \{L \mid \bar{L} \in NP\}$ .

Иначе говоря, если к  $NP$  относятся языки, принадлежность к которым легко доказать, то  $coNP$  — класс языков, принадлежность к которым легко опровергнуть. Самым естественным представителем класса  $coNP$  является язык





Рис. 2.1: Кирпичная кладка на стене факультета компьютерных наук принстонского университета. Каждый отсутствующий кирпич обозначает единицу, присутствующий — ноль. Строки кодируют в семибитной ASCII-записи вопрос « $P=NP?$ ».

тавтологий  $\text{TAUT} = \{\varphi \mid \varphi \text{ — тавтология}\}$ . Действительно, опровергнуть, что  $\varphi$  тавтология, очень легко: нужно предъявить набор значений, на котором формула ложна. Доказать, что  $\varphi$  тавтология, можно, предъявив вывод в исчислении высказываний, однако такой вывод может быть слишком длинным: длиннее, чем полином от длины  $\varphi$ . Коротких доказательств тавтологичности не известно.

Легко показать, что  $\mathbf{P} \subset \text{coNP} \subset \mathbf{EXP}$ . Таким образом,  $\mathbf{P} \subset \mathbf{NP} \cap \text{coNP}$ . Про это вложение тоже неизвестно, строго ли оно, однако большинство исследователей верят, что строго. В частности, задача  $\text{FACTORING}$  лежит в  $\mathbf{NP} \cap \text{coNP}$ : доказать наличие делителя в нужном интервале можно, просто предъявив его, а опровергнуть — приведя полное разложение на простые множители. Имея полиномиальный алгоритм проверки простоты, это разложение можно проверить и таким образом убедиться в отсутствии простых множителей в заданном интервале.<sup>6</sup> При этом научиться решать  $\text{FACTORING}$  достаточно для последую-

<sup>6</sup>Впрочем, можно обойтись и без полиномиального теста простоты. Достаточно, что само множество простых чисел лежит в  $\mathbf{NP} \cap \text{coNP}$ . Принадлежность  $\text{coNP}$  очевидна: для составного числа предъявляется разложение на множители, — а простота удостоверяется так:  $n$  простое тогда и только тогда, когда мультипликативная группа  $\mathbb{Z}_n^*$  циклическая. В сертификат нужно включить генератор  $g$  этой группы и доказательство её цикличности. Это доказательство заключается в предъявлении разложения  $n - 1$  на простые множители  $p_i$ , тогда можно проверить  $g^{n/p_i} \neq 1$ . А простота этих множителей удостоверяется подобными же сертификатами индуктивно. Также индуктивно доказывается, что общая длина всех сертификатов полиномиальна.

щего *поиска* разложения на множители, достаточно воспользоваться двоичным поиском. А эта задача, по всей видимости, не решается за полиномиальное время.

Также легко показать такую связь:

**Теорема 2.16.** *Если  $P = NP$ , то  $NP = coNP = P$ .*

*Доказательство.* Действительно, пусть  $L \in coNP$ . Значит,  $\bar{L} \in NP$ . Поскольку мы предположили, что  $P = NP$ , то  $\bar{L} \in P$ . Поскольку класс  $P$  замкнут относительно дополнения, то и  $L \in P$ . Значит,  $coNP \subset P$ . Поскольку  $P \subset coNP$ , имеем  $P = coNP$ , а значит и  $NP = coNP$  в предположении  $P = NP$ .  $\square$

В частности, если  $P = NP$ , то  $NP$  всё-таки замкнут относительно дополнения. Об истинности обратной теоремы ничего не известно: из  $NP = coNP$  автоматически не следует  $P = NP$ . В частности, для некоторого оракула  $A$  выполнено  $NP^A = coNP^A$ , но  $P^A \neq NP^A$ .

## 2.6 Класс NEXP

Для недетерминированных вычислений также можно определять высшие классы, например  $NE = NTIME(2^{O(n)})$  и  $NEXP = NTIME(2^{\text{poly}(n)})$ . Вопрос  $P \stackrel{?}{=} NP$  можно перемасштабировать в вопрос  $EXP \stackrel{?}{=} NEXP$ . Между ответами можно установить следующую связь:

**Теорема 2.17.** *Если  $P = NP$ , то  $EXP = NEXP$ .*

*Доказательство.* Поскольку  $EXP \subset NEXP$ , достаточно доказать  $NEXP \subset EXP$ . В предположении  $NP \subset P$  это доказывается при помощи метода «раздутия» (padding argument). Интуитивно, экспоненциальное время работы можно превратить в полиномиальное, если добавить в аргумент экспоненциальное количество ничего не значащего «мусора».

Формально, пусть  $A \in NEXP$ , причём  $A$  распознаётся некоторой недетерминированной машиной  $M$  за время  $2^{n^c}$ . Тогда язык  $\tilde{A} = \{x01^{2^{n^c}} \mid n = |x|, x \in A\}$  лежит в  $NP$ . Действительно, можно сначала легко проверить, что аргумент имеет вид  $x01^{2^{n^c}}$ , а затем применить к  $x$  машину  $M$ . Время работы машины  $M$  составит  $2^{n^c}$ , что будет полиномом (и даже линейной функцией) от длины аргумента  $|x01^{2^{n^c}}| = n + 1 + 2^{n^c}$ . В предположении  $NP \subset P$  найдётся детерминированная машина  $M'$ , распознающая  $\tilde{A}$  за время  $\text{poly}(n + 2^{n^c}) = 2^{O(n^c)}$ . Тогда детерминированная машина, приписывающая к слову  $x$  единицы в количестве  $2^{n^c}$  и запускающая  $M'$  на полученном слове, будет распознавать  $A$  за время  $2^{O(n^c)}$ , т.е. за экспоненциальное от длины входа время. Значит,  $A \in EXP$ , что и требовалось.  $\square$

Про истинность обратного утверждения (если  $EXP = NEXP$ , то  $P = NP$ ) ничего не известно. В частности, существует оракул  $A$ , для которого  $EXP^A = NEXP^A$ , но  $P^A \neq NP^A$ .

## 2.7 Исторические замечания и рекомендации по литературе

## 2.8 Задачи и упражнения



**2.1. Другой способ измерить время работы.** Докажите, что класс  $\mathbf{NP}$  не изменится, если в определении через недетерминированные машины вместо полиномиальности любой ветви потребовать полиномиальность минимальной принимающей ветви.

**2.2.**

а) Докажите, что если  $A \in \mathbf{NP}$ , то и  $A^* \in \mathbf{NP}$ .

б) Докажите, что если  $A \in \mathbf{P}$ , то и  $A^* \in \mathbf{P}$ .

**2.3.** Докажите, что если  $A_1$  и  $A_2$  лежат в  $\mathbf{NP} \cap \mathbf{coNP}$ , то  $A_1 \triangle A_2$  тоже лежит в  $\mathbf{NP} \cap \mathbf{coNP}$ .



# Глава 3

## NP-полнота

Теория NP-полноты — краеугольный камень структурной теории сложности, с которого началась эта наука в 1970-х годах. NP-полные задачи — в некотором роде самые сложные в классе NP. Если про какую-то задачу доказана NP-полнота, то можно не надеяться найти её точное и полное решение, лучше искать другие пути.

### 3.1 Полиномиальная сводимость по Карпу

В теории алгоритмов известно два основных вида сводимости одних задач к другим:  $m$ -сводимость и  $T$ -сводимость, или сводимость по Тьюрингу. В теории сложности вычислений им соответствуют сводимость по Карпу и сводимость по Куку. Для нас наиболее важной является полиномиальная сводимость по Карпу.

**Определение 3.1.** Пусть  $A$  и  $B$  суть два языка. Тогда  $A$  сводится по Карпу к  $B$ , если существует всюду определённая функция  $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ , вычисляемая за полиномиальное время, такая что  $x \in A \Leftrightarrow f(x) \in B$ . Обозначение:  $A \leq_p B$ . (Индекс  $p$  означает полиномиальность).

Можно заметить несколько простых свойств такой сводимости:

**Утверждение 3.2.** *Имеют место следующие факты:*

1. Полиномиальная сводимость рефлексивна:  $A \leq_p A$ ;
2. Полиномиальная сводимость транзитивна: если  $A \leq_p B$  и  $B \leq_p C$ , то  $A \leq_p C$ ;
3. Если  $A \in \mathbf{P}$ , а  $B \neq \emptyset$  и  $B \neq \{0, 1\}^*$ , то  $A \leq_p B$ ;
4. Если  $B \in \mathbf{P}$  и  $A \leq_p B$ , то  $A \in \mathbf{P}$ ;
5. Если  $B \in \mathbf{NP}$  и  $A \leq_p B$ , то  $A \in \mathbf{NP}$ .

*Доказательство.* Рефлексивность очевидна: достаточно рассмотреть функцию  $f(x) = x$ . Для транзитивности нужно рассмотреть композицию: если  $f$  сводит  $A$  к  $B$ , а  $g$  сводит  $B$  к  $C$ , то  $g \circ f$  сводит  $A$  к  $C$ :  $x \in A \Leftrightarrow f(x) \in B \Leftrightarrow g(f(x)) \in C$ . При этом если и  $f$ , и  $g$  вычисляются за полиномиальное время, то  $g \circ f$  также вычисляется за полиномиальное время.

Третье утверждение также несложно: если  $B \neq \emptyset$  и  $B \neq \{0, 1\}^*$ , то можно зафиксировать  $b_1 \in B$  и  $b_2 \notin B$  и рассмотреть функцию  $f(x) = \begin{cases} b_1, & x \in A \\ b_2, & x \notin A \end{cases}$ . В

силу разрешимости  $A$  эта функция будет вычислимой. Впрочем: тут имеет место неконструктивность: откуда брать нужные  $b_1$  и  $b_2$ , неясно. Ведь множество  $B$  может быть даже неразрешимым.

Четвёртое утверждение самое важное и полезное, но также несложно. Достаточно заметить, что характеристическая функция  $\chi_A$  представляется как композиция  $\chi_B \circ f$ . Если  $B \in P$ , то  $\chi_B$  вычисляется за полиномиальное время, а если  $A \leq_p B$ , то соответствующая  $f$  также вычисляется за полиномиальное время. Значит, их композиция, т.е.  $\chi_A$ , тоже вычисляется за полиномиальное время, что и означает  $A \in P$ .

Наконец, последнее утверждение докажем двумя способами. Во-первых, для определения через недетерминированные машины достаточно заметить, что если  $\chi_B$  вычисляется недетерминированной машиной за полиномиальное время, а  $f$  (детерминированно) вычисляется за полиномиальное время, то  $\chi_A = \chi_B \circ f$  также вычисляется недетерминированной машиной за полиномиальное время. Во-вторых, для определения через сертификаты: если  $W(y, s)$  будет верификатором принадлежности  $y$  к  $B$ , то  $V(x, s) = W(f(x), s)$  будет верификатором принадлежности  $x$  к  $A$ : если  $x \in A$ , то для  $f(x)$  будет существовать сертификат и  $V$  его примет, а если  $x \notin A$ , то для  $f(x)$  сертификата существовать не будет.  $\square$

## 3.2 NP-трудность и NP-полнота

Неформально говоря, задача трудна для некоторого класса, если её решение позволяет легко решить все задачи из этого класса, и полна для этого же класса, если она и сама лежит в нём. Это определение уточняется следующим образом для сводимости по Карпу:

**Определение 3.3.** Язык  $B$  является **NP-трудным**, если для любого  $A \in \mathbf{NP}$  выполнено  $A \leq_p B$ . Язык  $B$  является **NP-полным**, если он **NP-трудный** и лежит в **NP**.

Имеют место следующие простые утверждения:

**Утверждение 3.4.** Если  $B$  является **NP-трудным** и  $B \in P$ , то  $P = \mathbf{NP}$ .

*Доказательство.* Действительно, если  $A \in \mathbf{NP}$ , то  $A \leq_p B$ . А раз  $B \in P$ , то и  $A \in P$ . Значит,  $\mathbf{NP} \subset P$ , т.е.  $P = \mathbf{NP}$ .  $\square$

**Утверждение 3.5.** Если  $A$  является **NP-трудным** и  $A \leq_p B$ , то  $B$  тоже является **NP-трудным**.

*Доказательство.* Действительно, любой язык  $C$  из **NP** сводится к  $A$ , а по транзитивности и к  $B$ .  $\square$

Несмотря на свою простоту, второе утверждение очень важно, поскольку позволяет получать новые **NP-трудные** задачи: достаточно свести к новой задаче уже известную.

Тривиальным примером **NP-полного** языка является язык  $\mathbf{TMSAT} = \{(M, x, 1^t) \mid \exists y M(x, y) = 1 \text{ и } M(x, y) \text{ заканчивает работу не более чем за } t \text{ шагов}\}$ .

**Теорема 3.6.** Язык **TMSAT** является **NP-полным**.

*Доказательство.* Во-первых, докажем, что  $\text{TMSAT} \in \text{NP}$ . В качестве сертификата выступит тот самый  $y$ , существование которого утверждается в формулировке. При помощи универсальной машины Тьюринга можно проверить, что  $M(x, y) = 1$ , а если при запуске считать шаги, то и условие на остановку также проверяется. При этом, если  $M(x, y)$  сделало больше  $t$  шагов, но не остановилось, то нужно остановить её работу, а сертификат отвергнуть. При тривиальной реализации универсальной машины общее число шагов не превысит  $O(t^2)$ : каждый шаг машины  $M$  моделируется  $O(t)$  шагами на универсальной машине Тьюринга,<sup>1</sup> а поддержка счётчика увеличит число шагов ещё в  $O(\log t)$  раз. Более хитрая реализация позволяет сократить число шагов до  $O(t \log t)$  [10, раздел 1.7]. В любом случае время работы будет полиномом от длины входа, т.е.  $|M| + |x| + t$ . Заметим, что в этом месте важна именно унарная, а не двоичная запись  $t$  в списке аргументов.

Во-вторых, докажем, что если  $L \in \text{NP}$ , то  $L \leq_p \text{TMSAT}$ . Действительно, пусть  $V(x.s)$  есть верификатор принадлежности  $x$  к  $L$ , причём  $V$  вычисляется машиной  $M$ , работающей не дольше  $p(n)$  шагов. Тогда искомой совдимостью будет отображение  $x \mapsto (M, x, 1^{p(|x|)})$ . Действительно, определение  $\text{TMSAT}$  как раз и говорит о существовании сертификата.  $\square$

Однако задача  $\text{TMSAT}$  слишком искусственна. По сути, определение  $\text{NP}$  попросту зашито в определение этого языка. Гораздо интереснее содержательные  $\text{NP}$ -полные задачи, возникающие в различных областях логики, дискретной математики, алгебры и других дисциплин.

### 3.3 Выполнимость булевых формул

Типичным способом доказательства  $\text{NP}$ -трудности данной задачи является сведение к ней какой-нибудь уже известной  $\text{NP}$ -трудной задачи. Действительно, если  $B$  является  $\text{NP}$ -полным,  $B \leq_p C$  и  $C \in \text{NP}$ , то  $C$  также  $\text{NP}$ -полон: любой другой язык сводится в нему по транзитивности. Во многих случаях в качестве известной задачи выступает задача  $3\text{SAT}$ . В этом разделе мы докажем её  $\text{NP}$ -полноту.

**Определение 3.7.** Пропозициональная формула называется *выполнимой*, если она истинна хотя бы на каком-то наборе значений переменных. Языком  $\text{SAT}$  называется множество выполнимых пропозициональных формул.

**Определение 3.8.** Пропозициональная формула записана в *3-КНФ*, если она представлена в конъюнктивной нормальной форме, в которой любой дизъюнкт включает ровно три литерала. Языком  $3\text{SAT}$  называется множество выполнимых 3-КНФ.

#### 3.3.1 Сводимость к 3-КНФ

Вначале покажем, как от произвольной формулы перейти к 3-КНФ.

**Утверждение 3.9.**  $\text{SAT} \leq_p 3\text{SAT}$ .

*Доказательство.* Пусть  $\varphi$  — пропозициональная формула. Введём переменную для каждой её подформулы (включая исходные переменные и всю формулу). Каждая подформула, составленная из двух, задаёт утверждение вида  $q = r \wedge s$ .

<sup>1</sup>У универсальной машины некоторое фиксированное число лент, а у  $M$  — произвольное

Его можно представить как 3-КНФ (в данном случае  $(q \vee \neg r \vee \neg s) \wedge (\neg q \vee r \vee \neg s) \wedge (\neg q \vee \neg r \vee s) \wedge (\neg q \vee r \vee s)$ ). Взяв конъюнкцию всех таких формул, мы получим 3-КНФ, выполнимость которой эквивалентна выполнимости исходной формулы. Действительно, выполняющий набор исходной формулы задаст значения всех подформул, которые будут выполняющим набором полученной формулы. И наоборот, из выполняющего набора новой формулы можно выделить выполняющий набор исходной.

Осталось пояснить, почему этот алгоритм работает полиномиальное время. Ясно, что достаточно построить за полиномиальное время дерево синтаксического разбора формулы, дальнейшие операции занимают константное время для каждой его вершины. Построение дерева также несложно: путём подсчёта скобочного итога можно выделить из формулы две подформулы, из которых она получена, и повторить процедуру с ними рекурсивно. Это займёт линейное время для каждой подформулы, т.е. всего квадратичное время.  $\square$

### 3.3.2 Сводимость раскраски к выполнимости

Мы докажем, что любую задачу можно свести к задаче о выполнимости. Для примера покажем, как это делается для 3COL.

**Теорема 3.10.**  $3\text{COL} \leq_p \text{SAT}$ .

*Доказательство.* Заведём для каждой вершины  $v$  две переменные:  $p_{v1}$  и  $p_{v2}$ . Наложим на них условие  $p_{v1} \vee p_{v2}$ . Это значит, что для каждой вершины есть три варианта логических значений этой пары: 01, 10 и 11. Осталось добавить условие, что для соседних вершин эти варианты разные: если есть ребро  $(v, w)$ , то добавляем условие  $(p_{v1} \neq p_{w1}) \vee (p_{v2} \neq p_{w2})$ . Итоговая формула будет конъюнкцией всех условий. Очевидно, любая правильная раскраска перекодируется в выполняющий набор, и наоборот.  $\square$

Многие NP-задачи сводятся к задаче о выполнимости похожими несложными конструкциями. Следующая важнейшая теорема показывает, что эту сводимость можно провести универсальным образом.

### 3.3.3 Теорема Кука–Левина

**Теорема 3.11** (Теорема Кука–Левина). *Задача SAT является NP-полной.*

*Доказательство.* Пусть  $L \in \text{NP}$ . Рассмотрим верификатор  $V$ , работающий с сертификатами длины  $q(n)$ . Будем считать, что и вход, и сертификат заданы в двоичном алфавите. Будем также считать, что  $V$  задан машиной Тьюринга с одной лентой, бесконечной вправо, и работающей не дольше  $p(n)$  шагов. Идея состоит в записи булевой формулы, которая моделирует работу этой машины.

Напомним, что конфигурацией называется слово вида  $aqb$ , где  $q \in Q$ , а  $a, b \in \Gamma^*$ . Имеется в виду, что машина находится в состоянии  $q$  и указывает на первый символ слова  $b$ , а левее записано слово  $a$ . Все остальные ячейки заполнены бланками. Вначале машина находится в конфигурации  $q_0x\#s$ , а в конце должна быть в состоянии  $q_a$ , если сертификат  $s$  верный. Поскольку за 1 шаг машина сдвигается не больше, чем на 1 ячейку вправо, максимальная длина любой конфигурации не больше  $p(n)$ . Введём служебные символы  $\blacktriangleright$  и  $\blacktriangleleft$ , ограничивающие рабочую часть ленты. В новых обозначениях начальная конфигурация запишется как  $\blacktriangleright q_0x\#s\#^{p(n)-n-q(n)-2}\blacktriangleleft$ . Все возможные символы (элементы  $\Gamma$ , элементы  $Q$ ,  $\blacktriangleright$  и  $\blacktriangleleft$ ), закодируем в двоичной записи, пусть на это

потребуется  $k$  битов. В таком случае для кодирования всех конфигураций нам потребуется  $k(p(n) + 2)(p(n) + 1)$  битов:  $k$  битов на каждый символ,  $(p(n) + 2)$  символов в каждой конфигурации и  $(p(n) + 1)$  конфигураций за  $p(n)$  шагов, включая начальную и конечную.

Теперь нужно построить формулу, гарантирующую корректность протокола. Она будет конъюнкцией трёх формул, гарантирующих корректность начала, окончания и каждого шага. Для корректности начала нужно гарантировать, что первый символ равен  $\blacktriangleright$ , следующий —  $q_0$ , следующие  $n$  символов совпадают с битами  $x$ , далее идёт бланк, далее  $q(n)$  битов (т.е. не служебных символов), далее бланки в количестве  $p(n) - n - q(n) - 2$  и, наконец,  $\blacktriangleleft$ . Каждое такое равенство выражается простой формулой, содержащей  $k$  эквиваленций (в случае битов сертификата — дизъюнкций двух наборов из  $k$  эквиваленций), таким образом, общая длина конъюнкции всех формул будет порядка  $p(n)$ .

Для корректности окончания нужно построить формулу, проверяющую, что среди символов последней строки есть  $q_a$ . Это делается при помощи дизъюнкции эквиваленций.

Наконец, нужно построить формулу, проверяющую корректность каждого шага. Ключевой идеей здесь является то, что вычисления на машине Тьюринга локальны, т.е. на каждом шаге изменяется небольшая часть ленты, и это изменение зависит от небольшой части ленты на предыдущем шаге. Более точно, значение символа в каждой ячейке зависит от значений символов в 4 ячейках на предыдущем шаге:



(зависимость от самой правой ячейки появляется, когда машина сдвигается налево: например, 

$c$	$d$	$q$	$a$
$c$	$r$	$d$	$b$

 для команды  $qa \rightarrow rbL$ ). Вид этой зависимости полностью определяется программой машиной Тьюринга, а значит, может быть описан некоторой фиксированной пропозициональной формулой. Эту формулу нужно скопировать  $p(n)(p(n) - 1)$  раз для каждого из возможных положений фигуры 


 и взять конъюнкцию всех формул.

Таким образом, мы построили формулу, размер которой есть полином от исходных данных. Её выполнимость равносильна тому, что для некоторого сертификата  $s$  машина на входе  $x$  остановится в принимающем состоянии, т.е.  $x \in L$ . Таким образом,  $L$  сведён к SAT и теорема доказана.  $\square$

## 3.4 Примеры NP-полных задач

В этом разделе мы изучим несколько стандартных примеров NP-полных задач. Главное, что нужно помнить: для доказательства NP-полноты некоторой задачи нужно сводить не её, а к ней. Как правило, мы будем сводить к нашим задачам язык 3SAT. Общая идея заключается в построении специальных «гаджетов» для переменных и для дизъюнктов.

### 3.4.1 Клика, независимое множество, вершинное покрытие

**Определение 3.12.** Пусть дан неориентированный граф  $G$ . *Кликой* называется любой его полный подграф. *Независимым множеством* называется любое множество вершин, между которыми нет рёбер. *Вершинным покрытием* называется множество вершин, в котором лежит хотя бы один конец любого ребра.

С каждым из указанных понятий связана задача оптимизации: найти самую большую клику, самое большое независимое множество и самое маленькое вершинное покрытие. Соответствующие задачи распознавания формулируются следующим образом:

- $\text{CLIQUE} = \{(G, k) \mid \text{в графе } G \text{ есть клика из } k \text{ вершин}\};$
- $\text{INDSET} = \{(G, k) \mid \text{в графе } G \text{ есть независимое множество из } k \text{ вершин}\};$
- $\text{VERTEXCOVER} = \{(G, k) \mid \text{в графе } G \text{ есть вершинное покрытие из } k \text{ вершин}\}.$

Вначале покажем, что все эти задачи сводятся друг к другу, потом — NP-полноту одной из них.

**Утверждение 3.13.** *Задачи CLIQUE, INDSET и VERTEXCOVER полиномиально сводятся друг к другу.*

*Доказательство.* Задачи CLIQUE и INDSET полиномиально сводятся друг к другу переходом к дополнению графа, т.е. графу  $\overline{G}$ , в котором те и только те рёбра, которых нет в  $G$ .

Задачи INDSET и VERTEXCOVER сводятся друг к другу заменой  $k$  на  $n - k$ , где  $n$  — число вершин. Дело в том, что дополнение к любому вершинному покрытию есть независимое множество: если вершинное покрытие задело все рёбра, то между оставшимися вершинами рёбер быть не может, иначе эти рёбра не были бы заделаны. И наоборот, если между оставшимися рёбер нет, то все рёбра покрыты взятыми.  $\square$

**Теорема 3.14.** *Задача INDSET является NP-полной.*

*Доказательство.* Докажем, что  $3\text{SAT} \leq_p \text{INDSET}$ . Построим граф следующим образом: каждому вхождению литерала сопоставим вершину графа. Таким образом, всего будет  $3k$  вершин, где  $k$  — число дизъюнктов в 3-КНФ. Вершины, соответствующие литералам из одного дизъюнкта, соединим рёбрами. Также соединим все противоположные литералы, например  $p$  и  $\neg p$ . На этом построение графа заканчивается, размер независимого множества возьмём равным  $k$ .

Если у формулы есть выполняющий набор, то в каждом дизъюнкте есть хотя бы один истинный литерал. Соответствующие вершины образуют независимое множество, поскольку эти литералы из разных дизъюнктов, а литералы вида  $p$  и  $\neg p$  не могут быть истинны одновременно. Этим вершин как раз  $k$ .

Если, наоборот, есть независимое множество из  $k$  вершин, то эти вершины обязаны соответствовать различным дизъюнктам. А поскольку среди этих вершин нет одновременно литералов вида  $p$  и  $\neg p$ , то можно взять такие значения переменных, при которых все задействованные литералы истинны. Эти значения и будут выполняющим набором.  $\square$

### 3.4.2 Раскраски

*Лес, точно терем расписной,  
Лиловый, золотой, багряный...*

И.А.Бунин, *Листопад* (1900)



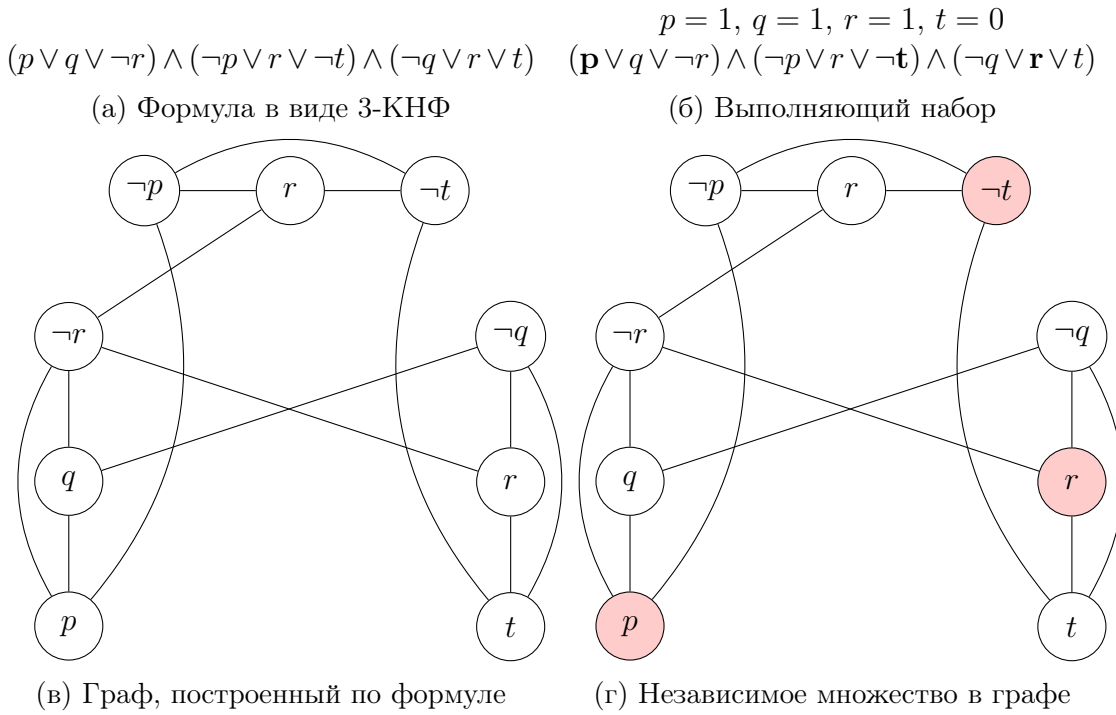
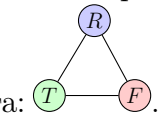


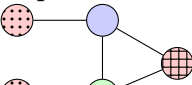
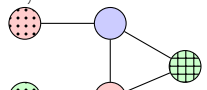
Рис. 3.1: Сводимость 3SAT к INDSET.

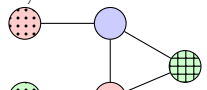
Важнейшей характеристикой графа является его хроматическое число, т.е. минимальное число цветов, в которые можно покрасить вершины графа, так чтобы вершины одного цвета не были соединены ребром. Соответствующей задачей распознавания будет задача  $\text{COL} = \{(G, k) \mid \text{вершины графа } G \text{ можно правильно раскрасить в } k \text{ цветов}\}$ . Если рассмотреть частные случаи для  $k = 1$  и  $k = 2$ , то задача будет, очевидно, полиномиальной: в первом случае достаточно проверить пустоту графа, во втором — двудольность. А вот задача 3COL о раскраске в 3 цвета уже будет NP-полной. Как следствие, NP-полной будет и общая задача COL.

**Теорема 3.15.** *Задача 3COL является NP-полной.*

*Доказательство.* 3 цвета, в которые нужно будет красить граф, назовём истинным (обозначать будем зелёным), ложным (обозначать будем красным) и синим. Частью графа будет «палитра» — треугольник, вершины которого мы по-

красим в указанные цвета:  $T$  (зелёная),  $F$  (красная),  $R$  (синяя). Далее рассмотрим такой гаджет: . Непосредственным перебором проверяется такое утверждение: если выделенные точками вершины покрашены в  $F$ , то клетчатая вершина обязательно по-

крашена в  $F$ , например . Если же точечные вершины покрашены в цвета  $TF$ ,  $FT$  или  $TT$ , то клетчатая вершина может быть покрашена в  $T$ , например .

Если же точечные вершины покрашены в цвета  $TF$ ,  $FT$  или  $TT$ , то клетчатая вершина может быть покрашена в  $T$ , например .

Состыковав два таких гаджета между собой, можно

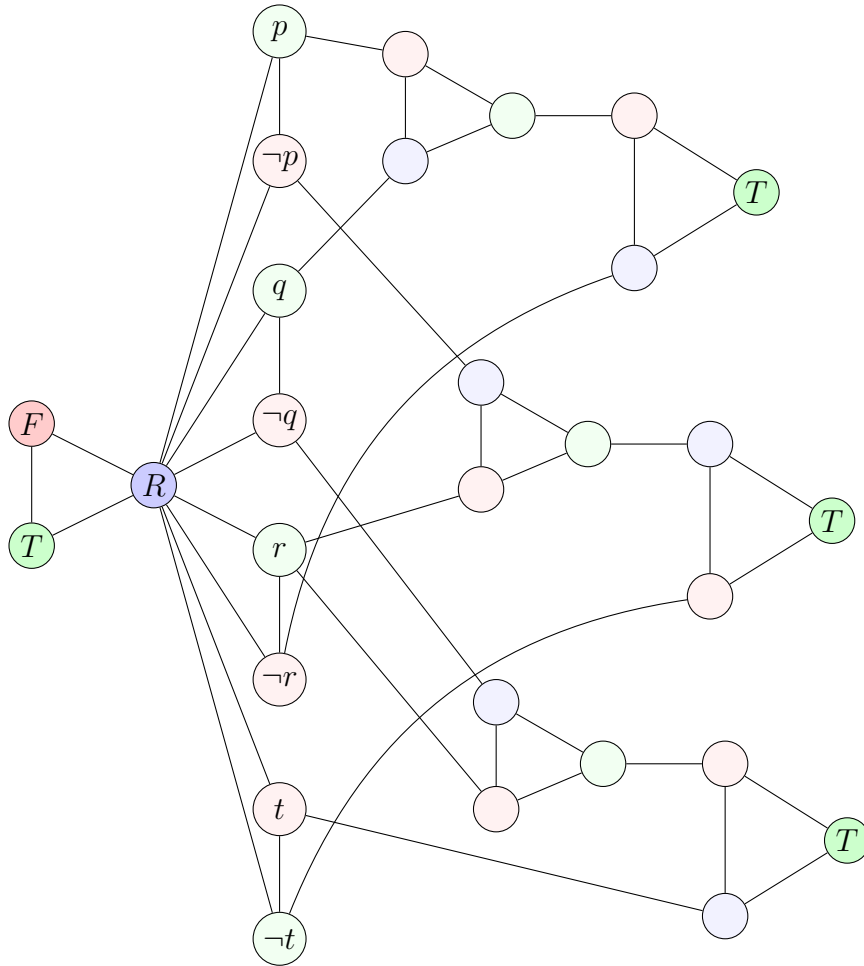
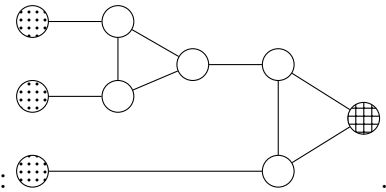
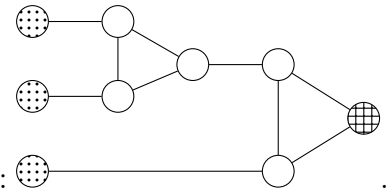


Рис. 3.2: Сводимость 3SAT к 3COL на примере формулы  $(p \vee q \vee \neg r) \wedge (\neg p \vee r \vee \neg t) \wedge (\neg q \vee r \vee t)$ . Более яркими цветами показаны жёстко зафиксированные цвета, более слабыми — раскраска, соответствующая выполняющему набору  $p = q = r = 1, t = 0$ . Вершина  $T$  нарисована несколько раз для удобства восприятия.



получить аналогичный гаджет для трёх переменных: . Теперь всё готово для построения всего графа. Во-первых, мы включим в него палитру. Во-вторых, для каждой переменной сделаем две вершины, соединённые между собой и с вершиной  $R$ . Это будет гарантировать, что они покрашены в цвета  $T$  и  $F$ , причём разные. Один вариант будет соответствовать истинности переменной, другой — ложности. Поэтому и назовём эти вершины  $p$  и  $\neg p$ . Наконец, для каждого дизъюнкта гаджет тройной дизъюнкции, подключим его точечные вершины к соответствующим литералам, а клетчатую — к вершине  $T$ . На рис. 3.2 показан граф для формулы  $(p \vee q \vee \neg r) \wedge (\neg p \vee r \vee \neg t) \wedge (\neg q \vee r \vee t)$ .

Докажем, что выполнимость исходной формулы эквивалентна 3-раскрашиваемости полученного графа. Пусть формула выполнима. Покрасим в зелёный истинные вершины-литералы и в красный ложные. В силу выполнимости к каждому гаджету будет подключена хотя бы одна зелёная вершина, и он успешно раскрасится. Обратно, пусть граф раскрашен. Вершины-литералы соединены с синей вершиной палитры, поэтому из каждой пары одна красная, а другая зелёная.

Поскольку каждый гаджет правильно раскрашен, он не может быть подключён к трём красным вершинам. Значит, если сделать истинными литералы, соответствующие которым вершины покрашены в зелёный, то в каждом дизъюнкте будет хотя бы один истинный, т.е. формула выполнима.  $\square$

### 3.4.3 Пути и циклы в графе

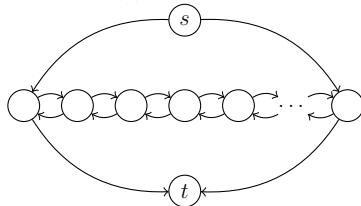
*Show me the way to the next whisky bar...*

Bertolt Brecht // Elisabeth Hauptmann // The Doors,  
*Alabama Song*

Напомним, что эйлеровым путём называется путь, проходящий по одному разу по каждому ребру, а гамильтоновым — проходящий по одному разу через каждую вершину. Для эйлеровости существует простой критерий: граф должен быть связан, а степени всех вершин, кроме, быть может, двух, чётны. Обе характеристики можно проверить за полиномиальное время. А вот для гамильтоновости простого критерия нет. Более того, задача о гамильтоновом пути является NP-полной. Сначала рассмотрим ориентированный случай, затем перейдём к неориентированному.

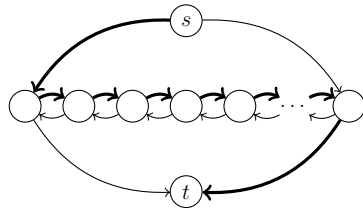
**Теорема 3.16.** Язык  $\text{DHAMPATH} = \{(G, s, t) \mid \text{в ориентированном графе } G \text{ есть ориентированный путь из } s \text{ в } t, \text{ проходящий ровно один раз через каждую вершину}\}$  является NP-полным.

*Доказательство.* Сведём 3SAT к DHAMPATH. Ключевым инструментом будет

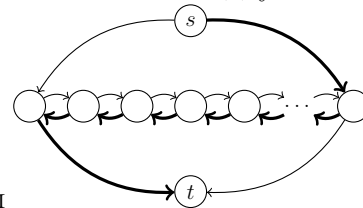


такой гаджет:

. Ясно, что его можно обойти двумя спо-



собами: зигзагом



и загзигом

Мы возьмём по одному такому гаджету для каждой переменной из формулы и соединим их в одну большую цепочку. Число промежуточных вершин в каждом гаджете возьмём равным  $3m + 1$ , где  $m$  — число дизъюнктов в 3-КНФ. Из этих вершина 1-ая, 4-ая, 7-ая, ...,  $(3m + 1)$ -ая точно больше ни с чем не соединены, а остальные могут быть соединены с вершинами для дизъюнктов, которые мы добавим по одной для каждого. Если переменная входит в дизъюнкт  $C_i$  без отрицания, то добавим рёбра из  $(3i - 1)$ -ой вершины в вершину для конъюкта, а оттуда в  $3i$ -ую. Если же переменная входит с отрицанием, то наоборот: из  $3i$ -ой вершины в вершину для конъюкта, а оттуда в  $(3i - 1)$ -ую.

Покажем, что формула выполнима тогда и только тогда, когда в построенном графе есть гамильтонов путь из начальной вершины первого гаджета цепочки в конечную вершину последнего. По выполняющему набору путь строится очень просто: гаджеты, соответствующие истинным переменным, путь будет проходить зигзагом, а гаджеты, соответствующие ложным, — загзигом. Когда путь проходит гаджет переменной, которая делает истинным некоторый

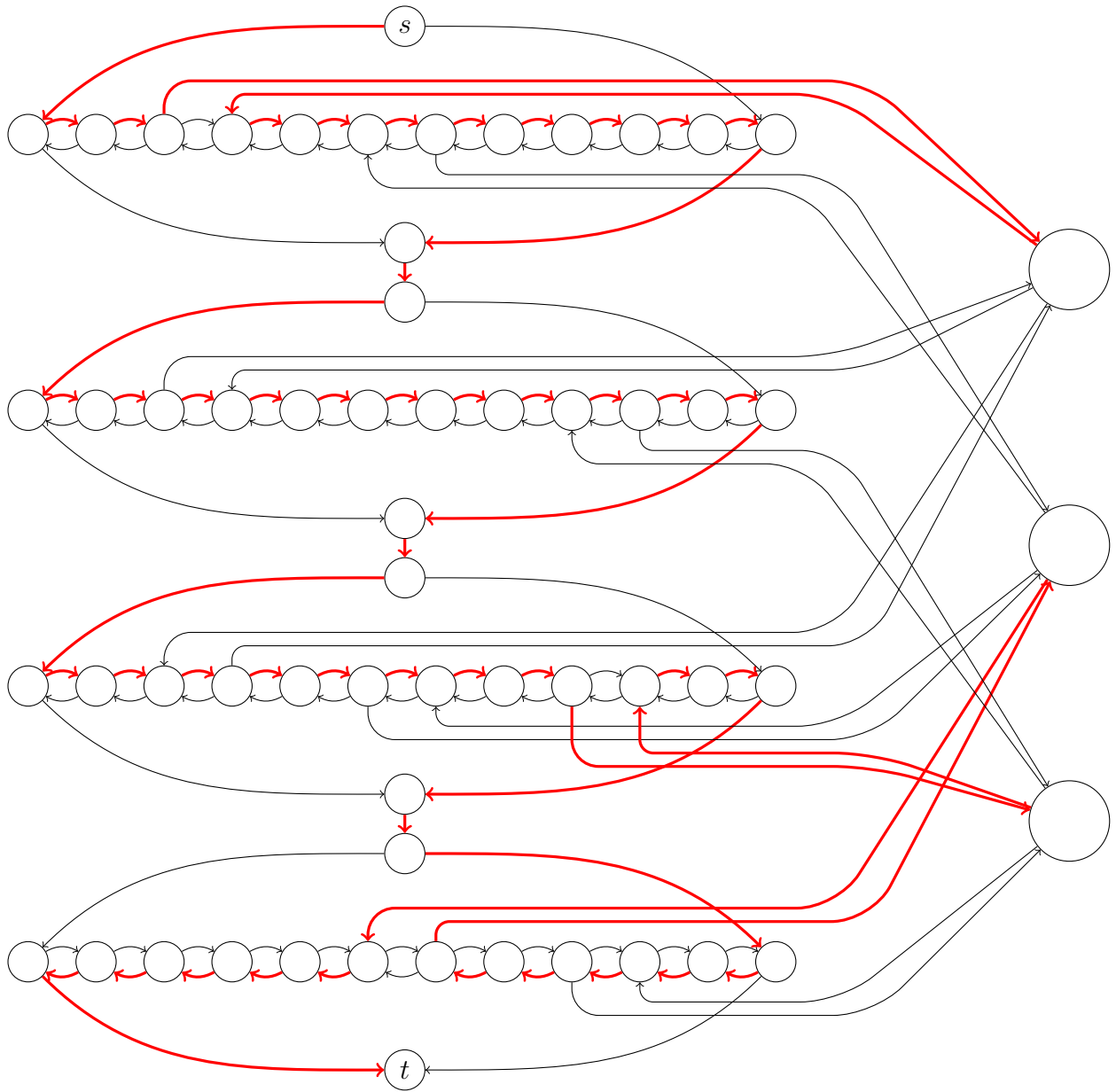


Рис. 3.3: Сводимость 3SAT к DHAMPATH на примере формулы  $(p \vee q \vee \neg r) \wedge (\neg p \vee r \vee \neg t) \wedge (\neg q \vee r \vee t)$  (конечные и начальные вершины гаджетов можно склеить, но мы их рисуем отдельно для лучшей читаемости). Гамильтонов путь выделен жирными дугами и красным цветом.

дизъюнкт, он отклоняется по дороге в вершину, соответствующую этому литералу, но тут же возвращается обратно. Поскольку все дизъюнкты истинны, все вершины будут посещены. Построение графа и гамильтонова пути в нём проиллюстрировано на рис. 3.3.

Осталось доказать, что любой гамильтонов путь будет иметь именно такой вид. Посетив вершину, соответствующую некоторому дизъюнкту, путь мог бы вернуться в какой-то другой гаджет. Однако в таком случае в оставленном гаджете осталась бы вершина, которую заведомо нельзя посетить: именно для этого мы оставляли каждую третью вершину ни с чем более не связанной. Значит, путь уже не может быть гамильтоновым, что и требовалось. Таким образом, по любому гамильтонову пути можно восстановить выполняющий набор.  $\square$

Теперь перейдём к случаю неориентированного графа.

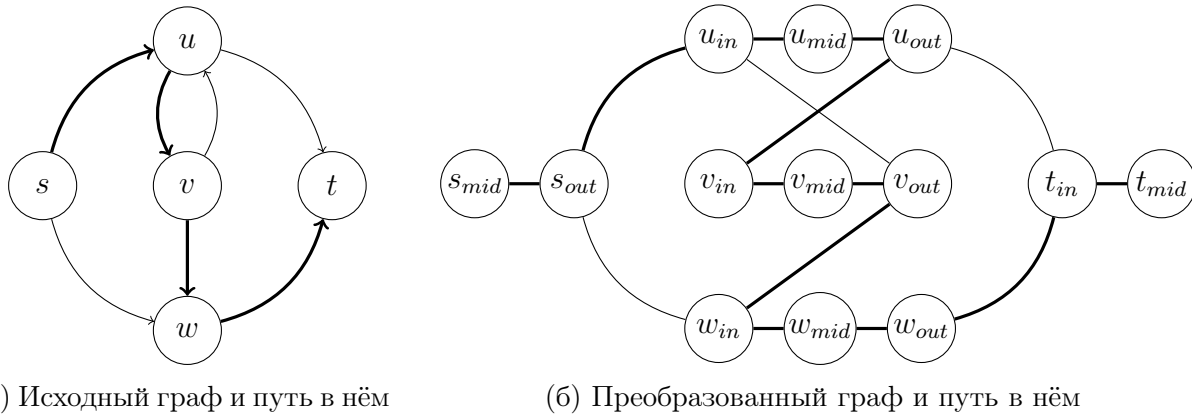


Рис. 3.4: Пример построения неориентированного графа по ориентированному с сохранением гамильтоновости

**Теорема 3.17.** Язык  $\text{UNAMPATH} = \{(G, s, t) \mid \text{в неориентированном графе } G \text{ есть путь из } s \text{ в } t, \text{ проходящий ровно один раз через каждую вершину}\}$  является **NP**-полным.

*Доказательство.* Сведём  $\text{DNAMPATH}$  к  $\text{UNAMPATH}$ . Проведём такое преобразование: каждую вершину  $v$ , кроме  $s$  и  $t$ , превратим в 3 вершины  $v_{in}$ ,  $v_{mid}$  и  $v_{out}$ , соединённые в цепочку:  $v_{in} - v_{mid} - v_{out}$ . Аналогично  $s$  и  $t$  превратим в пары соединённых вершин  $s_{mid} - s_{out}$  и  $t_{in} - t_{mid}$ . Для каждого ребра  $(u, v)$  исходного графа проведём ребро  $(u_{out}, v_{in})$  в новом. Стартовой и конечной вершинами будем считать  $s_{mid}$  и  $t_{mid}$ .

По гамильтонову пути в исходном графе легко получить гамильтонов путь в преобразованном. Достаточно вместо каждого ребра  $(u, v)$  из исходного пути взять цепочку вершин  $(u_{mid}, u_{out}, v_{in}, v_{mid})$ . Или, иначе говоря, первую вершину пути заменить на пару вершин  $(s_{mid}, s_{out})$ , каждую промежуточную вершину  $w$  заменить на три последовательные вершины  $(w_{in}, w_{mid}, w_{out})$ , а последнюю опять на пару  $(t_{in}, t_{mid})$ . Пример показан на рис. 3.4. Теперь покажем, что и по гамильтонову пути в преобразованном графе можно восстановить гамильтонов путь в исходном. Для этого достаточно заметить, что вершины обязательно идут тройками вида  $(w_{in}, w_{mid}, w_{out})$ . Это доказывается по индукции. База: из  $s_{mid}$  пусть идёт в  $s_{out}$ , поскольку других вариантов нет. Переход: пусть до сих пор все вершины шли тройками. Из очередной out-вершины можно попасть только в in-вершины, причём из пока не задействованных троек. Если дальше путь пойдёт не в соответствующую mid-вершину, то эта вершина никогда не сможет быть посещена без повторного посещения текущей in-вершины, а значит путь не будет гамильтоновым. Значит, в гамильтоновом пути после in-вершины будет mid-вершина, а затем и out-вершина, поскольку других вариантов нет. Значит, снова получилась целая тройка, что и требовалось.  $\square$

Небольшой модификацией можно получить **NP**-полноту и задач о гамильтоновом *цикле* в ориентированном и неориентированном варианте. Ещё одна известная задача о путях в графе — *задача коммивояжёра* (traveling salesman problem). В этом случае дан не просто граф, а граф с весами, т.е. неотрицательными числами на каждом ребре. *Путём коммивояжёра* называется путь минимального суммарного веса, проходящий хотя бы один раз через каждую вершину. Вершины интерпретируются как города, рёбра — как дороги между

городами, веса — как длины дорог. Коммивояжёр должен посетить все города, проехав как можно меньше.

**Теорема 3.18.** Язык  $TSP = \{(G, \mathbf{w}, s, t, l) \mid \text{во взвешенном графе } (G, \mathbf{w}) \text{ есть путь коммивояжёра из } s \text{ в } t \text{ длины не больше } l\}$  является NP-полным.

*Доказательство.* Сведём UNAMPATH к TSP. Достаточно сопоставить всем рёбрам единичные веса и взять  $l = n - 1$ , где  $n$  — число вершин. (В случае  $s = t$  нужно взять  $l = n$ ). Действительно, путь из  $n - 1$  ребра, проходящий через все вершины, обязан быть гамильтоновым.  $\square$

Иногда при формулировке задачи коммивояжёра требуют, чтобы граф обязательно был полным. Ясно, что это не влияет на суть задачи: все отсутствующие рёбра можно провести, но сопоставить им очень большие веса, так чтобы оптимальный путь через них заведомо не проходил. В рассуждении об NP-полноте можно взять все рёбра, отсутствующие в исходном графе, с весом больше 1: на вывод это не повлияет. При этом веса рёбер образуют метрику, т.е. для них выполняется неравенство треугольника, соответственно, метрическая задача коммивояжёра также будет NP-полной. Можно показать, что даже евклидова задача коммивояжёра, в которой все вершины лежат в евклидовом пространстве, а расстояния получаются при помощи обычной евклидовой метрики, является NP-полной. Конструкция работает уже в двумерном пространстве, т.е. на декартовой плоскости.

### 3.4.4 Покрытие множествами и задача о рюкзаке

Рассмотрим такую задачу о покрытии множествами: EXACTSETCOVER =  $\{(n, S_1, \dots, S_m) \mid \text{можно выбрать непересекающиеся множества } S_{i_1}, \dots, S_{i_k}, \text{ в объединении дающие все числа от } 1 \text{ до } n\}$ . Иными словами, все числа от 1 до  $n$  должны быть покрыты ровно по одному разу.

**Теорема 3.19.** Задача EXACTSETCOVER является NP-полной.

*Доказательство.* Сведём 3SAT к EXACTSETCOVER. Ясно, что элементы множеств можно называть как угодно. Мы заведём для каждой переменной три возможных элемента  $p$ ,  $p_0$  и  $p_1$ , а для каждого дизъюнкта — четыре элемента  $C, C_1, C_2, C_3$ . Множества будут трёх видов. Во-первых, для каждой переменной возьмём множества  $\{p, p_0\}$  и  $\{p, p_1\}$ . Во-вторых, для каждого дизъюнкта возьмём множества  $\{C\}$ ,  $\{C, C_1\}$ ,  $\{C, C_2\}$ ,  $\{C, C_3\}$ ,  $\{C, C_1, C_2\}$ ,  $\{C, C_2, C_3\}$  и  $\{C, C_1, C_3\}$ . Наконец, в-третьих, для каждой переменной заведём множества  $\{p_0, C_j, \dots\}$  и  $\{p_1, D_j, \dots\}$ , где в первое множество включены элементы для всех дизъюнктов  $C$ , содержащих литерал  $p$  на  $j$ -ом месте, а во второе — для дизъюнктов  $D$ , содержащих литерал  $\neg p$  на  $j$ -ом месте.

Пусть есть выполняющий набор. Если переменная  $p$  в этом наборе истинна, возьмём множество  $\{p, p_1\}$ , если ложна, то  $\{p, p_0\}$ . Далее, в первом случае возьмём  $\{p_0, C_j, \dots\}$ , во-втором —  $\{p_1, D_j, \dots\}$ . Поскольку в каждом дизъюнкте есть хотя бы один истинный литерал, мы уже покрыли хотя бы один из элементов  $C_1, C_2, C_3$ . Добавляем в покрытие множество из  $C$  и непокрытых элементов  $C_i$  и получаем покрытие всех элементов.

Обратно, пусть имеется покрытие. Для каждой переменной  $p$  в него входит ровно одно из множеств  $\{p, p_0\}$  и  $\{p, p_1\}$ , а для каждого дизъюнкта  $C$  — ровно одно из множеств  $\{C\}$ ,  $\{C, C_1\}$ ,  $\{C, C_2\}$ ,  $\{C, C_3\}$ ,  $\{C, C_1, C_2\}$ ,  $\{C, C_2, C_3\}$  и  $\{C, C_1, C_3\}$ . Выбор первого множества задаст истинность  $p$  в выполняющем

наборе. Оставшийся элемент из  $p_0$  и  $p_1$  должен быть покрыт множеством вида  $\{p_i, C_j, \dots\}$ . Поскольку покрыты все элементы, для каждого дизъюнкта хотя бы один элемент  $C_j$  покрывается множеством вида  $\{p_i, C_j, \dots\}$ . По построению это означает, что соответствующий литерал в дизъюнкте истинен. Значит, все дизъюнкты истинны, т.е. набор выполняющий, что и требовалось.  $\square$

Перейдём к задаче о рюкзаке и её частному случаю  $\text{SUBSET-SUM} = \{(k_1, \dots, k_m, N) \mid \exists \alpha \in \{0, 1\}^m \sum \alpha_i k_i = N\}$ .

**Теорема 3.20.** *Задача SUBSET-SUM является NP-полной.*

*Доказательство.* Сведём EXACTSETCOVER к SUBSET-SUM. В качестве  $N$  возьмём число из  $n$  единиц в  $(m+1)$ -ичной записи (это займёт  $O(n \log m)$  битов). Число  $k_i$  также будет записано в  $(m+1)$ -ичной записи: в разрядах, номера которых лежат в  $S_i$ , будут стоять единицы, в остальных нули.

Если существует покрытие, то сумма соответствующих  $k_i$  даёт в точности  $N$ : в каждом разряде ровно один раз встретится единица.

Пусть теперь  $\sum \alpha_i k_i = N$ . В силу того, что основание системы счисления больше числа слагаемых, а в разрядах стоят только нули и единицы, переносов при сложении не будет. Значит, для каждого разряда ровно в одном слагаемом стоит единица. Значит, множества  $S_i$ , для которых  $\alpha_i = 1$ , образуют покрытие.  $\square$

### 3.4.5 Вариации с выполнимостью

Многие частные случаи задачи о выполнимости булевых формул, а также многие смежные задачи являются NP-полными. Приведём несколько примеров.

**Теорема 3.21.** *Задача 3SAT остаётся NP-полной, даже если ограничиться формулами, в которых каждая переменная входит не более 3 раз, а каждый литерал — не более двух раз.*

*Доказательство.* Идея проста: каждую формулу преобразуем к указанному виду с сохранением выполнимости. Для этого заменим каждое вхождение каждой переменной на новую переменную. Если новые переменные, соответствующие  $p$ , суть  $p_1, \dots, p_k$ , то добавим в конъюнкцию формулу  $(p_1 \vee \neg p_2) \wedge (p_2 \vee \neg p_3) \wedge \dots \wedge (p_k \vee \neg p_1)$ . Условие на число вхождений, очевидно, будет соблюдено. Покажем, что выполнимость сохранилась. Действительно, выполнимость добавленной для переменной  $p$  формулы эквивалентна тому, что все переменные  $p_i$  принимают одно и то же значение. Значит, выполняющий набор новой формулы определяет набор значений старых переменных, при этом он также будет выполняющим. Обратное тоже верно, значит сводимость установлена.  $\square$

**Теорема 3.22.** *Задача NAESAT =  $\{\varphi \mid \text{для формулы } \varphi \text{ в форме 3-КНФ есть набор значений, при котором в каждом дизъюнкте есть как истинный, так и ложный литералы}\}$  является NP-полной.*

*Доказательство.* Сведём 3SAT к NAESAT. Для каждого дизъюнкта  $(\alpha_i \vee \beta_i \vee \gamma_i)$  введём два новых дизъюнкта:  $(\alpha_i \vee \beta_i \vee z_i)$  и  $(\neg z_i \vee \gamma_i \vee u)$ , где переменная  $z_i$  своя для каждого дизъюнкта, а переменная  $u$  одина для всех.

Пусть исходная формула была выполнима. Рассмотрим набор, в котором все исходные переменные те же самые, а  $u = 0$ . Если  $\alpha_i = 1$  или  $\beta_i = 1$ , положим

$z_i = 0$ . В таком случае в обоих дизъюнктах будет по истинному и ложному литералу. Если же  $\alpha_i = \beta_i = 0$ , положим  $z_i = 1$ . Поскольку в этом случае обязательно  $\gamma_i = 1$ , вновь в обоих дизъюнктах будет по истинному и ложному литералу.

Теперь обратно, пусть в новой формуле есть набор, при котором в каждой скобке есть истинный и ложный литералы. Если в этом наборе  $u = 0$ , то хотя бы один из литералов  $\alpha_i$ ,  $\beta_i$  и  $\gamma_i$  должен быть истинным, т.е. соответствующий дизъюнкт выполним. Если же в этом наборе  $u = 1$ , то рассмотрим другой набор, в котором все переменные принимают противоположные значения. Если в исходном наборе в каждом дизъюнкте были истинный и ложный литералы, то и в новом тоже так будет. Значит, этот случай свёлся к предыдущему, и сводимость доказана.  $\square$

Можно показать, что задача о выполнимости 2-КНФ решается за полиномиальное время. Но если расширить задачу в другом направлении, а именно максимизировать число выполненных дизъюнктов в 2-КНФ, она снова станет NP-полной.

**Теорема 3.23.** *Задача  $\text{MAX2SAT} = \{(\varphi, k) \mid \varphi \text{ — формула в виде 2-КНФ, для которой существует набор значений, при котором выполнены } k \text{ дизъюнктов}\}$  является NP-полной.*

*Доказательство.* Сведём 3SAT к MAX2SAT. Рассмотрим следующие 10 дизъюнктов:  $p, q, r, s, \neg p \vee \neg q, \neg p \vee \neg r, \neg q \vee \neg r, p \vee \neg s, q \vee \neg s, r \vee \neg s$ . Заметим, что если  $p, q$  и  $r$  одновременно истинны, то  $\neg p \vee \neg q, \neg p \vee \neg r$  и  $\neg q \vee \neg r$  ложны, при  $s = 1$  истинны ровно 7 из 10 дизъюнктов. Если  $p = q = 1, r = 0$ , то ложны  $r, \neg p \vee \neg q$  и один из дизъюнктов  $s, r \vee \neg s$ , т.е. снова истинны 7 из 10. Если  $p = 1, q = r = 0$ , то ложны  $q, r$  и либо  $s$ , либо  $q \vee \neg s$  и  $r \vee \neg s$ , снова истинны не больше 7 из 10. И только если  $p = q = r = 0$ , будут истинны не больше 6 из 10 дизъюнктов. Сводимость построим так: каждый дизъюнкт  $(p \vee q \vee r)$  заменим на 10 вышеуказанных дизъюнктов, взяв новую переменную  $s$ . В качестве  $k$  возьмём число  $7m$ , где  $m$  — общее число дизъюнктов. Из вышесказанного следует, что если все дизъюнкты выполнимы, то в каждой группе можно выполнить по 7 дизъюнктов, всего  $7m$ . А если все одновременно не выполнимы, то хотя бы в одной группе будут выполнены 6 дизъюнктов, а в остальных не больше 7, в итоге  $7m$  не наберётся.  $\square$

### 3.4.6 Линейное и квадратичное программирование

Многие практические задачи можно сформулировать как задачи линейного программирования (ЛП), т.е. оптимизации некоторой линейной функции при условии выполнения некоторых линейных уравнений и неравенств. На практике используют экспоненциальный в худшем случае, но обычно быстрый симплекс-метод. При этом существуют полиномиальные в любом случае алгоритмы, например, метод эллипсоидов Хачияна. Как обычно, можно превратить задачу оптимизации в задачу распознавания, поставив вопрос, может ли целевая функция быть меньше некоторого порога.

Ситуация сильно осложняется, если требуется найти не произвольное решение, а целочисленное (в таком случае задачу называют задачей целочисленного линейного программирования — ЦЛП). Даже если рассматривать лишь системы, в которых все коэффициенты есть нули или единицы, задача уже будет



NP-трудной. Практически любую задачу можно переформулировать как задачу ЦЛП. Например, задача о вершинном покрытии формулируется так: существуют ли такие целые  $x_1, \dots, x_n$ , что  $\sum x_i \leq k$ , но  $x_i + x_j \geq 1$  для всех рёбер  $(i, j)$ . Однако в данном случае неочевиден вопрос о том, лежит ли язык в классе NP. Понятно, что если решение есть, то его можно проверить, но при этом решение должно иметь короткую запись. Известно, что любую систему линейных неравенств при помощи добавления новых переменных можно привести к виду  $Ax = b, x \geq 0$ . Именно для такой системы мы докажем NP-полноту.

**Теорема 3.24.** Задача  $\text{INTPROG} = \{(A, b) \in \mathbb{Z}^{nm} \times \mathbb{Z}^n \mid \exists x \in \mathbb{N}^m Ax = b\}$  является NP-полной.

*Доказательство.* NP-трудность мы фактически уже доказали. Для полноты изложения приведём сводимость к задаче ровно такого вида. В этот раз будем сводить задачу EXACTSETCOVER. Пусть дана система множеств  $(S_1, \dots, S_m)$ . В столбцах матрицы  $A$  запишем характеристические функции этих множеств, а вектор  $b$  составим из единиц. Если покрытие есть, то возьмём вектор  $x$ , такой что  $x_i = 1$ , если  $S_i$  входит в покрытие, и  $x_i = 0$ , если  $S_i$  не входит. Поскольку каждый элемент входит ровно в одно множество покрытия, в сумме получится как раз вектор из всех единиц. Обратно, пусть есть решение задачи ЦЛП. Если в нём есть хотя бы одно число больше 1, то и в итоговой сумме будет число больше 1, т.е. вектор из всех единиц получиться не может. Значит, все числа будут либо 0, либо 1, что и задаст покрытие.

Осталось доказать принадлежность к NP. Для этого мы докажем, что если задача ЦЛП имеет какое-то решение, то она имеет и решение, все элементы которого не превосходят  $2^{\text{poly}(n, m, \log a)}$ , где  $a$  — максимальный элемент  $A$  и  $b$ . В качестве точного значения границы возьмём  $N = n(ma)^{2m+1}$ . Введём также обозначение  $M = (am)^m$ . Предположим, что во всех натуральных решениях системы  $Ax = b$  хотя бы одна координата будет больше  $N$ . Из всех решений выберем одно, у которого сумма координат, превышающих  $M$  (а вовсе не  $N$ ), будет наименьшей. Без ограничения общности для этого  $x$  выполнено  $x_1 > M, \dots, x_k > M, x_{k+1} \leq M, \dots, x_m \leq M$ . Обозначим через  $v_1, \dots, v_k$  соответствующие столбцы матрицы  $A$ . Рассмотрим два случая:

1. Существует нетривиальная целочисленная линейная комбинация с коэффициентами от 0 до  $M$ , равная нулю:  $\sum \lambda_i v_i = 0$ . В этом случае вектор  $(x_1 - \lambda_1, \dots, x_k - \lambda_k, x_{k+1}, \dots, x_m)$  также неотрицателен и удовлетворяет уравнению, что противоречит минимальности исходного  $x$ .
2. Такой линейной комбинации не существует. Для начала заметим, что в таком случае вообще никакой нетривиальной неотрицательной линейной комбинации не существует. Действительно, коэффициенты любой такой линейной комбинации находятся как решение некоторой системы линейных уравнений размерности не больше  $k - 1$  с коэффициентами не больше  $a$ . По правилу Крамера числа из этого решения будут дробями с числителем и знаменателем — определителями размера не больше  $(k - 1) \times (k - 1)$ . Каждый из этих определителей не будет превышать  $(k - 1)! \cdot a^{k-1} < m! \cdot a^m < M$ . При этом все знаменатели одинаковы, поэтому была бы и целочисленная комбинация с коэффициентами меньше  $M$ . А раз такой нет, то нет никакой.

Далее, рассмотрим множество  $S$  всех неотрицательных линейных комбинаций векторов  $v_1, \dots, v_k$ . Оно, очевидно, выпукло (и является конусом).

При этом точка  $\mathbf{0}$  находится на его границе, иначе была бы нетривиальная нулевая комбинация. По теореме об опорной гиперплоскости существует  $\mathbf{h} \in \mathbb{R}^n$ , такое что  $\langle \mathbf{h}, \mathbf{v}_i \rangle \geq 0$  при всех  $i$ . Более того, все неравенства можно сделать строгими, иначе в множестве  $S$  были бы два противоположных вектора, и нетривиальная нулевая комбинация существовала бы.<sup>2</sup> И более того, можно выбрать  $\mathbf{h}$  с коэффициентами по модулю не больше  $M$ , так чтобы  $\langle \mathbf{h}, \mathbf{v}_i \rangle \geq 1$ .<sup>3</sup>

Теперь заметим, что  $\sum_{j=1}^k x_j \leq \sum_{j=1}^k \langle \mathbf{h}, \mathbf{v}_j \rangle x_j = \langle \mathbf{h}, \sum_{j=1}^k \mathbf{v}_j x_j \rangle = \langle \mathbf{h}, A\mathbf{x} - \sum_{j=k+1}^m \mathbf{v}_j x_j \rangle = \langle \mathbf{h}, \mathbf{b} - \sum_{j=k+1}^m \mathbf{v}_j x_j \rangle$ . Последнее выражение есть скалярное произведение двух  $n$ -мерных целочисленных векторов, координаты первого из которых не больше  $nM$ , а второго — не больше  $a + a(m-1)M < (am)^{m+1}$ . Значит, всё произведение меньше  $n^2(am)^{2m+1} = N$ . Однако в левой части стоит сумма чисел каждое из которых больше  $N$ . Получаем противоречие, значит этот случай также невозможен.

Таким образом, у какого-то решения все компоненты не больше  $N$ , что и требовалось.  $\square$

От линейного программирования перейдём к квадратичному. Под задачей квадратичного программирования будем понимать систему уравнений второго порядка от  $n$  переменных. Вначале рассмотрим задачу с булевыми коэффициентами.

**Теорема 3.25.** *Задача  $\text{QUADEQ} = \{(A, \mathbf{b}) \in \{0, 1\}^{n^2} \times \{0, 1\}^n \mid \exists \mathbf{x} \in \{0, 1\}^n \mathbf{x}^T A \mathbf{x} = \mathbf{b}\}$  является NP-полной.*

*Доказательство.* Мы сведём SAT к QUADEQ. Доказательство в целом повторяет доказательство утверждения 3.9. Отличие в том, что утверждения вида  $p = q \vee s$  записываются не в 3-КНФ, а в виде многочленов. Поскольку с каждой стороны не больше двух переменных, получатся квадратичные многочлены.  $\square$

Наконец, рассмотрим задачу с произвольными коэффициентами:

**Теорема 3.26.** *Задача  $\text{REALQUADEQ} = \{(A, \mathbf{b}) \in \mathbb{Z}^{n^2} \times \mathbb{Z}^n \mid \exists \mathbf{x} \in \mathbb{R}^n \mathbf{x}^T A \mathbf{x} = \mathbf{b}\}$  является NP-полной.*

*Доказательство.* Слегка изменим предыдущую конструкцию, чтобы получилась сводимость к REALQUADEQ. Для этого потребуется две вещи: во-первых, для каждой переменной добавим уравнение  $x^2 = x$ , что ограничит её значение нулями и единицами. Во-вторых, многочлены для записи булевых функций запишем в действительнзначной арифметике, а не в булевой:  $\neg x = 1 - x$ ,  $x \wedge y = xy$ ,  $x \vee y = x + y - xy$ .  $\square$

<sup>2</sup>Подробнее: множество  $H$  таких векторов  $\mathbf{h}$ , таких что  $\langle \mathbf{h}, \mathbf{v}_i \rangle \geq 0$ , также является конусом, причём двойственным к  $S$ . Если все неравенства нельзя сделать строгими, то это конус неполной размерности, т.е. вложен в некоторую гиперплоскость. Если размерность конуса равна  $n - 1$ , то два вектора, нормальные к этой гиперплоскости и противоположные по направлению, должны найтись среди векторов  $\mathbf{v}_i$ . В общем случае, если размерность равна  $n - p$ , среди  $\mathbf{v}_i$  должны найтись  $p$  пар противоположных по направлению векторов, образующие ортогональное подпространство.

<sup>3</sup>А это делается так: вначале нужно найти целочисленные образующие конуса  $H$ . Это делается решением линейных систем из  $n$  уравнений с  $n$  неизвестными. Как мы уже знаем, все компоненты этих векторов будут меньше  $(an)^n$ . При этом можно считать, что  $n \leq k < m$ , иначе задача будет тривиальной. Из векторов можно выбрать  $k$ , в каждом из которых выполнено своё строгое неравенство  $\langle \mathbf{h}, \mathbf{v}_i \rangle > 0$ . Теперь нужно рассмотреть сумму этих векторов. У неё все компоненты будут меньше  $k(an)^n < M$ .

## 3.5 Математические доказательства

*До сих пор не доказано, что для нахождения  
математических доказательств нужно больше времени,  
чем для их проверки*

Леонид Левин, 1973

Часто бывает, что великие идеи опережают своё время. Так, древние греки создали прототип паровой машины, но использовали его как игрушку, Лейбниц говорил о сведении любого рассуждения к вычислению ещё в начале XVIII века, а его подход к бесконечно малым величинам был формализован только в теории нестандартного анализа, Чарльз Бэббидж и Ада Лавлейс писали и исследовали программы для «аналитической машины» в середине XIX века, предвосхитив современное программирование. То же случилось и со сложностью задачи о выполнимости. В 1956 году в письме [58] Джону фон Нейману Курт Гёдель ставит вопрос, который в современных терминах звучит так: существует ли линейный или квадратичный алгоритм для проверки выполнимости формулы? Гёдель связал этот вопрос со сложностью математических доказательств: при помощи такого алгоритма можно было бы относительно быстро проверить, существует ли короткое доказательство некоторой теоремы формальной арифметики. А от слишком длинного доказательства нет никакого толку: всё равно его никто не сможет прочесть и понять.

Самое длинное из доказательств, к текущему моменту придуманных и написанных людьми, это доказательство теоремы о классификации конечных простых групп. Оно раскидано по сотням статей десятков авторов, выходявших на протяжении 50 лет. Общий его объём составляет больше 10 тысяч страниц, то есть несколько десятков мегабайт. При этом никакого единого текста доказательства нет, а проект по соответствующему многотомному изданию застыл из-за постепенного отхода от дел основных авторов. Такая ситуация ставит ещё и вопрос, насколько можно эту теорему считать доказанной: не осталось ли незамеченной ранее ошибки. В случае компьютерных доказательств граница сдвигается на несколько порядков. Самое длинное доказательство из полученных на данный момент — контрпример к гипотезе о пифагоровых тройках [73].<sup>4</sup> В доказательстве показано, что натуральные числа от 1 до 7825 нельзя раскрасить в два цвета, так чтобы не было ни одной одноцветной пифагоровой тройки. Доказательство строится на упрощении полного перебора и всё равно занимает около 200 терабайт. Правда, его можно заархивировать до «всего лишь» 68 гигабайт. Предыдущий рекорд [86] относился к доказательству гипотезы Эрдёша о расходимости и занимал 14 гигабайт и архивировался до 750 мегабайт.<sup>5</sup>

В любом случае слишком длинные доказательства невозможно проверить, вопрос только в конкретном размере границы. Поэтому с практической точки зрения нет разницы между отсутствием доказательства и ситуацией, когда кратчайшее доказательство чересчур длинное. Если бы можно было быстро проверять наличие коротких доказательств, то с практической точки зрения все открытые математические проблемы были бы решены: для каждой гипотезы можно было бы найти либо доказательство, либо опровержение, либо узнать, что ни короткого доказательства, ни короткого опровержения нет. Впрочем, ситуация зависит ещё и от используемой формальной системы. Благодаря тому

<sup>4</sup>См. также популярную заметку [158]

<sup>5</sup>Впоследствии гипотезу передоказал Теренс Тао традиционными методами [133], см. также популярную заметку [164].

же Гёделю известна теорема об укорачивании доказательств: при расширении формальной системы длина кратчайшего доказательства может значительно уменьшиться. Сам Гёдель использовал трюки с диагонализацией, строя предложения вида: «Кратчайшее доказательство этого предложения имеет длину хотя бы гуголплекс». Впоследствии были построены более явные примеры, например, Харви Фридман рассмотрел такое утверждение:

Существует натуральное  $n$ , такое что для любой последовательности укоренённых деревьев  $T_1, T_2, \dots, T_n$ , такой что каждое  $T_k$  состоит не более чем из  $k + 10$  вершин, некоторое дерево может быть гомеоморфно вложено в более позднее.

Это утверждение является частным случаем теоремы Крускала и потому имеет короткое доказательство в арифметике второго порядка. Но длина кратчайшего доказательства в арифметике Пеано есть башня из экспонент высоты 1000. Подробности можно найти в статье Крэга Смюринского [128].

К сожалению, письмо осталось незамеченным: фон Нейман вскоре скончался от тяжёлой болезни, а сам Гёдель никогда не повторял публично эти идеи. К моменту публикации работ Кука и Левина он уже не был активен в науке в силу психического расстройства, а письмо было вновь обнаружено в архиве фон Неймана лишь в 1980-х годах [70].

В терминах NP-полноты мы получаем такую теорему о теоремах:

**Утверждение 3.27.** *Язык  $\text{THEOREMS} = \{(\varphi, 1^n) \mid \text{у арифметической формулы } \varphi \text{ есть доказательство в аксиоматике Пеано длины не больше } n\}$  является NP-полным.*

*Набросок доказательства.* Вначале докажем, что этот язык лежит в NP. Для этого нужен полиномиальный алгоритм проверки доказательства. Для этого нужно уметь проверять за полиномиальное время, что формула является аксиомой или выводится из предыдущих по одному из правил вывода. Во всех стандартных системах вывода это так и есть, поэтому мы не будем вдаваться в детали.

Теперь нужно объяснить, почему язык будет NP-трудным. Это можно сделать стандартным переводом с языка вычислений на машине Тьюринга на язык арифметических формул. Но проще всего использовать уже готовый инструмент — ЦЛП. Действительно, утверждение  $\exists \mathbf{x} \in \mathbb{N}^m \mathbf{Ax} = \mathbf{b}$  уже записано как арифметическая формула (точнее, нужно записать его в виде системы уравнений, а в каждом уравнении перенести отрицательные слагаемые в другую часть). Далее достаточно доказать, что любое верное равенство для конкретных чисел вида  $a \cdot b = c$  или  $a + b = c$  выводится за полиномиальное время от длины записи этих чисел. При этом, разумеется, числа  $a, b, c$  представлены не в унарной записи как  $SSS \dots S0$ , а в двоичной (в стандартной арифметике Пеано нужны некоторые ухищрения, но длина всё равно будет логарифмической). В виде такого доказательства можно представить стандартные алгоритмы сложения и умножения «в столбик» или какие-нибудь алгоритмы быстрой арифметики.  $\square$

## 3.6 Задачи поиска

С каждой задачей распознавания из NP связана задача поиска: по входу  $x$  найти сертификат  $s$ , такой что  $V(x, s) = 1$ , либо указать, что таких сертификатов нет. Ясно, что задача распознавания не сложнее задачи поиска: достаточно

посмотреть, найден сертификат или нет. В обратную сторону соотношение в общем случае неясно. Например, за полиномиальное время можно распознать, является ли число составным. Соответствующей задачей поиска является разложение числа на множители, а эту задачу решать не умеют. Более того, есть обширный класс задач, в которых вопроса о наличии сертификата вообще не стоит: он заведомо есть по принципу Дирихле или из других комбинаторных соображений. А вот задача поиска может быть сложной. На этом эффекте основано использование хеш-функций: если область определения больше, чем область значений, то у функции обязательно есть коллизия — пара аргументов, дающих одно значение. А вот быстрого алгоритма поиска коллизий для хороших хеш-функций неизвестно. На таких хеш-функциях, в частности, основано функционирование сети Bitcoin и других криптовалют. Более подробно о задачах поиска мы поговорим в главе 11.

Однако для **NP**-полных задач такого эффекта не возникает. Задача поиска не привносит дополнительной сложности. Разумеется, это нельзя показать сводимостью по Карпу, которая определена только для задач распознавания. Мы используем сводимость по Куку: если решатель задачи распознавания доступен как оракул, то можно решить и задачу поиска. Формализацию вычислений с оракулом мы изучим в следующей главе. Пока что будем представлять, что алгоритм за 1 шаг может получить ответ на требуемый вопрос. Нам потребуется ещё одна форма сводимости.

### 3.6.1 Сводимость по Левину

**Определение 3.28.** Задача  $A$  с верификатором  $V_A$  сводится по Левину к задаче  $B$  с верификатором  $V_B$ , если существуют полиномиально вычислимые функции  $f$ ,  $g$  и  $h$ , такие что  $V_A(x, y) = V_B(f(x), g(x, y))$  и  $V_B(f(x), z) = V_A(x, h(x, z))$ . Таким образом,  $g$  переделывает сертификат принадлежности  $x$  к  $A$  в сертификат принадлежности  $f(x)$  к  $B$  (а несертификат — в несертификат), а  $h$  — наоборот.

**Лемма 3.29.** Пусть языки  $A$  и  $B$  лежат в **NP** и задаются верификаторами  $W_A$  и  $W_B$ , соответственно. Пусть также  $A$  сводится по Карпу к  $B$ . Тогда существуют верификаторы  $V_A$  и  $V_B$ , для которых также имеет место сводимость по Левину.

*Доказательство.* Определим  $V_A$  и  $V_B$  так:<sup>6</sup>

$$\begin{aligned} V_A(x, (0, x', y)) &= \begin{cases} W_A(x', y), & \text{если } f(x') = f(x), \\ 0, & \text{иначе} \end{cases} \\ V_A(x, (1, x', y)) &= \begin{cases} W_B(f(x), y), & \text{если } f(x') = f(x), \\ 0, & \text{иначе} \end{cases} \\ V_B(z, (0, x, y)) &= \begin{cases} W_A(x, y), & \text{если } f(x) = z, \\ 0, & \text{иначе} \end{cases} \\ V_B(z, (1, x, y)) &= \begin{cases} W_B(z, y), & \text{если } f(x) = z, \\ 0, & \text{иначе} \end{cases} \end{aligned}$$

---

<sup>6</sup>Идея взята со страницы [cs.stackexchange.com/questions/2689/is-karp-reduction-identical-to-levin-reduction](https://cs.stackexchange.com/questions/2689/is-karp-reduction-identical-to-levin-reduction)

Верификаторы  $V_A$  и  $V_B$  распознают те же языки  $A$  и  $B$ . Действительно, если  $V_A$  принимает  $x$ , то либо  $x \in A$ , либо  $f(x) \in B$ , т.е. в любом случае  $x \in A$ . Аналогично, если  $V_B$  принимает  $z$ , то  $z \in B$  или  $z = f(x)$  для некоторого  $x \in A$ , то есть в любом случае  $z \in B$ . С другой стороны, теперь  $V_A$  и  $V_B$  принимаются просто одни и те же сертификаты, что и означает тривиальную сводимость по Левину. Например, если  $V_A(x, (0, x', y)) = 1$ , то  $W_A(x', y) = 1$  и  $f(x') = f(x)$ . Но тогда и  $V_B(f(x), (0, x', y)) = W_A(x', y) = 1$ . Аналогично разбираются все остальные случаи.  $\square$

### 3.6.2 Самосводимость в NP-полных задачах

**Теорема 3.30.** *Пусть  $A$  — NP-полная задача. Пусть машина может за один шаг получать ответ, лежит ли слово  $x$  в языке  $A$ . Тогда существует полиномиальный алгоритм, решающий задачу поиска для  $A$ .*

*Доказательство.* В силу леммы 3.29 можно считать, что и  $A$  сводится к 3SAT, и 3SAT сводится к  $A$  в смысле Левина. В большинстве случаев (в том числе в теореме Кука–Левина) это получается и без дополнительных конструкций. Поэтому достаточно свести  $A$  к 3SAT, решить задачу поиска для 3SAT и свести результат обратно.

Пусть дана формула  $\varphi$ , зависящая от переменных  $p_1, \dots, p_n$ . Вначале можно проверить выполнимость  $\varphi$  (при помощи оракула), и если формула невыполнима, такой ответ и вернуть. Если же формула выполнима, рассмотрим формулы  $\varphi'_1$  и  $\varphi''_1$ , полученные из  $\varphi$  фиксацией значений  $p_1 = 0$  и  $p_1 = 1$  соответственно. Хотя бы одна из них будет выполнима, какая именно, можно найти при помощи двух запросов к оракулу (или даже одного). Обозначим выполнимую формулу через  $\varphi_1$  и применим к ней ту же процедуру рекурсивно. Так будут определяться значения всех переменных, пока не останется только одна. Это будет база рекурсии: значение последней переменной можно найти непосредственной подстановкой обоих вариантов.  $\square$

Для некоторых задач из NP можно провести подобное рассуждение о «самосводимости» непосредственно, без сводимости к 3SAT и обратно. Приведём один пример.

**Утверждение 3.31.** *Если проверять наличие в графе клики данного размера можно за 1 шаг, то можно и найти максимальную клику за полиномиальное время.*

*Доказательство.* Тут утверждается немного больше, чем в предыдущей теореме, а именно ищется не просто клика заданного размера, а максимальная. Но это небольшое улучшение: последовательно проверяя наличие клик размера 2, 3, 4, ..., можно установить размер максимальной. (Можно чуть сократить перебор за счёт двоичного поиска). Теперь покажем, как найти максимальную клику, когда её размер  $k$  уже установлен, без использования общей конструкции.

Возьмём произвольную вершину  $v$  и удалим её из графа вместе со всеми рёбрами. Далее узнаем, сохранится ли в этом графе клика размера  $k$ . Если сохранится, то продолжим её искать в уменьшенном графе. Если же не сохранится, значит, вершина  $v$  в клике точно есть. Но тогда можно исключить из графа все вершины, не соединённые с  $v$ : их в клике точно нет. В сокращённом же графе будем искать клику размера  $k - 1$ , вместе с  $v$  она образует клику размера  $k$  в исходном графе.  $\square$



### 3.7 Как решать NP-полные задачи на практике

*В греческой мифологии Гидра отращивала три головы, когда Геракл срубал одну. Так и в сложности вычислений: как только теоретики находят препятствие для некоторого подхода, возникает несколько альтернативных. «Задача NP-полна? Ну что ж, сконцентрируемся на планарных графах, или на случайных графах, или будем аппроксимировать».*

Христос Пападимитриу, [105]

Многие вычислительные задачи, возникающие на практике, являются NP-полными. Поскольку рассчитывать на быстрые универсальные алгоритмы для решения таких задач не приходится, исследователи ищут другие методы:

- **Решение для частного случая.** Часто бывает, что в частном случае полиномиальный алгоритм существует. Например, общего решения для задачи о выполнимости нет, но есть полиномиальный алгоритм для задачи о выполнимости 2-КНФ. Поиск частного случая, под который подпадут интересующие задачи, и для которого есть полиномиальный алгоритм, может быть весьма нетривиальной задачей. При этом многие важные частные случаи остаются NP-полными, такие как SUBSET-SUM для задачи о рюкзаке или задача коммивояжёра на евклидовой плоскости.
- **Приближённое решение.** В задачах оптимизации обычно годится не только оптимальное решение, но и просто достаточно хорошее, например, лежащее в пределах нескольких процентов от оптимального. Задача оптимизации ставится так: имеется полиномиально вычисляемая функция  $F: \{0, 1\}^* \times \{0, 1\}^* \rightarrow \mathbb{N}$ . Требуется по входу  $x$  найти параметр  $s$ , при котором значение  $F(x, s)$  минимально (или максимально). Если оптимальное значение равно  $OPT$ , то решение с точностью  $\varepsilon$  означает поиска такого  $s$ , что  $F(x, s) < (1 + \varepsilon)OPT$  (для задачи максимизации  $F(x, s) > (1 - \varepsilon)OPT$ ). Можно также ставить требование иначе: найти число в полуинтервале  $[OPT, (1 + \varepsilon)OPT)$  (для задачи максимизации — в  $((1 - \varepsilon)OPT, OPT]$ ). Как правило, задачи оптимизации даже не лежат в NP: непонятно, как проверить оптимальность найденного решения. Однако в NP лежит класс задач, в которых нужно проверить, что оптимальное решение не меньше (в задачах минимизации — не больше) некоторого порога. Многие такие задачи NP-полны, например, оценки для кликового числа, числа независимости, хроматического числа графа, наличие во взвешенном графе короткого пути коммивояжёра, наличие для 3-КНФ набора, делающего истинным максимальное число дизъюнктов, совместность некоторого числа уравнений в задаче целочисленного линейного программирования и т.д.

В различных конкретных ситуациях получены совершенно различные результаты для приближённых задач оптимизации. Для одних задач построены полиномиальные приближённые алгоритмы, так называемые PTAS (polynomial-time approximation schemes — аппроксимирующие схемы, работающие за полиномиальное время). Примером такой задачи является задача коммивояжёра на евклидовой плоскости. Если алгоритм работает полиномиальное время не только от входа  $x$ , но и от параметра приближения  $\frac{1}{\varepsilon}$ , то аппроксимирующая схема называется полностью полиномиальной (fully polynomial-time, FPTAS). Примером такой задачи является

задача о рюкзаке. Для других задач есть алгоритмы для одних приближений, но поиск лучшего приближения уже является **NP**-трудной задачей.<sup>7</sup> Примером такой задачи является задача о максимизации числа выполненных дизъюнктов в 3-КНФ: легко построить алгоритм для  $\varepsilon = \frac{1}{8}$ , а вот более точное решение уже **NP**-трудно (в этом состоит утверждение знаменитой **РСР**-теоремы). Другой пример такой задачи — задача коммивояжёра в метрическом пространстве. Ну а для некоторых задач не существует приближённого алгоритма ни с какой степенью точности, например для задачи о клике или для задачи коммивояжёра с произвольными весами.

Построение приближённых алгоритмов, а также доказательство их отсутствия в тех или иных предположениях — активно развивающаяся область, важная и для теории, и для практики.

- **Алгоритмы, быстрые в среднем.** Несмотря на то, что в общей задаче линейного программирования известен полиномиальный метод эллипсоидов Хачияна, на практике используют симплекс-метод Данцига. В отдельных специально построенных случаях он работает экспоненциально долго, но в типичных ситуациях работает быстрее метода эллипсоидов. Другими примерами относительно быстрых на практике алгоритмов являются метод ветвей и границ для задач дискретной оптимизации и DPLL-алгоритм проверки выполнимости формулы. Разумеется, скорость работы зависит от того, какие именно входы будут встречаться на практике. Предполагается, что алгоритма, быстрого в среднем для любых адекватных распределений, для **NP**-полной задачи быть не может. Подробнее об этом говорится в главе 10.
- **Эвристики.** Для некоторых задач существуют алгоритмы, про которые не доказана не только оценка на скорость работы, но даже корректность. Тем не менее, на практике они могут быть весьма успешными. Такие алгоритмы называются эвристическими. Особенно много их в задачах оптимизации: генетические алгоритмы, метод пчелиного роя, метод муравьиной колонии, метод имитации отжига и т.д. Разумеется, часто они также ищут не точное, а приближённое решение. Но некоторые задачи решаются в точности. Например, задача коммивояжёра успешно решается для входов с десятками тысяч городов.

## 3.8 Исторические замечания и рекомендации по литературе

## 3.9 Задачи и упражнения

**3.1.** Язык называется *унарным*, если все слова в нём имеют вид  $1^k$ . Докажите, что если существует унарный **NP**-полный язык, то  $P = NP$ .

---

<sup>7</sup>**NP**-трудность тут понимается в таком смысле: описывается процедура, которая по слову из **NP**-полного языка строит задачу с оптимальным значением  $OPT$ , в по слову не из языка — с оптимальным значением не меньше  $(1+\varepsilon)OPT$ . Приближённый алгоритм с точностью выше  $\varepsilon$  позволил бы разделить один случай от другого.



# Глава 4

## Техника диагонализации

*Это доказательство замечательно [...] главным образом потому, что содержащийся в нём принцип можно просто распространить [...]*

Георг Кантор, [152]

Техника диагонализации, восходящая к Георгу Кантору, — мощный инструмент, позволяющий доказать многие факты в математике и логике. Так, с её помощью доказываются несчётность множества действительных чисел, неразрешимость проблемы самоприменимости, существование вычислимой непродолжаемой функции и другие утверждения, а также получаются некоторые логические парадоксы. Основная идея метода состоит в том, чтобы в бесконечной клетчатой таблице взять диагональ, изменить её в каждой точке и сделать вывод, что полученная последовательность в таблице отсутствует. На этой лекции мы изучим примеры использования этой техники к сложности вычислений, но также и увидим границы её применимости.

### 4.1 Иерархия по времени

*Сделайте то, что прошу я: помимо супруга Кронида,  
Дайте мне сына, чтоб силою был не слабее он Зевса.  
Но превзошёл бы его, как Кроноса Зевс превосходит*

Гомер, К Аполлону Пифийскому

В теории сложности вычислений зачастую можно легко доказать, что один класс вложен в другой (например,  $\mathbf{P} \subset \mathbf{NP}$ ), но строгость такого вложения не получается ни доказать, ни опровергнуть. Тем интереснее случаи, когда классы вложены заведомо строго. Один из инструментов, позволяющий доказать, например,  $\mathbf{P} \subsetneq \mathbf{E} \subsetneq \mathbf{EXP}$  — это теорема об иерархии по времени. Неформально говоря, теорема об иерархии гласит, что если функция  $g(n)$  растёт существенно быстрее, чем  $f(n)$ , то за время  $g(n)$  можно распознать больше языков, чем за время  $f(n)$ . Точная формулировка такая:

**Теорема 4.1** (об иерархии по времени). Пусть  $f: \mathbb{N} \rightarrow \mathbb{N}$  и  $g: \mathbb{N} \rightarrow \mathbb{N}$  — строго монотонные конструируемые по времени функции, такие что  $f(n) \log f(n) = o(g(n))$ . Тогда  $\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n))$ .

*Доказательство.* Идея доказательства состоит в том, чтобы перебирать все возможные алгоритмы, работающие не больше  $O(f(n))$ , и постепенно строить

язык так, чтобы все эти алгоритмы ошибались. При этом само построение работает за время, не сильно большее, чем  $f(n)$ .

Более формально, рассмотрим конструируемую по времени функцию  $h(n)$ , такую что  $f(n) = o(h(n))$  и  $h(n) \log h(n) = O(g(n))$ . Например, можно взять  $h(n) = \frac{g(n)}{\log g(n)}$ , тогда  $h(n) \log h(n) = \frac{g(n)}{\log g(n)} (\log g(n) - \log \log g(n)) \in [\frac{1}{2}g(n), g(n)]$ . С другой стороны,  $f(n) = o(h(n))$ , т.к. иначе не было бы выполнено  $f(n) \log f(n) = o(g(n))$ . Далее, занумеруем все машины Тьюринга, так чтобы каждая машина имела бесконечно много номеров, и по числу  $n$  можно было построить машину  $M_n$  за время  $O(n)$ .<sup>1</sup> Рассмотрим язык  $D = \{n \mid \text{машина } M_n \text{ или не останавливается на } n \text{ за время } h(n), \text{ или останавливается и выдаёт } 0\}$ . Покажем, что  $D \in \mathbf{DTIME}(g(n)) \setminus \mathbf{DTIME}(f(n))$ .

Вначале докажем, что  $D \in \mathbf{DTIME}(g(n))$ . Алгоритм можно построить непосредственно: сначала за время  $O(n)$  нужно получить программу  $M_n$ , потом вычислить  $h(n)$  (за время  $O(h(n))$ ), затем запустить  $M_n$  на такое число шагов и посмотреть, выполнено ли условие. Последний запуск требует времени  $O(h(n) \log h(n))$  т.к. машина с произвольным числом лент моделируется на машине с фиксированным числом лент. В силу  $h(n) \log h(n) = O(g(n))$  получаем, что общее время составит  $O(g(n))$ , что и требовалось.

Теперь пусть  $D \in \mathbf{DTIME}(f(n))$ . Пусть  $T$  — машина, распознающая  $D$ , работающая время  $Cf(n)$ . Найдём  $k$ , такое что  $T = M_k$  и при этом  $Cf(k) < h(k)$  (именно здесь мы пользуемся бесконечностью номеров каждой машины). Чему может быть равно  $M_k(k)$ ? Если  $k \in D$ , то  $M_k(k) = 1$ . Но это означает, что  $M_k(k)$  не останавливается за время  $h(k)$  или останавливается и выдаёт 0. Но по предположению  $M_k(k)$  равно 1, а не 0, и останавливается за время  $Cf(k) < h(k)$ , противоречие. Если же  $k \notin D$ , то с одной стороны должно быть  $M_k(k) = 0$ , а с другой стороны по определению  $D$  машина  $M_k$  останавливается на  $k$  за  $h(k)$  шагов и выдаёт 1, снова противоречие. Значит,  $D \notin \mathbf{DTIME}(f(n))$ , что и требовалось.  $\square$

Теорема об иерархии позволяет доказать, что не все функции конструируемые.

**Теорема 4.2.** *Существуют строго монотонные функции, не конструируемые по времени.*

*Доказательство.* По теореме об иерархии существует множество  $S$  из  $\mathbf{EEXP} \setminus \mathbf{EXP}$ . Рассмотрим функцию  $t(n) = 2n + \chi_S(n)$ , где  $\chi_S(n)$  понимается как характеристическая функция  $S$ , применённая к двоичной записи  $n$ . Коэффициент 2 гарантирует монотонность. Если бы  $t(n)$  была вычислима за время  $t(n)$ , то и  $\chi_S(n)$  была бы вычислима за время  $t(n)$ , то есть за время, экспоненциальное от длины двоичной записи  $n$ . Но в таком случае  $S \in \mathbf{EXP}$ , что противоречит выбору  $S$ .  $\square$

Можно доказать и более сильное утверждение.

**Теорема 4.3.** *Для любой вычислимой функции  $f(n)$  существует не конструируемая по времени функция  $t(n) > f(n)$ .*

*Идея доказательства.* Нужно взять  $S \in \mathbf{DTIME}(2^{2^{f(n)}}) \setminus \mathbf{DTIME}(2^{f(n)})$  и провести аналогичное рассуждение.  $\square$

<sup>1</sup>Подойдёт любое стандартное кодирование. Достаточно, чтобы не только каждая функция, но и каждая программа имела хотя бы один номер, так что можно дописывать недостижимые команды, пока не будет достигнут нужный размер.

Для построенных функций теорема об иерархии верна: они совсем чуть-чуть отличаются от конструируемых и мелкие отличия не влияют на класс **DTIME**. Однако в общем случае теорема об иерархии для функций, не конструируемых по времени, может быть неверной. Более того, верна такая теорема:

**Теорема 4.4** (Теорема Бородэна о разрыве, [24]). *Для любой всюду определённой вычислимой функции  $g(n) \geq n$  существует всюду определённая вычислимая функция  $t(n)$ , такая что  $\mathbf{DTIME}(g(t(n))) = \mathbf{DTIME}(t(n))$ .*

*Доказательство.* Идея состоит в следующем: каждая всюду определённая машина распознаёт некоторый язык за некоторое время. Мы будем строить  $t(n)$  так, чтобы гарантировать, что если уж машина не остановилась за  $t(n)$ , то она проработает хотя бы  $g(t(n))$ . Если бы машин было конечное число, это не составило бы труда. Мы будем с каждым шагом рассматривать всё больше и больше машин, и асимптотически это будет верно для всех.

Опишем конструкцию более формально. Определим через  $\text{time}_M(n)$  максимальное время работы машины  $M$  на входах длины  $n$  (если на каком-то входе  $M$  не останавливается, то значение равно  $\infty$ ). Далее, определим  $t(n)$  так:

$$t(n) = \min\{k > t(n-1) \mid \forall i \leq n \text{ time}_{M_i}(n) < k \text{ или } \text{time}_{M_i}(n) > ng(k)\}.$$

Заметим, что  $\text{time}_M(n)$  невычислима: из-за неразрешимости проблемы остановки невозможно понять, равна она какому-нибудь конечному числу или бесконечности. Однако предикат « $\text{time}_M(n) < k$  или  $\text{time}_M(n) > ng(k)$ » разрешим: достаточно запустить  $M$  на всех входах длины  $n$  на  $ng(k)$  шагов (а  $g$  вычислима по условию). Поэтому разрешим и предикат « $\forall i \leq n \text{ time}_{M_i}(n) < k$  или  $\text{time}_{M_i}(n) > ng(k)$ ». Более того, он верен для бесконечно многих  $k$ , поскольку  $\text{time}_{M_i}(n)$  принимает только конечное число значений для  $i$  от 1 до  $n$ . Значит, функция  $t(n)$  корректно определена как минимум непустого множества и вычислима.

Докажем, что  $t(n)$  искомая. Пусть неверно, что  $\mathbf{DTIME}(g(t(n))) = \mathbf{DTIME}(t(n))$ . Поскольку  $g(n) \geq n$ , это возможно, только если найдётся  $S$  из  $\mathbf{DTIME}(g(t(n))) \setminus \mathbf{DTIME}(t(n))$ . Пусть  $S$  распознаётся машиной  $M_q$  за время  $Cg(t(n))$ . Рассмотрим произвольное  $n > \max\{q, C\}$ . По определению  $t(n)$  для любого  $i \leq n$  выполнено  $\text{time}_{M_i}(n) < t(n)$  или  $\text{time}_{M_i}(n) > ng(t(n))$ . В частности,  $\text{time}_{M_q}(n) < t(n)$  или  $\text{time}_{M_q}(n) > ng(t(n))$ . Но второе невозможно, т.к.  $\text{time}_{M_q}(n) < Cg(t(n)) < ng(t(n))$ . Значит,  $\text{time}_{M_q}(n) < t(n)$  при всех  $n > q$ . Но это значит, что  $S \in \mathbf{DTIME}(t(n))$ , противоречие. Значит,  $\mathbf{DTIME}(g(t(n))) = \mathbf{DTIME}(t(n))$ , что и требовалось.  $\square$

## 4.2 Существование NP-промежуточных задач

Для большинства задач из **NP** известен либо полиномиальный алгоритм, либо доказательство **NP**-полноты. Однако известно несколько задач, про которых не известно ни того, ни другого. Прежде всего, это задача об изоморфизме графов, задача о разложении числа на множители и задача дискретного логарифмирования. Разумеется, в случае  $\mathbf{P} = \mathbf{NP}$  все эти задачи решаются за полиномиальное время. Однако ни про одну из них нет теоремы о том, что она именно **NP**-промежуточная в случае  $\mathbf{P} = \mathbf{NP}$ . Зато подобная теорема доказана про некоторую очень искусственную задачу, к которой мы сейчас и переходим.

**Теорема 4.5** (Ладнера, [88]). *Если  $\mathbf{P} \neq \mathbf{NP}$ , то существует язык из **NP**, не лежащий в **P** и не **NP**-полный.*

*Доказательство.* Представим доказательство в виде последовательности лемм. Вначале введём обозначение. Пусть  $H: \mathbb{N} \rightarrow \mathbb{N}$  — некоторая функция. Тогда  $\text{SAT}_H = \{\psi 01^{n^{H(n)}} : \varphi \in \text{SAT} \text{ и } |\psi| = n\}$ . Ясно, что для полиномиально (от  $n$ ) вычислимой  $H$  задача  $\text{SAT}_H$  будет лежать в **NP**: сначала полиномиальным алгоритмом можно проверить, имеет ли слово вид  $\psi 01^{n^{H(n)}}$ , где  $|\psi| = n$ , а потом проверить выполнимость  $\psi$  при помощи сертификата. Начнём изложение доказательства, вначале доказав утверждение, верное для любой  $H$ .

**Лемма 4.6.** *Если  $H$  вычислима за полиномиальное время,  $\lim_{n \rightarrow \infty} H(n) = \infty$  и  $\text{SAT}_H$  является NP-полным языком, то  $\text{SAT} \in \mathbf{P}$ .*

*Доказательство.* Заметим, что при известном  $n$  можно за полиномиальное время проверить, имеет ли слово  $x$  вид  $\psi 01^{n^{H(n)}}$ . Если не имеет, то точно  $x \notin \text{SAT}_H$ , иначе  $x \in \text{SAT}_H \Leftrightarrow \psi \in \text{SAT}$ . Если  $\text{SAT}_H$  является **NP**-полным, то  $\text{SAT}$  сводится к  $\text{SAT}_H$  некоторой конкретной функцией, увеличивающей длину в некоторое полиномиальное число раз  $n^c$ . Если  $n$  настолько большое, что  $H(n) > 2c$ , то последовательное сведение  $\text{SAT}$  к  $\text{SAT}_H$ , а потом обратно к  $\text{SAT}$  превратит формулу длины  $n$  в формулу длины  $\sqrt{n}$ . Порядка  $\log \log n$  таких операций достаточно, чтобы получить формулу константной длины, выполнимость которой можно проверить непосредственно.  $\square$

В дальнейшем будем считать, что

$$H(n) = \min\{\log \log n, \min\{d < \log \log n : \\ \forall x \in \{0, 1\}^{\leq \log n} M_d(x) = \text{SAT}_H(x) \text{ и работает } \leq d|x|^d \text{ шагов}\}\}.$$

Поясним, что имеется в виду. Внешний минимум нужен только для случая, когда множество под внутренним минимумом пустое, и оттого внутренний минимум равен  $\infty$ . Под записью  $\text{SAT}_H(x)$  мы понимаем значение характеристической функции. Таким образом, утверждение в второй строчке говорит, что машина  $M_d$  правильно распознаёт  $\text{SAT}_H$  для всех коротких  $x$  и при этом работает за полиномиальное время (впрочем, степень этого полинома равна номеру машины, т.е. достаточно велика). Заметим, что это определение рекурсивно, но при этом рекурсия раскрывается корректно: значение на  $n$  зависит только от значений на входах не больше  $\log n$ . Непосредственной реализацией можно получить алгоритм, вычисляющий  $H(n)$  за полиномиальное время.<sup>2</sup> Перейдём к следующим леммам.

**Лемма 4.7.** *Если  $\text{SAT}_H \in \mathbf{P}$ , то  $H$  ограничена.*

*Доказательство.* Действительно, если  $\text{SAT}_H \in \mathbf{P}$ , то  $\text{SAT}_H$  распознаётся некоторой машиной  $M_c$  за полиномиальное время. Можно считать, что  $c$  настолько велико, что этот полином меньше  $c|x|^c$ : нужно взять эквивалентную машину с

<sup>2</sup>Докажем, что  $H(n)$  можно вычислять за  $O(n^2)$ . Действительно, для вычисления  $H(n)$  нужно перебрать все  $d$  от 1 до  $\log \log n$ , все  $x$  длины не больше  $\log n$  и для каждой пары запустить  $M_d(x)$  на  $d|x|^d$  шагов. Всего запусков будет порядка  $n \log \log n$ , а каждый запуск займёт порядка  $(\log n)^{\log \log n}$  шагов, т.е. общее количество шагов будет порядка  $n^{1+\varepsilon}$ . Также нужно будет вычислять  $\text{SAT}_H(x)$  для всех  $x$  короче  $\log n$ . Для этого нужно знать  $H(1), \dots, H(\log n)$ . Для этих значений верно предположение индукции, т.е. каждое из них можно вычислить за  $O(\log^2 n)$ . Значит, их все можно вычислить за  $O(\log^3 n)$ . Также нужно решать задачу  $\text{SAT}$  для формул длины порядка  $\log n$ . Это можно делать полным перебором за время порядка  $n$ . Таким образом, все  $\text{SAT}_H(x)$  также вычисляются за время  $O(n^{1+\varepsilon})$ . Осталось сравнить все  $M_d(x)$  и  $\text{SAT}_H(x)$ , что делается за такое же время. Таким образом, времени  $n^2$  заведомо хватит.

достаточно большим номером. В таком случае для  $n > 2^{2^c}$  имеем  $H(n) \leq c$ , откуда  $H$  ограничена.  $\square$

Следующая лемма показывает, что верно и обратное: если  $H$  ограничена, то  $SAT_H \in P$ . Мы докажем её для более общей посылки, чтобы использовать в дальнейшем рассуждении.

**Лемма 4.8.** *Если  $H$  принимает некоторое значение бесконечно много раз, то  $SAT_H \in P$ .*

*Доказательство.* Пусть  $H(n) = k$  для бесконечно многих  $n$ . В таком случае  $M_k$  распознаёт  $SAT_H$  за время  $k|x|^k$  для всех  $x$  длины не больше  $\log n$ , и происходит это для бесконечно многих  $n$ . Так как  $n$  растёт неограниченно, то и  $\log n$  растёт неограниченно, значит,  $M_k$  распознаёт  $SAT_H$  за время  $k|x|^k$  просто для всех  $x$ . А это и означает, что  $SAT_H \in P$ .  $\square$

Наконец, последняя лемма:

**Лемма 4.9.** *Если  $H$  ограничена, то  $SAT \in P$ .*

*Доказательство.* Поскольку ограниченная функция принимает некоторое значение бесконечно много раз, то из предыдущей леммы  $SAT_H \in P$ . Но если  $H(n) \leq M$  при всех  $n$ , то сводимость  $\psi \mapsto \psi 01^{|\psi|^{H(|\psi|)}}$  будет полиномиальной, откуда  $SAT \leq_p SAT_H$  и  $SAT \in P$ .  $\square$

Теперь теорема доказывается так: если  $SAT_H \in P$ , то  $H$  ограничена, а тогда  $SAT \in P$  и  $P = NP$ , что противоречит предположению. Если  $H$  не стремится к бесконечности, то она принимает некоторое значение бесконечно много раз, и всё равно  $SAT_H \in P$ . Если  $H$  стремится к бесконечности, а  $SAT_H$  является  $NP$ -полной, то вновь  $SAT \in P$  и  $P = NP$ . Остаётся вариант, когда  $H$  стремится к бесконечности, но  $SAT_H \notin P$  и  $SAT_H$  не  $NP$ -полна, что и требовалось.  $\square$

## 4.3 Релятивизация утверждения $P = NP$

### 4.3.1 Вычисления с оракулом

*Будут они вопрошать мой оракул. И всем непреложно  
В храме моем благолепном начну подавать я советы*

Гомер, К Аполлону Пифийскому

В теории вычислимости известна концепция вычислений с *оракулом*, когда можно делать запросы к некоторому множеству или некоторой функции. Наличие оракула может расширить классы вычислимых функций и разрешимых множеств. В теории сложности вычислений оракулы обычно вычислимы, но запросы к ним выполняются за 1 шаг, что может существенно ускорить вычисление. Мы будем придерживаться следующей модели:

**Определение 4.10.** Машиной Тьюринга с оракулом  $A \subset \{0, 1\}^*$  называется машина, имеющая специальную ленту, называемой оракульной и три выделенных состояния  $q_{ask}$ ,  $q_{yes}$  и  $q_{no}$ . Если машина переходит в состояние  $q_{ask}$ , то на следующем шаге она переходит в состояние  $q_{yes}$  или  $q_{no}$  в зависимости от того, лежит ли содержимое оракульной ленты в множестве  $A$ . Изначально оракульная лента пуста. Если при запросе на ней записано не двоичное слово, то она переходит, например, в состояние  $q_{no}$ .

Если машина недетерминированная, то ответы оракула всё равно получаются детерминированно, а вот запросы к оракулу могут быть различными в зависимости от недетерминированного выбора. Для машин с оракулом можно определять сложностные классы:

**Определение 4.11.** Классом  $\mathbf{DTIME}^A(T(n))$  называется класс языков, которые распознаются на детерминированной машине Тьюринга с оракулом  $A$  за время  $O(T(n))$ . Классом  $\mathbf{NTIME}^A(T(n))$  называется класс языков, которые распознаются на недетерминированной машине Тьюринга с оракулом  $A$  за время  $O(T(n))$ . Используются стандартные обозначения  $\mathbf{P}^A = \bigcup_{c=1}^{\infty} \mathbf{DTIME}(n^c)$  и  $\mathbf{NP}^A = \bigcup_{c=1}^{\infty} \mathbf{NTIME}(n^c)$ .

Теорема о сертификатном определении  $\mathbf{NP}$  и вложенность  $\mathbf{P}$  в  $\mathbf{NP}$  тривиальным образом переносятся на аналогичные для классов с оракулом (как говорят, *релятивизируются*). А вот теорему Кука–Левина так перенести уже нельзя: ответы оракула не моделируются булевыми формулами. Более того, следующая теорема показывает, что никакой техникой, переносящейся на вычисления с оракулом, нельзя ни доказать, ни опровергнуть утверждение  $\mathbf{P} = \mathbf{NP}$ .

### 4.3.2 Теорема Бейкера–Джилла–Соловэя

**Теорема 4.12.** (Бейкер, Джилл, Соловэй, [15]) Для некоторого оракула  $A$  выполнено  $\mathbf{P}^A = \mathbf{NP}^A$ , а для некоторого другого оракула  $B$  выполнено  $\mathbf{P}^B \neq \mathbf{NP}^B$ .

Идея состоит в том, что за счёт использования оракула может расшириться как класс  $\mathbf{P}$ , так и класс  $\mathbf{NP}$ . Теорема утверждает, что для одного оракула она заведомо расширяется одинаково, а для другого по-разному. Первая идея состоит в том, чтобы в качестве  $A$  взять  $\mathbf{NP}$ -полную задачу. Однако это не сработает: класс  $\mathbf{P}$  с таким оракулом включит в себя и  $\mathbf{NP}$ , и  $\mathbf{coNP}$ , и ещё некоторые задачи, но  $\mathbf{NP}$  тоже расширится, причём в некоторых предположениях заведомо сильнее, см. подробнее раздел 6.4.

*Доказательство.* В качестве оракула  $A$  возьмём язык  $\mathbf{EXPCOM} = \{(M, x, 1^t) \mid M(x) = 1 \text{ и } M(x) \text{ работает не дольше } 2^t \text{ шагов}\}$ . Неформально говоря, этот оракул позволяет проводить экспоненциальные вычисления за 1 шаг. Равенство следует из того, что  $\mathbf{EXP} \subset \mathbf{P}^{\mathbf{EXPCOM}} \subset \mathbf{NP}^{\mathbf{EXPCOM}} \subset \mathbf{EXP}$ . Первое вложение выполнено, исходя из самой сути оракула: если  $A \in \mathbf{EXP}$  и принадлежность к  $A$  распознаётся машиной  $M$  за время  $2^{p(n)}$ , то достаточно запросить оракул о тройке  $(M, x, 1^{p(n)})$ . Второе вложение выполнено для любого оракула. Наконец, третье вложение выполнено, поскольку экспоненциального времени хватит как для перебора всех ветвей алгоритма, так и для непосредственного вычисления ответов на запросы к оракулу.

С оракулом  $B$  конструкция будет не такая явная. Идея состоит в том, чтобы построить  $B$  постепенно, так чтобы гарантировать ошибку для всех возможных полиномиальных алгоритмов. Для начала заметим, что при любом  $B$  язык  $U_B = \{1^n \mid B \cap \{0, 1\}^n \neq \emptyset\}$  лежит в  $\mathbf{NP}^B$ . Действительно, можно недетерминированно выбрать слово  $x \in \{0, 1\}^n$  и запросить оракул о принадлежности к нему этого слова. Мы будем строить  $B$  так, чтобы гарантировать  $U_B \notin \mathbf{P}^B$ .

Построение языка разделим на стадии. На каждой стадии определяется принадлежность к  $B$  только конечного числа слов. Более того, после каждой стадии для каждой длины либо про все слова известно, лежат ли они в  $B$ , либо ни про одно не известно. На стадии  $k$  выберем минимальную длину  $m$ , такую что про



слова этой длины ещё ничего не известно, и запустим машину  $M_k$  на входе  $1^m$  на  $\frac{2^m}{10}$  шагов. Если в процессе выполнения машина делает запросы к оракулу про слова, принадлежность которых уже установлена, отвечаем соответственно. Если она делает запросы к оракулу про новые слова, отвечаем «нет» и помечаем, что эти слова не лежат в  $B$ . Наконец, если машина даёт некоторый ответ, делаем так, чтобы она ошиблась: если ответ «да», то помечаем, что все слова длины  $m$  не лежат в  $B$  (это можно сделать, т.к. перед запуском ни одно не было помечено, а в процессе запуска могло помечаться только отсутствие в  $B$ ), а если ответ «нет», то выбираем некоторое слово длины  $m$ , не рассмотренное в ходе запуска (такое есть, поскольку всего слов  $2^m$ , а машина запросила не больше  $\frac{2^m}{10}$ ), и говорим, что оно лежит в  $B$ . Далее про все нерассмотренные слова длины  $m$ , а также всех длин, про которые машина делала запросы, говорим, что они не принадлежат  $B$ , и переходим к следующей стадии.

Ясно, что в ходе этого процесса про каждое слово рано или поздно будет сказано, лежит оно в  $B$  или нет (именно для этого мы выбирали минимальную длину  $m$ ; также заметим, что из конструкции получилось, что  $m \geq k$ ). Докажем, что для полученного  $B$  действительно  $U_B \notin P^B$ . Пусть это не так, и  $U_B$  распознаётся некоторым алгоритмом за время  $p(n)$ . Поскольку у каждого алгоритма бесконечно много номеров, выберем такой номер  $k$ , что  $\frac{2^m}{10} > p(m)$  при всех  $m \geq k$ . В таком случае на стадии  $k$  построения  $B$  мы запустим  $M_k$  на  $1^m$  для некоторого  $m \geq k$ . Поскольку время работы  $M_k$  меньше времени, на которое мы её запустим, она должна выдать некоторый ответ. Но  $B$  было построено таким образом, чтобы этот ответ был неправильным. Значит, никакого полиномиального алгоритма для распознавания  $U_B$  нет, что и требовалось.  $\square$

Заметим, что аккуратным анализом и небольшим уточнением предложенной конструкции можно доказать, что построенный язык  $B$  лежит в **EXP**.

### 4.3.3 Случайный оракул

Хотя из предыдущего доказательства может показаться, что найти оракул, при котором  $P^B \neq NP^B$  сложнее, чем оракул, при котором  $P^A = NP^A$ , это не совсем так. Для случайного оракула почти наверняка  $P^B \neq NP^B$ . Мы докажем эту теорему для двух распределений:

**Теорема 4.13.** Пусть  $C$  — случайный язык, полученный по следующему правилу: для каждой длины  $n$  с вероятностью  $\frac{1}{2}$ , независимо от остальных длин, в  $C$  нет ни одного слова этой длины, а с вероятностью ещё  $\frac{1}{2}$  — ровно одно слово, равномерно среди всех  $2^n$  слов. В таком случае с вероятностью 1 выполнено  $P^C \neq NP^C$ .

*Идея доказательства.* Нужно доказать, что с вероятностью 1 язык  $U_C$  не содержится в  $P^C$ . Идея состоит в том, что для достаточно большой длины  $m$  полиномиальный алгоритм сможет запросить оракул о небольшой доле слов длины  $m$ , с высокой вероятностью получит ответ «нет» про все эти слова и не сможет понять, есть ли в языке какое-нибудь слово такой длины. Любой его ответ будет ошибочным с вероятностью, близкой к  $\frac{1}{2}$ , уж точно больше  $\frac{1}{3}$ . Поскольку это происходит в бесконечном числе длин независимо друг от друга, хоть где-нибудь будет ошибка почти наверняка.  $\square$

**Теорема 4.14.** (Беннета–Джилла, [21]) Пусть  $C$  — случайный язык, в котором каждое слово содержится с вероятностью  $\frac{1}{2}$  независимо от остальных. В таком случае с вероятностью 1 выполнено  $P^C \neq NP^C$ .

*Доказательство.* Мы будем доказывать, что  $\mathbf{NP}^C \neq \mathbf{coNP}^C$  с вероятностью 1. Как следствие,  $\mathbf{P}^C \neq \mathbf{NP}^C$ : теорема о том, что  $\mathbf{P} = \mathbf{NP}$  влечёт  $\mathbf{NP} = \mathbf{coNP}$ , выполняется и для вычислений с оракулом.

Через  $C(x)$  будем обозначать характеристическую функцию оракула  $C$ . Рассмотрим функцию  $\xi_C: \{0, 1\}^* \rightarrow \{0, 1\}^*$ , заданную соотношением

$$\xi_C(x) = C(x1)C(x10)C(x100) \dots C(x10^{|x|-1}).$$

Эта функция сохраняет длину и легко вычислима при доступе к оракулу  $C$ . В частности, множество  $\text{RANGE}^C = \{y \mid \exists x \ y = \xi_C(x)\}$  лежит в  $\mathbf{NP}^C$ : достаточно недетерминированно угадать  $x$  и проверить  $\xi_C(x) = y$ . Мы докажем, что с вероятностью 1 оно не лежит в  $\mathbf{coNP}^C$ : интуитивно, чтобы доказать, что  $y$  не лежит в области значений функции, нужно её вычислить на всех входах, а их экспоненциальное число.

Поскольку оракул  $C$  случаен, для фиксированных  $x$  и  $y$  длины  $n$  событие  $\xi_C(x) = y$  происходит с вероятностью  $\frac{1}{2^n}$ . Более того, все эти события независимы: для разных  $x$  берутся значения оракула на разных входах. По теореме Пуассона предельное распределение количества прообразов  $y$  каждого слова будет пуассоновским с параметром 1. Иными словами, для любого  $y$

$$\lim_{n \rightarrow \infty} \Pr_C [y \text{ имеет ровно } k \text{ прообразов под действием } \xi_C] = \frac{1}{e k!}.$$

В частности, вероятности того, что  $y$  имеет ни одного прообраза или ровно один прообраз, стремятся к  $\frac{1}{e}$ . Можно показать, что при  $n \geq 5$  они будут заключены между 0.36 и 0.37. Обозначим через  $D_{n,0}$  множество всех оракулов  $C$ , для которых слово  $0^n$  не имеет прообразов, а через  $D_{n,1}$  — множество всех оракулов, для которых оно имеет ровно один прообраз, при этом отличный от  $0^n$ . Как уже было сказано, вероятности этих множеств стремятся к  $\frac{1}{e}$  (последнее требование не влияет на асимптотику и введено в технических целях). Дальнейшая идея такова: мы докажем, что вероятности того, что машина примет  $0^n$  в каждом из этих множеств, близки друг к другу. Поэтому вероятность ошибки для каждой машины отделена от нуля. А поскольку в существенно разных длинах результаты друг от друга не зависят, вероятность того, что ошибка случится хоть где-то, равна единице.

Пусть фиксирована нумерация всех недетерминированных машин Тьюринга:  $M_1, M_2, \dots$ . Обозначим через  $\alpha_{n,i,0}$  условную вероятность того, что машина  $M_i$  примет  $0^n$  за  $\leq \frac{2^n}{100}$  шагов, обращаясь к оракулу  $C \in D_{n,0}$ . Через  $\alpha_{n,i,1}$  обозначим аналогичную вероятность при условии  $C \in D_{n,1}$ . Вероятность того, что машина  $M_i$  даст неверный ответ на  $0^n$ , будет не меньше  $(1 - \alpha_{n,i,0}) \Pr [C \in D_{n,0}] + \alpha_{n,i,1} \Pr [C \in D_{n,1}] > 0.36(1 + \alpha_{n,i,1} - \alpha_{n,i,0})$ . Мы докажем, что  $\alpha_{n,i,1} \approx \alpha_{n,i,0}$ , откуда вероятность ошибки  $M_i$  на  $0^n$  будет больше  $\frac{1}{3}$ .

Идея заключается в том, чтобы изменить  $C \in D_{n,0}$  на  $C' \in D_{n,1}$ , так чтобы машина с большой вероятностью дала тот же ответ. При этом эта замена будет сохранять распределение вероятностей, поэтому вероятность ответа будет такая же, как на случайном  $C \in D_{n,1}$ . Более подробно, замена будет такой: выберем случайное  $z \neq 0^n$  длины  $n$  и исключим из  $C$  слова  $z1, z10, \dots, z10^{n-1}$ . В этом случае, очевидно,  $\xi_{C'}(z) = 0^n$ . При этом никакого  $w \neq z$ , такого что  $\xi_{C'}(w) = 0^n$ , быть не может, иначе и  $\xi_C(w) = 0^n$ . Значит,  $C' \in D_{n,1}$ . При этом  $C'$  мог получиться из  $2^n - 1$  различных  $C$ , а каждый  $C$  мог превратиться в  $2^n - 1$  различных  $C'$ . Поэтому преобразование  $C \mapsto C'$  сохраняет меру.

С другой стороны, машина  $M_i$  в процессе работы вдоль принимающей ветви ни разу не запросит оракул ни про одно слово из хотя бы 99% цепочек



$(z_1, z_{10}, \dots, z_{10^{n-1}})$ . Значит, с вероятностью не меньше 99% она на этой ветви даст одинаковый положительный ответ для  $C$  и  $C'$ . Но этот ответ будет истинным для  $C'$  и ложным для  $C$ . Отсюда и получится  $\alpha_{n,i,1} > 0.99\alpha_{n,i,0}$ .

Осталось заметить, что ошибочность  $M_i$  на  $0^n$  и ошибочность  $M_j$  на  $0^m$  при  $m > 2n$  — независимые события. Поскольку каждая машина ошибается на конкретной длине с вероятностью хотя бы  $\frac{1}{3}$ , каждая машина встречается в последовательности бесконечно много раз и каждая полиномиальная машина останавливается за  $\leq \frac{2^n}{100}$  при достаточно больших  $n$ , с единичной вероятностью каждая машина ошибётся хотя бы где-то.  $\square$

Беннет и Джилл выдвинули «гипотезу о случайном оракуле» (random oracle hypothesis): если какое-то соотношение между классами выполнено для случайного оракула, то оно выполнено и для вычислений без оракула. Эта гипотеза была опровергнута работами [122] и [29]. В первой из них доказано, что  $\mathbf{IP} = \mathbf{PSPACE}$ , а во второй — что это соотношение ложно для случайного оракула. Более подробно об интерактивных доказательствах и классе  $\mathbf{IP}$  рассказывается в главе 12.

#### 4.3.4 Нерелятивизируемость теоремы Кука–Левина

### 4.4 Исторические замечания и рекомендации по литературе

### 4.5 Задачи и упражнения

**4.1.** Докажите, что  $\mathbf{P}^A \subsetneq \mathbf{EXP}^A$  для любого оракула  $A$ .



## Глава 5

# Пространственная сложность

*Продолжительность времени определяется нашим восприятием. Размеры пространства обусловлены нашим сознанием. Поэтому, коли дух покоен, один день сравнится с тысячей веков, а коли помыслы широки, крохотная хижина вместит в себя целый мир*

Хун Цзычен, *Вкус корней*

На заре современной вычислительной техники оперативная память была самым дорогим элементом компьютера. Поэтому было очень важно экономить её при написании программ. Основной способ экономии — освобождение уже использованных участков памяти от более не нужных данных и перезапись нового. Возможность вторичного использования — принципиальная черта, отличающая память от времени. Глобальный (и нерешённый) теоретический вопрос заключается в том, даёт ли это реальное преимущество, например, можно ли на полиномиальной памяти решить задачи, не решаемые за полиномиальное время. Современная наука не знает ответа на этот вопрос. Мы познакомимся с тем, какие есть продвижения.

### 5.1 Сложностные классы, связанные с памятью

Если включать в используемую машиной Тьюринга память место, занятое входным словом, то память в любом случае получится как минимум линейной. Это не отражает ситуации, когда задача относительно простая. Например, конечный автомат по сути использует константную память, а не линейную. Поэтому подсчитывают только дополнительную по отношению к входу память. Моделируют это следующим образом: у машины есть отдельная входная лента, по которой она может перемещаться в обе стороны, но при этом не может ничего записывать. И есть одна или несколько рабочих лент. Используемая память считается как общее количество ячеек на рабочих лентах, на которые машина хоть раз указывала в процессе работы.

Также нам понадобится считать использованную память для машин, вычисляющих функции, а не распознающих языки. В этом случае биты выхода также подсчитывать не нужно. Это моделируется так: у машины есть ещё и лента для выхода, на которой можно двигаться только слева направо и записывать биты ответа. Альтернативно можно сказать, что каждый отдельный бит выхода можно подсчитать на ограниченной памяти.

Как и в теории временной сложности, от функций, ограничивающих зону работы программы, мы часто будем требовать конструируемости:

**Определение 5.1.** Функция  $s: \mathbb{N} \rightarrow \mathbb{N}$  называется *конструируемой по памяти*, если существует машина Тьюринга, которая по входу  $1^n$  вычисляет  $s(n)$ , используя память  $O(s(n))$ .

Как и раньше, все обычные функции будут конструируемы, но можно специальным образом построить и неконструируемые. Также неконструируемыми могут быть и слишком медленно растущие функции, например  $\log \log n$ . Можно было бы оставить только ограничения, растущие не медленнее логарифма, однако некоторые сверхмалые ограничения вполне осмысленны, например на константной памяти распознаются все регулярные языки (на самом деле только они). Теперь определим стандартные сложностные классы:

**Определение 5.2.** Пусть  $s(n)$  — неубывающая функция. Классом  $\mathbf{DSPACE}(s(n))$  называется класс языков, которые можно распознать на детерминированной машине Тьюринга, на любом входе длины  $n$  использующей  $O(s(n))$  ячеек на рабочих лентах.

**Определение 5.3.** Пусть  $s(n)$  — неубывающая функция. Классом  $\mathbf{NSPACE}(s(n))$  называется класс языков, которые можно распознать на недетерминированной машине Тьюринга, на любом входе длины  $n$  использующей  $O(s(n))$  ячеек на рабочих лентах (при любых исходах недетерминированного выбора).

Как и со временем, нас будут интересовать задачи, которые можно решить на полиномиальной памяти, без конкретизации полинома:

**Определение 5.4.** Классом  $\mathbf{PSPACE}$  называется  $\bigcup_{c=0}^{\infty} \mathbf{DSPACE}(n^c)$ . Классом  $\mathbf{NPSPACE}$  называется  $\bigcup_{c=0}^{\infty} \mathbf{NSPACE}(n^c)$ .

Докажем простейшие соотношения между классами:

**Теорема 5.5.** Имеют место соотношения  $\mathbf{DTIME}(t(n)) \subset \mathbf{NTIME}(t(n)) \subset \mathbf{DSPACE}(t(n)) \subset \mathbf{NSPACE}(t(n)) \subset \mathbf{DTIME}(2^{O(t(n))})$ . Последняя запись означает  $\bigcup_{c=0}^{\infty} \mathbf{DTIME}(2^{ct(n)})$ .

**Замечание 5.6.** Рассматривая класс  $\mathbf{DTIME}(t(n))$ , мы фактически говорим о том, что  $t(n) \geq n$ . Впрочем, вложение  $\mathbf{NSPACE}(t(n)) \subset \mathbf{DTIME}(2^{O(t(n))})$  выполнено уже для  $t(n) \geq \log n$ .

*Доказательство.* Первое соотношение мы уже доказывали. Второе следует из того, что за один шаг машина может занять только одну новую ячейку. Поэтому общее число занятых ячеек не больше числа шагов. Третье следует из того, что детерминированная машина — частный случай недетерминированной.

Прежде чем доказывать четвёртое соотношение, докажем более слабое:  $\mathbf{DSPACE}(t(n)) \subset \mathbf{DTIME}(2^{O(t(n))})$ . Для простоты будем считать, что машина имеет одну рабочую ленту, бесконечную только в одну сторону. Тогда если машина в ходе работы занимает на ленте не больше  $s$  ячеек, то количество всевозможных конфигураций не больше, чем  $a^s \cdot snq$ , где  $a$  — размер ленточного алфавита, а  $q$  — число состояний машины. Действительно, возможных состояний рабочей ленты  $a^s$ , есть  $s$  возможных позиций на рабочей ленте и  $n$  на входной,<sup>1</sup> а также  $q$  возможных

<sup>1</sup>Тут возникает вопрос, почему машина не может выходить за пределы входа на входной ленте. На этот вопрос есть два ответа. Во-первых, это противоречит сути определения: получается, что машина использует какие-то ячейки помимо тех, где записан вход. Таким образом, можно запретить перемещаться за края входа на уровне определения. Во-вторых, даже без запрета никакого преимущества машина за счёт этого не получит: можно составить другую машину, которая вместо того, чтобы ходить по пустым ячейкам на входной ленте, заведёт счётчик, насколько далеко и в какую сторону она вышла за край слова и будет его изменять по мере надобности. Можно адаптировать рассуждение к изменённой машине.

состояний. Если машина сделала больше  $a^s \cdot snq$  шагов и не остановилась, то некоторая конфигурация повторилась, а значит, машина уже не остановится никогда. Значит, любая останавливающаяся машина делает не больше такого числа шагов, а это как раз  $2^{O(s)}$ .

Для доказательства последнего соотношения нужно видоизменить доказательство, так чтобы оно подошло и для недетерминированных машин. Предыдущие рассуждения показывают, что каждая ветвь вычисления имеет длину  $2^{O(s)}$ . Однако общее число ветвей может быть и двойной экспонентой, поэтому просто все их перебрать не получится. Вместе с тем число *различных* конфигураций на этих ветвях будет лишь экспоненциальным, поэтому всё дерево обходить не нужно. Вместо этого мы запустим алгоритм динамического программирования. Простейшая реализация такова: заведём таблицу, в которой будем записывать про каждую конфигурацию, можно ли из неё добраться до принимающей. Изначально пометим все принимающие конфигурации. Далее будем много раз перебирать все конфигурации по циклу и про каждую смотреть, можно ли из неё добраться до некоторой уже помеченной. Если можно, то пометим и её. Рано или поздно случится одно из двух: либо начальная конфигурация будет помечена, либо на очередном проходе не будет помечена ни одна новая конфигурация. Более того, это случится не более чем через экспоненциальное число проходов. Поскольку каждый проход занимает экспоненциальное время, общее время останется экспоненциальным.  $\square$

**Следствие 5.7.**  $P \subset NP \subset PSPACE \subset NPSPACE \subset EXP$ .

Мы покажем, что вложение  $PSPACE \subset NPSPACE$  нестрогое: на самом деле эти классы равны. Про все остальные вложения ничего неизвестно, за исключением того, что  $P \neq EXP$  по теореме об иерархии по времени. Более того, для любой цепочки вложений, среди которых хотя бы одно строгое, можно построить оракул, при котором именно такая цепочка будет истинной.

Также отметим, что для вычислений с ограниченной памятью верна теорема об иерархии.

**Теорема 5.8.** Пусть  $f$  и  $g$  — возрастающие функции, конструируемые по памяти, причём  $f(n) = o(g(n))$  и  $f(n) \geq \log n$ . Тогда  $DSPACE(f(n)) \subsetneq DSPACE(g(n))$ .

*Доказательство.* Доказательство в целом будет повторять идею теоремы об иерархии по времени. Вложение классов очевидно. Покажем, почему оно строгое. Пусть  $h(n) = \frac{f(n)+g(n)}{2}$ . Тогда  $h(n)$  также конструируема по времени и при этом  $f(n) = o(h(n))$  и  $h(n) = o(g(n))$ . Пусть  $M_1, M_2, \dots$  — нумерация всех машин Тьюринга. Рассмотрим машину  $D$ , которая на входе  $n$  запускает  $M_n(n)$ , ограничивая её зону работы  $h(n)$  ячейками. В отличие от теоремы для времени, нужно отдельно проконтролировать, что машина не заикнется. Для этого нужно завести счётчик шагов и проверять, что он не превысил  $a^{h(n)}h(n)nq$ . Если вычисление  $M_n(n)$  закончилось с некоторым ответом, то  $D$  возвращает противоположный. Если же вычисление заикнулось (т.е. счётчик был превышен), то  $D$  возвращает любой ответ, например 0 для определённости.

Машина  $D$  распознаёт некоторый язык. Покажем, что он лежит в  $DSPACE(g(n)) \setminus DSPACE(f(n))$ . Вначале посчитаем, сколько места заняло вычисление. Зона работы машины  $M_n$  искусственно ограничена  $h(n)$  ячейками, счётчик дополнительно займёт  $O(h(n))$  ячеек, а в силу конструируемости  $h$  вычисление максимального значения счётчика также будет выполнено на такой памяти. Моделирование на универсальной машине Тьюринга увеличит память в константу

раз. Поскольку  $h(n) = o(g(n))$ , язык будет распознан на памяти  $O(g(n))$ . С другой стороны, пусть он распознаётся на памяти  $cf(n)$ . Это делает некоторая машина  $M_k$ . Можно считать, что  $k$  достаточно большое, чтобы выполнялось  $cf(k) < h(k)$ . Но в таком случае  $D(k)$  вычисляет  $M_k(k)$  и обращает результат. Поэтому  $M_k$  не может распознавать тот же язык, что и  $D$ . Полученное противоречие показывает, что этот язык не лежит в  $\mathbf{DSPACE}(f(n))$ , поэтому теорема доказана.  $\square$

## 5.2 Теорема Сэвича

Для работы с ограниченной памятью удобно использовать конфигурационный граф, который мы неявно уже ввели в рассказе о динамическом программировании. Его вершинами будут все конфигурации для конкретного входа, а рёбра — возможность перехода из одной конфигурации в другую за один шаг работы машины на этом входе. Для определённости будем считать, что в графе есть ровно одна принимающая конфигурация (например, добавим её искусственно и пустим в неё рёбра из всех конфигураций с принимающим состоянием). В таком случае (недетерминированная) машина принимает слово тогда и только тогда, когда в графе есть путь из начальной вершины в принимающую. Фактически, рассказанный выше алгоритм динамического программирования отвечал на этот вопрос для каждой конфигурации в графе, из-за чего он использовал экспоненциальную память. Покажем, что память можно сэкономить, хитрым образом освобождая и вновь используя (но с проигрышем по времени из-за того, что многие ответы приходится вычислять повторно).

**Теорема 5.9 (Сэвич).** *Если  $s(n) \geq \log n$ , то  $\mathbf{NSPACE}(s(n)) \subset \mathbf{DSPACE}(s(n)^2)$ .*

*Доказательство.* Итак, нам нужно понять, есть ли путь между двумя фиксированными вершинами  $u$  и  $v$  в графе экспоненциального размера  $2^N$ . Заметим, что если путь есть, то есть и путь длины не больше числа вершин в графе.<sup>2</sup> Обозначим через  $R(x, y, k)$  предикат существования пути не длиннее  $2^k$  из вершины  $x$  в вершину  $y$ . В таком случае имеем следующее рекурсивное соотношение:  $R(x, y, k)$  верно тогда и только тогда, когда для некоторой вершины  $z$  выполнены одновременно  $R(x, z, k-1)$  и  $R(z, y, k-1)$ . Таким образом, получается следующий рекурсивный алгоритм: перебираем все  $z$ , для каждого вычисляем истинность  $R(x, z, k-1)$ , запоминаем результат и на той же памяти вычисляем  $R(z, y, k-1)$ . Далее возвращаем конъюнкцию двух результатов. Базой рекурсии будет случай  $k = 0$ : в этом случае нужно проверить, что либо  $x$  и  $y$  совпадают, либо из  $x$  в  $y$  можно попасть за один шаг машины Тьюринга. Ясно, что это требует совсем небольшой дополнительной памяти. На каждом шаге рекурсии требуется запомнить только текущую вершину  $z$ , что требует памяти  $O(s(n))$ . (В этом месте мы используем нижнее ограничение на  $s(n)$ : число вершин в конфигурационном графе равно  $O(n2^s)$ , что будет равно  $2^{O(s(n))}$  только при  $s(n) \geq \log n$ ). Глубина рекурсии составит также  $O(s(n))$ , поэтому общая использованная память будет равна  $O(s(n)^2)$ , что и требовалось.  $\square$

Поскольку квадрат полинома также является полиномом, мы получаем

**Следствие 5.10.**  $\mathbf{PSPACE} = \mathbf{NPSPACE}$ .

<sup>2</sup>Это верно в любом графе, поскольку в более длинных путях найдётся цикл, который можно удалить, но в данном случае циклов вообще быть не может, потому что тогда в вычислении была бы бесконечная ветвь, не приводящая к остановке.

## 5.3 PSPACE-полнота

### 5.3.1 Общая теория

Как уже говорилось, мы не умеем доказывать  $\text{NP} \neq \text{PSPACE}$  и даже  $\text{P} \neq \text{PSPACE}$ . Однако, как и в классе  $\text{NP}$ , в классе  $\text{PSPACE}$  есть «самые сложные» задачи, которые называются **PSPACE-полными**. Общая идея такая же как и раньше: язык должен лежать в **PSPACE**, а любой другой язык из **PSPACE** должен к нему сводиться. Вопрос лишь в том, какую сводимость использовать: не нужно ли наложить на сводящую функцию условие полиномиальности по памяти? Разумеется, не нужно: нас же интересует отделимость **PSPACE** от  $\text{P}$ , поэтому сводимость должна быть из более маленького класса. Поэтому оставим сводимость по Карпу.

**Определение 5.11.** Язык  $B$  называется **PSPACE-трудным**, если для любого языка  $A$  из **PSPACE** выполнено  $A \leq_p B$ . Язык называется **PSPACE-полным**, если он **PSPACE-труден** и лежит в **PSPACE**.

Как и прежде, выполнены простые утверждения, которые остаются в качестве упражнений:

**Утверждение 5.12.** Если  $B$  является **PSPACE-трудным** и  $B \leq_p C$ , то  $C$  также **PSPACE-труден**.

**Утверждение 5.13.** Если  $B$  является **PSPACE-трудным** и лежит в  $\text{P}$  (в  $\text{NP}$ ), то  $\text{P} = \text{PSPACE}$  (соответственно,  $\text{NP} = \text{PSPACE}$ ).

### 5.3.2 Первые примеры полных задач

**PSPACE-полных** задач известно не так много, как **NP-полных**, но всё же существенное количество. Вначале рассмотрим две задачи, **PSPACE-полнота** которых получается из самого определения.

**Определение 5.14.** Языком **SPACETMSAT** называется множество  $\{(M, x, 1^s) \mid M(x) = 1 \text{ и } M(x) \text{ занимает не больше } s \text{ ячеек памяти}\}$ .

**Теорема 5.15.** Язык **SPACETMSAT** является **PSPACE-полным**.

*Доказательство.* Принадлежность к **PSPACE** доказывается непосредственно: нужно запустить  $M(x)$ , ограничив зону работы величиной  $s$  и контролируя, что машина не заиклилась. Если машина остановилась и выдала 1, нужно тоже выдать 1. Если же она выдала 0, попыталась выйти за пределы зоны или заиклилась, то нужно выдать 0. Построенный алгоритм занимает зону  $O(s)$ , то есть полиномиален по памяти.

Если же  $A \in \text{PSPACE}$ , то он распознаётся некоторой машиной  $M$  на памяти  $p(n)$ . В таком случае  $A$  сводится к **SPACETMSAT** при помощи функции  $x \mapsto (M, x, 1^{p(n)})$ .  $\square$

**Определение 5.16.** Языком **SUCCINCTPATH** называется множество  $\{(\varphi, u, v) \mid \text{в графе, построенном по формуле } \varphi, \text{ есть путь из } u \text{ в } v\}$ . Граф по формуле строится таким образом:  $u$  и  $v$  есть слова из  $n$  битов, а формула  $\varphi$  зависит от  $2n$  переменных. Ребро между  $u$  и  $v$  проводится в случае, когда  $\varphi(u, v) = 1$ .

Нас будет интересовать случай, когда длина формулы  $\varphi$  также полиномиальна от  $n$ . Таким образом, граф имеет экспоненциальный размер, но описывается полиномиальной формулой. Именно этим объясняется слово “succinct” — «сжатый» в названии языка.

**Теорема 5.17.** *Язык SUCCINCTPATH является PSPACE-полным.*

*Доказательство.* Вначале докажем, что он лежит в **PSPACE**. Точнее, мы докажем, что он лежит в **NPSPACE** и воспользуемся теоремой Сэвича. Идея состоит в запуске недетерминированного блуждания длины  $2^n$ . Вначале положим  $u_0 = u$ . На  $i$ -ом шаге, имея  $u_i$ , выберем недетерминированно вершину  $u_{i+1}$  и проверим  $\varphi(u_i, u_{i+1}) = 1$ . Если проверка прошла, переходим к следующему шагу. Иначе возвращаем ответ 0. Если оказалось, что  $u_{i+1} = v$ , то возвращаем 1, если же число шагов превысило  $2^n$ , то возвращаем 0. Если путь есть, то на некоторой ветви машина каждый раз будет выбирать его рёбра и потому вернёт 1. Если пути нет, то на любой ветви машина либо придёт в тупик, либо заикнется и в обоих случаях вернёт 0. Дополнительная память нужна для хранения счётчика, текущей вершины и вычисления  $\varphi$ . Во всех случаях она полиномиальна. Поэтому язык лежит в **NPSPACE**, а значит, и в **PSPACE**.

Теперь докажем **PSPACE**-полноту. Идея заключается в том, чтобы по машине, распознающей язык, построить граф конфигураций, а в качестве  $u$  и  $v$  взять начальную и принимающую конфигурации соответственно. Нужно доказать, что соседство в графе можно выразить короткой формулой. Фактически, мы это уже делали в теореме Кука–Левина: каждая вершина в графе представляет собой некоторую конфигурацию, и нужно проверить, что из одной конфигурации в другую можно пройти за один шаг. Небольшая трудность заключается в том, что теперь у машины заведомо больше одной ленты, но основная идея та же: изменится лишь ограниченное число символов в окрестности указателей, причём изменится в соответствии с программой. Технические детали остаются в качестве упражнения. Отметим лишь, что размер формулы действительно получается полиномиальным от длины исходного  $x$ .  $\square$

### 5.3.3 Булевы формулы с кванторами

Рассматривая теорию **NP**-полноты, мы доказали полноту задачи **SAT**, а затем уже её сводили к другим. В теории **PSPACE**-полноты тоже будет такая задача: **TQBF**.

**Определение 5.18.** Языком **TQBF** называется множество булевых формул  $\varphi$ , таких что для некоторого  $x_1 \in \{0, 1\}$  найдётся  $x_2 \in \{0, 1\}$ , такое что для некоторого  $x_3 \in \{0, 1\}$  найдётся  $\dots$  (цепочка чередующихся кванторов по всем переменным)  $\varphi(x_1, x_2, x_3, \dots)$  истинна.

Буквы **TQBF** означают “totally quantified boolean formulae”, по-русски эту задачу называют задачей о булевых формулах с кванторами и иногда обозначают **БФК**. Заметим, что требование строгого чередования кванторов излишне: если изначально два одноимённых квантора идут подряд, то можно между ними вставить квантор по новой переменной, от которой  $\varphi$  на самом деле не зависит. (Если требовать обязательного вхождения новой переменной, можно взять конъюнкцию  $\varphi$  с  $z \vee \neg z$ ). Формально можно сказать, что язык **TQBF'**, в котором кванторы могут не чередоваться, сводится к обычному **TQBF**.

**Теорема 5.19.** *Язык TQBF является PSPACE-полным.*



*Доказательство.* Сначала покажем, что  $\text{TQBF} \in \text{PSPACE}$ . Действительно, можно запустить рекурсивный алгоритм. Базой рекурсии будет следующее: если значения всех переменных заданы, то значение формулы можно просто вычислить. Далее, пусть мы уже умеем вычислять истинность утверждений вида  $\exists x_{i+1} \forall x_{i+2} \dots \varphi(x_1, \dots, x_i, x_{i+1}, \dots)$  для некоторого чётного  $i$  и любых значений  $x_1, \dots, x_i$  или утверждений вида  $\forall x_{i+1} \exists x_{i+2} \dots \varphi(x_1, \dots, x_i, x_{i+1}, \dots)$  для некоторого нечётного  $i$  и любых значений  $x_1, \dots, x_i$ . Тогда совершив два рекурсивных запуска и взяв конъюнкцию (дизъюнкцию) результатов, мы сможем вычислить истинность утверждений вида  $\forall x_i \exists x_{i+1} \forall x_{i+2} \dots \varphi(x_1, \dots, x_i, x_{i+1}, \dots)$  (соответственно,  $\exists x_i \forall x_{i+1} \exists x_{i+2} \dots \varphi(x_1, \dots, x_i, x_{i+1}, \dots)$ ) для любых значений  $x_1, \dots, x_{i-1}$ . Сделав рекурсию на  $n$  уровней, мы вычислим ответ к исходной задаче. На каждом уровне мы используем лишь константную память, поэтому общая память будет полиномиальной (и даже линейной).

Теперь докажем **PSPACE**-полноту. Для этого сведём **SUCCINCTPATH** к **TQBF**. Будем постепенно строить формулу  $R(x, y, k)$ , означающую, что из вершины  $x$  в вершину  $y$  есть путь не длиннее  $2^k$ . Формула  $R(x, y, 0)$  строится непосредственно: она означает  $x = y \vee \varphi(x, y)$ . Далее, заметим, что  $R(x, y, k)$  эквивалентно тому, что для некоторого  $z$  выполнено  $R(x, z, k-1)$  и  $R(z, y, k-1)$ : в качестве  $z$  нужно взять вершину посередине пути. Однако если просто написать  $\exists z (R(x, z, k-1) \wedge R(z, y, k-1))$  и запустить индукцию, то итоговая формула получится экспоненциальной длины. Вместо этого нужно записать следующее:  $\exists z \forall t \forall u ((t = x \wedge u = z) \vee (t = z \vee u = y)) \rightarrow R(t, u, k-1)$ . Эта формула эквивалентна предыдущей, однако по сравнению с формулой для  $k-1$  увеличение составит не два раза, а 30 символов. Таким образом, общая длина формулы будет полиномиальной (и даже линейной) от  $k$ . Однако в полученной формуле кванторы стоят в глубине. Для окончательного сведения нужно привести её к предварённой нормальной форме, вынеся все кванторы наружу.  $\square$

Как и в случае с **NP**-полнотой, можно считать, что формула из задачи **TQBF** после отбрасывания кванторов находится в 3-КНФ. Это делается так же, как и раньше: нужно завести новую переменную для каждой подформулы и записать все соотношения в виде 3-КНФ. По всем новым переменным нужно поставить кванторы существования после кванторов по старым переменным.

Заметим, что задачу **TQBF** можно интерпретировать как игру. Сначала первый игрок выбирает значение  $x_1$ , потом второй игрок выбирает значение  $x_2$ , потом первый игрок выбирает значение  $x_3$ , и так далее. Когда значения всех переменных определены, вычисляется значение формулы  $\varphi$ . Если она оказалась истинной, выиграл первый игрок, если ложной — второй. В таком случае  $\varphi \in \text{TQBF}$  тогда и только тогда, когда в этой игре побеждает первый игрок: он может так выбрать  $x_1$ , что как бы ни выбрал второй значение  $x_2$ , первый может так выбрать значение  $x_3$ , и так далее, чтобы в конце концов  $\varphi$  была истинной. Более скрупулёзный анализ показывает, что вообще любой язык из **PSPACE** можно представить как множество выигрышных стратегий в некоторой игре. Подробнее мы про это поговорим на следующей лекции, а пока докажем **PSPACE**-полноту одной конкретной игры.

### 5.3.4 Обобщённая игра в города

- А теперь ты, Федя, говори на «Д».
- Воржута. — Почему? — Я там сидел.

«Джентльмены удачи», диалог из фильма

Известна игра в города, в которой двое по очереди называют города, причём каждый следующий должен начинаться на ту же букву, на которую заканчивается предыдущий: Москва → Архангельск → Кустанай → Йошкар-Ола → ...<sup>3</sup> Проигрывает тот, кто не может сделать ход. На практике проигрыш случается из-за ограниченного знания городов конкретным игроком, но в теории список может попросту исчерпаться. Формализовать игру можно так: дан граф городов, проведены рёбра между двумя городами, если второй начинается на ту же букву, на которую заканчивается первый. Двое по очереди двигают фишку по рёбрам, ставить фишку на уже посещённую вершину нельзя. Проигрывает тот, кто не может сделать ход. В обычной игре граф удовлетворяет такому свойству: если есть рёбра  $a \rightarrow c$ ,  $a \rightarrow d$  и  $b \rightarrow c$ , то есть и ребро  $b \rightarrow d$ . Мы обобщим игру, отказавшись от каких-либо условий на граф. Кроме того, будем фиксировать начальную вершину.

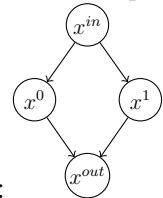
**Определение 5.20.** Языком **GG** называется множество  $\{(G, x) \mid \text{в обобщённой игре в города на графе } G \text{ с начальной вершиной } x \text{ выигрывает первый игрок}\}$ . Игра происходит следующим образом: изначально фишка ставится в вершину  $x$ , затем двое поочерёдно сдвигают её по рёбрам, при этом запрещено сдвигать фишку в вершину, где она уже была. Проигрывает тот, кто не может сделать ход.

Буквы **GG** обозначают “Generalized Geography” — обобщение игры в географию, т.е. в города.

**Теорема 5.21.** Язык **GG** является **PSPACE**-полным.

*Доказательство.* Полиномиальный по памяти алгоритм для проверки принадлежности к **GG** строится, как и раньше рекурсивно. Для разнообразия докажем  $\mathbf{GG} \in \mathbf{PSPACE}$  при помощи  $\mathbf{GG} \leq_p \mathbf{TQBF}$ . Действительно, пусть дан граф с  $n$  вершинами. Тогда в игре будет заведомо сделано не более  $n$  ходов. Можно считать, что  $n$  ходов делается в любом случае, а проигрывает тот, кто первым ошибётся. Тогда условие выигрыша запишется так:  $\exists v_1 \forall v_2 \exists v_3 \dots \text{Wins}_1(v_1, v_2, \dots, v_n)$ . Формула  $\text{Wins}_1$  говорит о том, что первый не сделал ни одного неверного хода, т.е. для любого нечётного  $i$  в случае, когда все вершины  $v_0 = x, v_1, v_2, \dots, v_{i-1}$  различны и образуют путь в графе, то в графе есть и ребро  $v_{i-1} \rightarrow v_i$ , причём  $v_i$  отличается от всех предыдущих. Длина такой формулы для конкретного  $i$  при простейшей реализации будет порядка  $i^2$ , а общая длина всех формул (т.е. длина формулы  $\text{Wins}_1$ ) — порядка  $n^3$  (написать квантор по  $i$  нельзя, нужно брать конъюнкцию).

Теперь докажем, что  $\mathbf{TQBF} \leq_p \mathbf{GG}$ . Напомним, что про формулу  $\varphi$  можно считать, что она приведена к 3-КНФ. Начнём строить граф. Для каждой пере-



менной  $x$  заведём 4 вершины:  $x^{in}, x^0, x^1, x^{out}$ , — соединённые вот так:

Далее соединим эти вершины в цепочку, для каждого  $i$  проведя ребро из  $x_i^{out}$  в  $x_{i+1}^{in}$ . В случае нечётного  $n$  добавим ещё одну вершину  $y$  и ребро  $x_n^{out} \rightarrow y$ . В случае чётного  $n$  будем считать  $y = x_n^{out}$ . Если формула  $\varphi$  была конъюнкцией

<sup>3</sup>Эта цепочка взята из социальной рекламы 1995 года, <https://www.youtube.com/watch?v=FmD8nyOGSYg>

дизъюнктов  $C_1, \dots, C_m$ , то заведём по вершине для каждого дизъюнкта и добавим рёбра  $y \rightarrow C_j$ . Наконец, если  $C_j$  имеет вид  $x_p^a \vee x_q^b \vee x_r^c$ , то проведём рёбра из  $C_j$  в каждую из вершин  $x_p^a, x_q^b, x_r^c$  (напомним, что мы используем обозначения  $x_p^0 = \neg x_p$  и  $x_p^1 = x_p$ ). Начальной вершиной будет  $x_1^{in}$ .

Покажем, что выигрыш в игре с формулой равносителен выигрышу в игре на графе. Выбор значения  $a$  для переменной  $x_i$  мы будем отождествлять с ходом из  $x_i^{in}$  в  $x_i^{1-a}$  на графе. За счёт рёбер вида  $x_i^{out} \rightarrow x_{i+1}^{in}$  очередность хода в  $in$ -вершинах будет чередоваться. Таким образом, первые  $3n$  ходов на графе (или  $3n - 1$  для чётного  $n$ ) определяются значениями некоторого набора. Если в исходной игре у первого игрока есть выигрышная стратегия, то формула будет выполнена. Значит, в каждом дизъюнкте будет хотя бы один истинный литерал. Какой бы дизъюнкт ни выбрал второй игрок (а именно он ходит в вершине  $y$ ), первый сможет найти в нём истинный литерал и сходить в соответствующую вершину (по ходу выбора значений занимались вершины, соответствующие ложным литералам). Поскольку следующая  $out$ -вершина занята, у второго игрока не будет хода, и он проиграет. Если же выигрышная стратегия есть у второго игрока, то формула будет ложна. Значит, в ней будет хотя бы один ложный дизъюнкт, именно эту вершину второй игрок и выберет. После этого у первого уже не будет хода, поэтому он проиграет.  $\square$

## 5.4 Вычисления на логарифмической памяти

*Дарси был маленьким созданием с птичьим мозгом, в котором никогда не помещается больше одной мысли сразу*

Редьярд Киплинг, *Рикки-Тикки-Тави*

**Определение 5.22.** Классом **L** называется **DSPACE**( $\log n$ ). Классом **NL** называется **NSPACE**( $\log n$ ). Классом **coNL** называется множество языков  $A$ , таких что  $\bar{A} \in \text{NL}$ .

Из теоремы 5.5 следует, что  $L \subset \text{NL} \subset \text{P}$ . К сожалению, ни про одно из вложений не известно, строго ли оно. Теорема Сэвича говорит лишь о том, что  $\text{NL} \subset \text{DSPACE}(\log^2 n)$ . А теорема об иерархии по памяти позволяет заключить, что  $\text{L} \neq \text{PSPACE}$  (и даже  $\text{L} \neq \text{DSPACE}(\log^2 n)$ ).

### 5.4.1 Примеры языков из L

Приведём примеры задач, которые решаются на логарифмической памяти. Сначала приведём несколько примеров из арифметики.

**Утверждение 5.23.** Язык  $\text{LE} = \{(x, y) \mid x \leq y\}$  принадлежит **L**. (Имеется в виду, что числа  $x$  и  $y$  записаны в двоичной записи, возможно, с ведущими нулями.)

*Доказательство.* Вначале нужно сравнить количество битов в  $x$  и  $y$ . Для этого нужно пройти по записи  $x$  до первой единицы, завести счётчик и увеличивать его на единицу при чтении нового бита, пока запись не кончится. Затем нужно сделать то же самое для  $y$ . Полученные счётчики нужно сравнить. Поскольку они имеют логарифмическую длину, их можно сравнить совсем простым алгоритмом, например, вычитая по единице из каждого счётчика, пока один не обратится в ноль. Можно запустить и рекурсию.

Теперь, если в записи одного из чисел битов больше, то оно и будет больше. Иначе нужно поочерёдно сравнивать соответствующие биты, пока один не будет больше. Тогда число, из которого получен этот бит, будет больше. Если же все биты совпали, значит, и числа равны.  $\square$

**Утверждение 5.24.** Язык  $\text{ADD} = \{(x, y, z) \mid x + y = z\}$  принадлежит  $\mathbf{L}$ .

*Доказательство.* На логарифмической памяти можно реализовать обычный, «школьный» алгоритм сложения в столбик. Для этого нужно складывать соответствующие биты записи  $x$  и  $y$  с конца, прибавлять бит переноса и сравнивать последний бит результата с соответствующим битом  $z$ , а первый запоминать как бит переноса. Если всё совпало, значит пример верный, иначе ошибочный. Небольшая трудность состоит в устройстве адресации: на ленте нельзя оставлять пометки, какие биты уже проверены. Вместо этого нужно завести два счётчика: в одном будет храниться количество обработанных битов, другой будет служить для отсчёта до соответствующего бита  $x$ ,  $y$  или  $z$ . Поскольку каждый счётчик не превышает  $n$ , достаточно будет логарифмического числа битов.  $\square$

**Утверждение 5.25.** Язык  $\text{MULT} = \{(x, y, z) \mid x \cdot y = z\}$  принадлежит  $\mathbf{L}$ .

*Доказательство.* Непосредственное применение алгоритма умножения в столбик тут невозможно, поскольку он предполагает рисование квадратичной таблицы. Однако всю её рисовать и не нужно. Дело в том, что при нумерации битов с конца, начиная с нуля, бит  $z_i$  есть последний бит суммы  $x_i y_0 + x_{i-1} y_1 + \dots + x_0 y_i$ , к которой прибавлен перенос с предыдущих этапов. Для заданного  $i$  такую сумму можно посчитать при помощи двух счётчиков (или даже одного). Сложить с переносом тоже можно на небольшой памяти. Последний бит результата должен совпасть с битом ответа, а всё предыдущее является переносом на следующий этап. Таким образом, для всех операций хватит логарифмической памяти, что и требовалось.  $\square$

Теперь перейдём к примеру из теории формальных языков. Напомним, что правильной скобочной последовательностью называется последовательность открывающих и закрывающих скобок, которую можно построить из пустой последовательности при помощи конкатенации двух уже построенных последовательностей или заключения уже построенной последовательности в скобки. Можно рассмотреть аналогичное определение для нескольких типов скобок: при втором способе получения новой последовательности скобки должны быть одного типа. Ясно, что если в последовательности  $2n$  скобок, то всего их не больше  $n$  типов, поэтому можно считать, что последовательность записана  $2n(\lceil \log n \rceil + 1)$  битами, а парные скобки — это числа  $2k$  и  $2k + 1$ .

**Утверждение 5.26.** Язык правильных скобочных последовательностей для нескольких типов скобок лежит в  $\mathbf{L}$ .

*Доказательство.* Если скобки только одного вида, то нужно воспользоваться теоремой о скобочном балансе: последовательность правильная тогда и только тогда, когда в любом начальном отрезке открывающих скобок не меньше, чем закрывающих, а во всей последовательности их поровну. Нужно пройти один раз по последовательности, подсчитывая скобочный баланс и следя за выполнением условия. На подсчёт баланса требуется логарифмическое число битов.

Для нескольких видов такой алгоритм уже не пройдёт. Например, последовательность  $([])$  неправильная, но со скобочным балансом всё хорошо. Выход

состоит в том, чтобы проверять баланс скобок не только во всей последовательности, но и внутри каждой пары соответствующих друг другу скобок. Для этого нужно по каждой открывающей скобке найти соответствующую ей закрывающую. Для этого нужна идти вправо по последовательности, подсчитывая баланс только по скобкам этого вида. Скобка, на которой он обнулится, и будет соответствующей. (Если такой не нашлось, то последовательность точно неправильная). Далее нужно проверить баланс скобок между найденными (можно проверить по каждому виду в отдельности, но достаточно по всем сразу). Для реализации этого алгоритма требуется небольшое число счётчиков логарифмического размера. Формальное доказательство корректности алгоритма остаётся в качестве упражнения.  $\square$

Наконец, разберём один пример из теории графов.

**Утверждение 5.27.** *Язык  $\text{TREE} = \{G \mid \text{неориентированный граф } G \text{ является деревом}\}$  лежит в  $\mathbf{L}$ .*

Как известно, дерево — это связный граф, в котором число рёбер на единицу меньше числа вершин. Второе условие проверить совсем легко. Первое на самом деле тоже проверяется на логарифмической памяти, поскольку на такой памяти можно проверять наличие пути. Однако это совсем недавний результат (теорема Рейнгольда), использующий сложную технику. Мы изложим доказательство, основанное на другой идее.

*Доказательство.* Вначале мы всё-таки проверим, что рёбер на единицу меньше, чем вершин. Также проверим, что в нём нет изолированных вершин. Тогда либо граф является деревом, либо несвязен, и в каждой компоненте связности строго меньше  $n - 1$  ребра. Дальнейшая идея доказательства основана на специальном обходе графа. Если дерево положить на плоскость и обходить его «по правилу левой руки», то путь пройдёт по каждому ребру дважды: в одну и в другую сторону — прежде чем заиклиться. Таким образом, длина цикла будет равна  $2e$ , где  $e$  — число рёбер. При этом более длинным цикл быть не может ни для какого графа: если одно ребро повторилось вместе с направлением, то и дальнейшее будет повторяться. Таким образом, для дерева длина цикла равна  $2(n - 1)$ , а для других графов меньше. Проверять планарность можно по критерию Понтрягина–Куратовского, используя теорему Рейнгольда, однако мы хотим избежать сложной техники. Поэтому вместо проверки планарности мы определим обход без опоры на плоскую структуру.

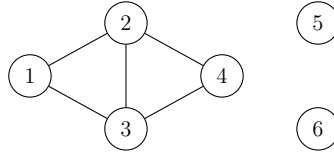
Пусть  $u_1, \dots, u_n$  — все вершины графа. Будем считать, что индексы упорядочены по циклу: после  $n$  идёт снова 1, и так далее. Будем говорить, что за ребром  $(u_i, u_j)$  следует такое ребро  $(u_j, u_k)$ , что  $k$  — минимальный номер вершины, смежной с  $u_j$ , больший  $i$  (с учётом заикливания). В частности, если  $u_j$  — висющаяся вершина, то после ребра  $(u_i, u_j)$  идёт  $(u_j, u_i)$ . Утверждается, что для дерева цикл при таком обходе будет всегда иметь длину  $2(n - 1)$ . Это доказывается по индукции: если в дереве одно ребро, то это очевидно. Иначе в дереве есть висющаяся вершина  $u_i$ . Пусть она соединена с вершиной  $u_j$  и находится в списке её соседей между  $u_k$  и  $u_l$ . (Т.е.  $k$  — максимальный номер соседа  $u_j$ , меньший  $i$ , а  $l$  — минимальный номер соседа  $u_j$ , больший  $i$ , с учётом заикливания. При этом случай  $k = l$  не исключён). Для дерева с вычеркнутой вершиной  $u_i$  по предположению индукции есть обход длины  $2(n - 2)$ , в котором есть цепочка  $(u_k, u_j, u_l)$ . По описанной процедуре в новом графе она должна замениться на цепочку  $(u_k, u_j, u_i, u_j, u_l)$ . Таким образом, прибавится два ребра, и общее число рёбер в пути составит  $2(n - 1)$ , что и требовалось.

Таким образом, алгоритм получается такой:

- Проверить, что число рёбер на единицу меньше числа вершин;
- Проверить, что нет изолированных вершин;
- Запустить обход, начиная с произвольного ребра, проверить, что оно впервые повторится через  $2(n - 1)$  шагов.

Если все три проверки прошли, то граф — дерево, иначе нет. Для каждого из шагов достаточно логарифмической памяти: для первого и второго это совсем просто, для третьего это объясняется тем, что достаточно хранить только начальную, текущую и предыдущую вершины, а также общее число уже сделанных шагов.  $\square$

Заметим, что проверка на наличие изолированных вершин обязательна. На-



пример, для такого графа:

(6 вершин, 5 рёбер), и третье (обход устроен так:  $1-2-3-4-2-1-3-2-4-3-1$ ).

#### 5.4.2 NL-полнота

Как часто бывает, если два класса не получается разделить, в большем из них возникают самые сложные — полные — задачи. Все остальные задачи должны к ним сводиться методами из меньшего класса. Таким образом, полиномиальная сводимость уже не подойдёт, нужна сводимость, логарифмическая по памяти. Напомним, что такая функция должна вычисляться машиной Тьюринга с тремя лентами: входной, с которой можно только читать и по которой можно двигаться в обе стороны, рабочей для чтения и записи и выходной только для записи битов ответа слева направо. Нам также понадобится альтернативная характеристика:

**Утверждение 5.28.** *Функция  $f$  вычислима на логарифмической памяти тогда и только тогда, когда языки  $D_f = \{(x, k) : |f(x)| \leq k\}$  и  $E_f = \{(x, i) \mid f(x)_i = 1\}$  лежат в  $\mathbf{L}$ , при этом максимальное  $k$ , при котором  $(x, k) \in D_f$ , ограничено полиномом от  $|x|$ . ( $D$  от слова *domain* — область определения,  $E$  от слова *evaluation* — вычисление значения).*

*Доказательство.* Пусть функция вычислима на логарифмической памяти. Тогда  $D_f$  распознаётся так: нужно запустить вычисление  $f(x)$ , но вместо вывода битов ответа на выходную ленту просто считать их количество. Если это количество достигло  $k$ , вернуть ответ 1, если не достигло, а вычисление закончилось, то вернуть ответ 0. Поскольку счётчик не превысит  $k$ , дополнительная память составит не больше  $\log k$ , поэтому вся занятая память останется логарифмической. Язык  $E_f$  распознаётся аналогично: нужно запустить вычисление  $f(x)$  и также считать биты ответа, дожидаясь  $i$ -го. Если он оказался равным 1, вывести ответ 1. Если он оказался равным 0 или вычисление завершилось раньше, вывести ответ 0. Также заметим, что машина, работающая на логарифмической памяти, может сделать лишь полиномиальное число шагов, отсюда следует условие на максимальное  $k$ , при котором  $(x, k) \in D_f$ .

Обратно, пусть  $D_f$  и  $E_f$  разрешимы на логарифмической памяти. Тогда значение  $f(x)$  вычисляется так. Нужно запустить цикл по  $k = 1, 2, 3, \dots$ . Если

$(x, k) \in D_f$  и  $(x, k) \in E_f$ , вывести 1. Если  $(x, k) \in D_f$  и  $(x, k) \notin E_f$ , вывести 0. Если  $(x, k) \notin D_f$ , закончить работу. На каждом шаге можно использовать одну и ту же память, поэтому общая память будет логарифмической. Причём, в силу того, что  $k \leq \text{poly}(|x|)$ , она будет логарифмической не только от  $|x| + k$ , но и просто от  $|x|$ .  $\square$

Заметим, что условие на максимальное значение  $k$  существенно. Например,  $D_f = \{(x, k) : k \leq 2^{|x|}\}$  и  $E_f = \emptyset$  лежат в  $\mathbf{L}$ , однако функция  $f : x \mapsto 0^{2^{|x|}}$  не может быть вычислена на логарифмической памяти.

Альтернативное определение вычислимости на логарифмической памяти позволяет доказать следующее:

**Утверждение 5.29.** *Композиция функций, вычислимых на логарифмической памяти, вычислима на логарифмической памяти.*

*Доказательство.* Нам будет удобно комбинировать оба определения. Пусть для  $f$  верно, что множества  $D_f$  и  $E_f$  лежат в  $\mathbf{L}$ , а  $g$  логарифмически вычислима в исходном смысле. Покажем, как можно вычислить  $g(f(x))$ . Запустим обычное вычисление  $g$  на входе  $f(x)$ . Заведём счётчик, который показывает, на какой бит  $f(x)$  в данный момент указывает машина, и значение этого бита. Если машина, вычисляющая  $g$ , сдвигается вдоль своего аргумента, мы соответствующим образом изменяем счётчик и вычисляем новое значение при помощи алгоритмов, разрешающих  $D_f$  и  $E_f$ . Поскольку длина  $f(x)$  полиномиальна, счётчик займёт логарифмическую память. Вычисленные биты  $f(x)$  храниться не будут, а будут перезаписываться. Таким образом, память останется логарифмической.  $\square$

Теперь мы подошли к определению сводимости.

**Определение 5.30.** Язык  $A$  логарифмически сводится к языку  $B$ , если существует функция  $f$ , вычислимая на логарифмической памяти, такая что для всех  $x$  выполнено  $x \in A$  тогда и только тогда, когда  $f(x) \in B$ . Обозначение  $A \leq_l B$ .

Как и для полиномиальной сводимости, для логарифмической верны простые свойства:

**Утверждение 5.31.**

1. *Логарифмическая сводимость рефлексивна:  $A \leq_l A$ ;*
2. *Логарифмическая сводимость транзитивна: если  $A \leq_l B$  и  $B \leq_l C$ , то  $A \leq_l C$ ;*
3. *Если  $A \in \mathbf{L}$ , а  $B \neq \emptyset$  и  $B \neq \{0, 1\}^*$ , то  $A \leq_l B$ ;*
4. *Если  $B \in \mathbf{L}$  и  $A \leq_l B$ , то  $A \in \mathbf{L}$ ;*
5. *Если  $B \in \mathbf{NL}$  и  $A \leq_l B$ , то  $A \in \mathbf{NL}$ .*

*Доказательство.* Первое утверждение очевидно: тождественная функция вычислима на логарифмической памяти (и даже на константной). Третье утверждение доказывается так: фиксируются  $b_1 \in B$  и  $b_0 \notin B$ . Тогда функция  $f(x)$ , равная  $b_0$ , если  $x \notin A$ , и  $b_1$ , если  $x \in A$ , будет вычислима на логарифмической памяти: именно столько нужно, чтобы понять, какой из случаев имеет место. Второе и четвёртое утверждение непосредственно следуют из утверждения 5.29. Наконец, пятое утверждение доказывается аналогично утверждению 5.29, отличие состоит в том, что «внешняя» функция будет вычисляться недетерминированной машиной.  $\square$

Перейдём к понятию полноты.

**Определение 5.32.** Язык  $B$  называется **NL**-трудным, если для любого  $A \in \mathbf{NL}$  выполнена сводимость  $A \leq_l B$ . Язык называется **NL**-полным, если он **NL**-труден и лежит в **NL**.

Как всегда, выполнены простые утверждения:

**Утверждение 5.33.** Если  $B$  является **NL**-трудным и  $B \leq_l C$ , то  $C$  также **NL**-труден.

**Утверждение 5.34.** Если  $B$  является **NL**-трудным и лежит в  $\mathbf{L}$ , то  $\mathbf{L} = \mathbf{NL}$ .

### 5.4.3 Сертификатное определение **NL**

Для класса **NP** у нас было 2 определения: через недетерминированные машины и через сертификаты. Для класса **NL** тоже удобно иметь сертификатное определение. Однако если перенести определение механически, то получится не **NL**, а снова **NP**: например, истинность 3-КНФ на некотором наборе можно проверить на логарифмической памяти. Нужно ограничить возможности машины по обращению к сертификату. Это делается так: машина может читать сертификат только один раз, слева направо.

**Теорема 5.35.** Язык  $A$  лежит в **NL** тогда и только тогда, когда существует машина  $V$  с двумя входами, к первому из которых доступ не ограничен, а второй можно читать только один раз слева направо, использующая логарифмическую память на рабочей ленте, такая что  $x \in A$  тогда и только тогда, когда для некоторого  $s$  выполнено  $V(x, s) = 1$ .

*Доказательство.* Пусть  $A$  лежит в **NL**, т.е. распознаётся некоторой недетерминированной машиной  $M$  на логарифмической памяти. В таком случае сертификат  $s$  будем понимать как последовательность выборов для машины  $M$ , а машина  $V$  будет просто моделировать машину  $M$  на соответствующей ветви и вернёт то же значение. Если  $x \in A$ , то есть принимающая ветвь, именно её и возьмём в качестве сертификата. Если  $x \notin A$ , то все ветви отвергающие, значит и любой сертификат будет отвергнут.

Обратно, если для языка есть логарифмический верификатор, то можно написать недетерминированную машину, которая будет угадывать его биты один за другим по мере надобности. Она использует ровно такое же количество памяти и выдаст единицу, если сможет угадать верный сертификат.  $\square$

Заметим, что из-за того, что машина использует логарифмическую память, сертификат в этом определении автоматически получается полиномиальной длины: все последующие биты верификатор всё равно не успеет прочесть.

### 5.4.4 **NL**-полнота задачи PATH

**Определение 5.36.** Языком **PATH** называется множество  $\{(G, s, t) \mid \text{в ориентированном графе } G \text{ есть путь из } s \text{ в } t\}$ .

Подобно тому, как язык **SUCCINCTPATH** был полным в классе **PSPACE**, язык **PATH** будет полным в классе **NL**.

**Теорема 5.37.** Язык **PATH** является **NL**-полным.



*Доказательство.* Вначале докажем, что  $\text{PATH} \in \text{NL}$ . Для этого удобно использовать сертификатное определение: сам путь и будет сертификатом. Верификатор хранит лишь последнюю вершину. Получив новую, он проверяет, что соответствующее ребро есть в графе, и затем перезаписывает последнюю вершину. Поэтому использованная память будет логарифмической.

Теперь докажем  $\text{NL}$ -полноту. Идея доказательства проста: мы построим конфигурационный граф для машины на данном входе, а в качестве  $s$  и  $t$  возьмём начальную и принимающую вершины. Надо лишь показать, что сводимость будет логарифмической. Независимо от входа все конфигурации можно пронумеровать единообразно: конфигурация будет определяться содержимым рабочей ленты, состоянием машины и положением указателей на обеих лентах. Нужно понять на логарифмической памяти, можно ли из одной конфигурации перейти в другую для данного входа. Для этого нужно проверить, что символы на ленте совпадают везде, кроме того места, на которое указывает машина, а в этом месте переход осуществлён в соответствии с программой. И то, и другое делается на логарифмической памяти.  $\square$

### 5.4.5 $\text{NL} = \text{coNL}$

Когда речь шла о полиномиальном времени, то помимо открытого вопроса о равенстве  $\text{P}$  и  $\text{NP}$  был и открытый вопрос о равенстве  $\text{NP}$  и  $\text{coNP}$ . Долгое время исследователи верили и в то, что  $\text{NL}$  и  $\text{coNL}$  не совпадают, однако это оказалось неверным.

**Теорема 5.38** (Иммерман–Селепченъи).  $\text{NL} = \text{coNL}$ .

*Доказательство.* Ясно, что достаточно доказать, что  $\overline{\text{PATH}} \in \text{NL}$ . Действительно, поскольку  $\text{PATH}$  является  $\text{NL}$ -полным, то  $\overline{\text{PATH}}$  является  $\text{coNL}$ -полным. Тогда любой язык из  $\text{coNL}$  сводится к  $\overline{\text{PATH}}$ , который по предположению лежит в  $\text{NL}$ . Отсюда  $\text{coNL} \subset \text{NL}$ . Но тогда верно и обратное включение: если  $A \in \text{NL}$ , то  $\overline{A} \in \text{coNL}$ , из предыдущего  $\overline{A} \in \text{NL}$ , а тогда  $A \in \text{coNL}$ .

Итак, требуется доказать  $\text{PATH} \in \text{NL}$ . Идея состоит в следующем: если известно количество всех вершин, достижимых из  $s$ , то можно доказать, что  $t$  недостижима. Для этого нужно предъявить список всех достижимых из  $s$  вершин в порядке возрастания, вместе с путями от  $s$  до этих вершин. Но откуда узнать количество достижимых вершин? Нужно по индукции посчитать, сколько вершин достижимо за  $i$  ходов. Опишем эту процедуру подробнее.

Обозначим через  $c_i$  количество вершин, достижимых из  $s$  путём из не более чем  $i$  рёбер. Тогда  $c_0 = 1$ , а  $c_n$  — общее число вершин, достижимых из  $s$ : если какой-то путь есть, то есть и путь не длиннее числа вершин. Сертификат будет состоять из последовательных сертификатов для подсчёта  $c_1, \dots, c_n$  и, наконец, сертификата того, что  $t$  не достижимо за  $n$  шагов.

Пусть верификатор уже убедился в истинности значения  $c_i$ . Тогда про каждую вершину можно предоставить сертификат достижимости или недостижимости за  $i + 1$  шаг. Сертификатом достижимости вершины  $v$  будет путь. Для его проверки нужно хранить  $i$ , текущую вершину и общее число пройденных рёбер в пути. Нужно проверять наличие каждого ребра, совпадение последней вершины с  $v$  и то, что общее число рёбер не превысило  $i + 1$ . Сертификат недостижимости устроен сложнее. Он состоит из отсортированного списка всех вершин, достижимых за  $i$  шагов, вместе с соответствующими путями. Для его проверки нужно хранить  $i$ ,  $c_i$ , текущее число уже проверенных вершин, текущую проверяемую вершину, текущую вершину в пути до проверяемой вершины

и общее число пройденных вершин в пути до проверяемой вершины — всего логарифмическое число битов. Проверка заключается в том, что новая проверяемая вершина по порядку больше старой (т.е. список и правда отсортирован), она и правда достижима не больше чем за  $i$  шагов (т.е. путь корректен) и в том, что из неё не ведёт ребра в вершину  $v$ . Отсортированность списка препятствует вторичному употреблению той же вершины, поэтому верификатор получит доказательства достижимости  $c_i$  различных вершин за  $\leq i$  шагов, ни из одной из которых нельзя пройти в  $v$ . Если значение  $c_i$  было корректным, то  $v$  и правда недостижимо за  $i + 1$  шаг. Таким образом, верификатор ожидает увидеть для каждой вершины сертификат достижимости или недостижимости за  $i + 1$  шаг. Чтобы перебор был полным, вершины должны подаваться в порядке возрастания. В таком случае, подсчитав число вершин, для которых был получен сертификат достижимости, верификатор узнает  $c_{i+1}$ .

Начав с известного  $c_0$ , верификатор через  $n - 1$  раунд узнает  $c_{n-1}$ . Теперь ему не обязательно вычислять  $c_n$ , достаточно получить сертификат недостижимости только для  $t$ . Если он верный, значит  $(G, s, t) \notin \text{PATH}$ , что и требовалось.  $\square$

#### 5.4.6 Другие NL-полные задачи

В это разделе мы посмотрим на примеры задач из **NL**, к которым можно свести **PATH**. Таким образом, они будут **NL**-полными.

**Утверждение 5.39.** Язык  $\overline{\text{PATH}}$  является **NL**-полным.

*Доказательство.* Мы уже доказали  $\overline{\text{PATH}} \in \text{NL}$ , осталось доказать полноту. Действительно, пусть  $A \in \text{NL}$ . Тогда в силу теоремы Иммермана–Селепченки  $\overline{A}$  также лежит в **NL**. Тогда  $\overline{A} \leq_l \text{PATH}$ , а значит,  $A \leq_l \overline{\text{PATH}}$ . Таким образом, **NL**-полнота языка  $\overline{\text{PATH}}$  установлена.  $\square$

**Утверждение 5.40.** Язык  $\text{SCONNECTED} = \{G \mid \text{ориентированный граф } G \text{ сильно связан}\}$  является **NL**-полным.

*Доказательство.* Вначале покажем, что **SCONNECTED** лежит в **NL**. Действительно, достаточно проверить, что из каждой вершины ведёт ориентированный путь в каждую другую. Сертификатом будет набор соответствующих путей, например в лексикографическом порядке по парам вершин.

Теперь сведём **PATH** к **SCONNECTED**. Пусть дан ориентированный граф  $G$  и вершины  $s$  и  $t$  в нём. Построим новый граф  $G'$ , добавив рёбра из любой вершины в  $s$ , а из  $t$  в любую вершину. Если в графе  $G$  был путь из  $s$  в  $t$ , то в графе  $G'$  будет путь из любой вершины в любую: сначала ребро из  $a$  в  $s$  (если  $a \neq s$ ), потом цепочка из  $s$  в  $t$ , потом ребро из  $t$  в  $b$  (если  $b \neq t$ ). Если же граф  $G'$  сильно связан, то в нём есть путь из  $s$  в  $t$ . Рассмотрим фрагмент этого пути, в котором  $s$  и  $t$  встречаются только в начале и в конце, соответственно. В этом пути добавленных рёбер быть не может, значит он был и в старом графе. Таким образом, в графе  $G$  есть путь из  $s$  в  $t$  тогда и только тогда, когда граф  $G'$  сильно связан. Мы построили сводимость, осталось проверить, что она логарифмическая. Это просто: в новом графе есть ребро  $(u, v)$ , если либо оно было в исходном графе, либо  $u = t$ , либо  $v = s$ . Всё это легко проверяется на логарифмической памяти.  $\square$

**Утверждение 5.41.** Язык  $2\text{SAT} = \{\varphi \mid \varphi \text{ — выполнимая формула в виде 2-КНФ}\}$  является **NL**-полным.

*Доказательство.* Вначале покажем, что  $2SAT \in NL$ . Для этого покажем, как можно решить  $2SAT$  при помощи  $PATN$ . Пусть  $\varphi$  — 2-КНФ, зависящая от переменных  $x_1, \dots, x_n$ . Построим по ней граф  $G_\varphi$  с  $2n$  вершинами, которые мы будем обозначать  $x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n$ . Если  $\varphi$  содержит дизъюнкт  $(x_i \vee x_j)$ , добавим в граф рёбра  $\bar{x}_i \rightarrow x_j$  и  $\bar{x}_j \rightarrow x_i$ . Смысл такой: если формула выполнена, но  $x_i$  ложна, то  $x_j$  должна быть истинной, и наоборот. Аналогично для дизъюнкта  $(\bar{x}_i \vee x_j)$  добавим рёбра  $x_i \rightarrow x_j$  и  $\bar{x}_j \rightarrow \bar{x}_i$ , а для дизъюнкта  $(\bar{x}_i \vee \bar{x}_j)$  — рёбра  $x_i \rightarrow \bar{x}_j$  и  $x_j \rightarrow \bar{x}_i$ . Утверждается, что формула будет выполнима тогда и только тогда, когда в построенном графе ни для одной переменной не будет одновременно путей из  $x_i$  в  $\bar{x}_i$  и из  $\bar{x}_i$  в  $x_i$ .

Действительно, с каждым набором связана раскраска графа в два цвета: истинный и ложный: если  $x_i = 1$ , то вершина  $x_i$  покрашена в истинный цвет,  $\bar{x}_i$  — в ложный, а если  $x_i = 0$ , то наоборот. Из построения графа следует, что для выполняющего набора если начало ребра покрашено в истинный цвет, то и конец тоже. Однако такого не может быть, если есть путь и из  $x_i$  в  $\bar{x}_i$  и из  $\bar{x}_i$  в  $x_i$ . Теперь покажем, что если для каждой вершины хотя бы одного пути нет, то выполняющий набор будет. Действительно, пусть, например, нет пути из  $x$  в  $\bar{x}$  (здесь и далее  $x$  и другие буквы обозначают литералы, мы считаем  $\bar{\bar{x}} = x$ ). Тогда не может быть и двух путей из  $x$  в  $y$  и из  $x$  в  $\bar{y}$ . Действительно, по построению графа если есть путь из  $x$  в  $y$ , то есть и путь из  $\bar{y}$  в  $\bar{x}$ . А если есть ещё и путь из  $x$  в  $\bar{y}$ , то будет и объединённый путь из  $x$  в  $\bar{x}$ , которого нет по предположению. Теперь сделаем истинными  $x$  и все достижимые из него литералы. По указанным выше соображениям это можно сделать непротиворечиво. Если в некотором дизъюнкте  $(z \vee t)$  один литерал (например,  $z$ ) стал ложен, значит  $\bar{z}$  достижим из  $x$ , а благодаря ребру  $\bar{z} \rightarrow t$  второй литерал также достижим и потому истинен. Значит, в каждом дизъюнкте либо будет хотя бы один истинный литерал, либо оба литерала не получают значений. Если второй случай будет иметь место, то продолжим присваивание значений, пока все переменные их не получают (если все дизъюнкты уже истинны, а неозначенные переменные остались, они могут быть любыми).

Таким образом, мы свели  $2SAT$  к проверке отсутствия некоторых  $n$  путей в графе (по одному из каждой пары  $x_i \rightarrow \bar{x}_i$  и  $\bar{x}_i \rightarrow x_i$ ). Поскольку  $\overline{PATN} \in NL$ , это можно сделать при помощи  $n$  сертификатов, отсортированных по возрастанию  $i$ .

Теперь докажем  $NL$ -полноту. Сведём  $\overline{PATN}$  к  $2SAT$ . По тройке  $(G, s, t)$  построим формулу так: для каждой вершины заведём переменную, а для каждого ребра  $u \rightarrow v$  заведём дизъюнкт  $(\bar{u} \vee v)$ . Кроме того, заведём отдельные дизъюнкты  $s$  и  $\bar{t}$ . Если пути из  $s$  в  $t$  нет, то рассмотрим набор, в котором все переменные, достижимые из  $s$ , истинны, а все недостижимые ложны. Любое ребро идёт либо между двумя истинными переменными, либо между двумя ложными, либо от ложной к истинной. В любом случае дизъюнкт  $(\bar{u} \vee v)$  истинен. Дизъюнкты  $s$  и  $\bar{t}$  также истинны. Таким образом, формула выполнима. Обратно, если формула выполнима, то любая переменная, достижимая из  $s$ , обязана быть истинна в выполняющем наборе. Это делается индукцией по длине пути: если вершина  $u$  достижима и есть ребро  $u \rightarrow v$ , то  $u$  истинна, и  $(\bar{u} \vee v)$  истинен, откуда  $v$  также должна быть истинна. Но  $\bar{t}$  также должно быть истинно, откуда  $t$  не может быть достижимо.  $\square$

## 5.5 Исторические замечания и рекомендации по литературе

## 5.6 Задачи и упражнения

**5.1.** Докажите, что для любого **PSPACE**-полного языка  $A$  выполнено  $\mathbf{P}^A = \mathbf{NP}^A$ .

# Глава 6

## Полиномиальная иерархия

### 6.1 Уровни полиномиальной иерархии

Одним из примеров **NP**-трудных задач является задача коммивояжёра: по графу  $G$  и весам рёбер найти цикл наименьшего веса, проходящий хотя бы однажды через каждую вершину. Это задача поиска, соответствующая задаче распознавания **NP**-трудна. Однако непонятно, лежит ли она в **NP**: легко проверить, что данная длина достигается, но почему нет ещё более короткого пути? Это пример задачи **NP**-оптимизации, которая в общем случае формулируется так: дана некоторая полиномиально вычислимая функция  $P(x, y)$  с натуральными значениями. Требуется для данного  $x$  найти  $y$ , для которого  $P(x, y)$  максимальна (или минимальна; в дальнейшем будем всё писать только для задач максимизации). Для всех таких задач язык  $\{(x, k) \mid \exists y P(x, y) \geq k\}$  лежит в **NP**, а вот язык  $\text{OPT}_P = \{(x, k) \mid \max_y P(x, y) = k\}$  не обязательно лежит. Также он скорее всего не лежит и в **coNP**. Причина состоит в том, что принадлежность языку из **NP** задаётся выражением вида  $\exists s V(x, s) = 1$ , принадлежность языку из **coNP** — выражением вида  $\forall s V(x, s) = 1$  (или  $\neg \exists s V(x, s) = 0$ ), а в данном случае требуется одновременно одновременно  $\exists y P(x, y) = k$  и  $\neg \exists y P(x, y) > k$ . Иными словами, язык является пересечением некоторого языка из **NP** и некоторого другого языка из **coNP**. Класс всех таких языков носит название **DP** (от difference polynomial — такие языки являются разностями двух языков из **NP**). Обратите внимание, что **DP** это не тоже самое, что  $\text{NP} \cap \text{coNP}$ : это пересечение *содержится* и в **NP**, и в **coNP**, а **DP** *содержит* и **NP**, и **coNP**. Если  $\text{NP} \neq \text{coNP}$ , то заведомо  $\text{DP} \neq \text{NP} \cap \text{coNP}$ . Формулу, задающую принадлежность к **DP**, можно привести к предварённой нормальной форме, например  $\exists y \forall z (P(x, y) = k \wedge P(x, z) \geq k)$ . Этот пример мотивирует рассмотрение классов языков, принадлежность к которым описывается формулой с несколькими чередующимися кванторами перед полиномиальным предикатом.

**Определение 6.1.** Классом  $\Sigma_k^P$  называется множество языков  $A$ , для которых существует полиномиально вычислимый предикат  $V$ , такой что

$$x \in A \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \forall y_k V(x, y_1, y_2, \dots, y_k) = 1$$

(последний квантор будет  $\forall$  при чётном  $k$  и  $\exists$  при нечётном  $k$ ). Аналогично классом  $\Pi_k^P$  называется множество языков  $A$ , для которых существует полиномиально вычислимый предикат  $V$ , такой что

$$x \in A \Leftrightarrow \forall y_1 \exists y_2 \forall y_3 \dots \forall y_k V(x, y_1, y_2, \dots, y_k) = 1$$

(последним квантором будет  $\exists$  при чётном  $k$  и  $\forall$  при нечётном  $k$ ). Классом **PH** называется объединение всех  $\Sigma_k^P$ .

Отметим, что некоторые тексты используют обозначения  $\Sigma_k \mathbf{P}$  и  $\Pi_k \mathbf{P}$ . Из определения сразу видно, что  $\Sigma_0^p = \Pi_0^p = \mathbf{P}$ ,  $\Sigma_1^p = \mathbf{NP}$ ,  $\Pi_1^p = \mathbf{coNP}$ . Также видна справедливость следующих утверждений:

**Утверждение 6.2.** Язык  $A$  лежит в  $\Sigma_k^p$  тогда и только тогда, когда  $\bar{A}$  лежит в  $\Pi_k^p$ .

*Доказательство.* Действительно,  $x \in \bar{A} \Leftrightarrow x \notin A \Leftrightarrow \neg \exists y_1 \forall y_2 \exists y_3 \dots \forall y_k V(x, y_1, y_2, \dots, y_k) = 1 \Leftrightarrow \forall y_1 \exists y_2 \forall y_3 \dots \forall y_k V(x, y_1, y_2, \dots, y_k) = 0$ . Заменяя предикат  $V$  на его отрицание, получаем формулу для  $\Pi_k^p$ , что и требовалось.  $\square$

**Утверждение 6.3.** Если  $A \in \Sigma_k^p$  и  $B \in \Sigma_k^p$ , то  $A \cup B$  и  $A \cap B$  также лежат в  $\Sigma_k^p$ . Аналогично для  $\Pi_k^p$ . Если же  $A \in \Sigma_k^p$ , а  $B \in \Pi_k^p$ , то  $A \cup B$  и  $A \cap B$  лежат в  $\Sigma_{k+1}^p \cap \Pi_{k+1}^p$ .

*Доказательство.* Все эти утверждения следуют из процедуры приведения формулы к предварённой нормальной форме. Например, пусть

$$x \in A \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \forall y_k V(x, y_1, y_2, \dots, y_k) = 1$$

и

$$x \in B \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \forall y_k W(x, y_1, y_2, \dots, y_k) = 1.$$

Тогда

$$\begin{aligned} x \in A \cap B \\ \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \forall y_k V(x, y_1, y_2, \dots, y_k) = 1 \wedge \exists z_1 \forall z_2 \exists z_3 \dots \forall z_k W(x, z_1, z_2, \dots, z_k) = 1 \\ \Leftrightarrow \exists y_1 \exists z_1 \forall y_2 \forall z_2 \exists y_3 \exists z_3 \dots \forall y_k \forall z_k (V(x, y_1, y_2, \dots, y_k) = 1 \wedge W(x, z_1, z_2, \dots, z_k) = 1) \\ \Leftrightarrow \exists (y_1, z_1) \forall (y_2, z_2) \exists (y_3, z_3) \dots \forall (y_k, z_k) (V(x, y_1, y_2, \dots, y_k) = 1 \wedge W(x, z_1, z_2, \dots, z_k) = 1). \end{aligned}$$

Последняя запись показывает принадлежность к  $\Sigma_k^p$ , поскольку стоящий после всех кванторов предикат вычислим за полиномиальное время, если использовать полиномиально вычисляемое кодирование пар.

Аналогично доказываются и остальные включения. Принадлежность к  $\Sigma_{k+1}^p \cap \Pi_{k+1}^p$  за счёт двух возможных порядков приведения к предварённой нормальной форме.  $\square$

Как следствие, получаем, что  $\mathbf{DP} \subset \Sigma_2^p \cap \Pi_2^p$ , а также  $\mathbf{PH} = \bigcup_{k=1}^{\infty} \Pi_k^p$ .

**Утверждение 6.4.**  $\Sigma_k^p \cup \Pi_k^p \subset \Sigma_{k+1}^p \cap \Pi_{k+1}^p$ .

*Доказательство.* Нужно доказать 4 отдельных вложения каждого из классов слева в каждый из классов справа. Каждое из них доказывается добавлением фиктивной переменной в предикат  $V$  и квантора по ней в начало или в конец цепочки. Например, для доказательства вложения  $\Sigma_k^p \subset \Pi_{k+1}^p$  нужно добавить квантор всеобщности по новой переменной  $y_0$  в начало.  $\square$

**Утверждение 6.5.**  $\mathbf{PH} \subset \mathbf{PSPACE}$ .

*Доказательство.* Можно предъявить рекурсивный алгоритм, который будет распознавать истинность соответствующей формулы и занимать полиномиальную память, а можно свести любую задачу из полиномиальной иерархии к TQBF: полиномиально вычисляемый предикат можно превратить в булеву формулу стандартной техникой, как в теореме Кука–Левина.  $\square$

Помимо задач оптимизации есть и другие естественные примеры задач с различных уровней иерархии (обычно второго или третьего), например:

- *Задача о минимальной ДНФ.* Рассматривается язык  $\text{MINEQDNF} = \{(\varphi, k) \mid \text{у формулы } \varphi \text{ есть эквивалентная ей ДНФ длины не более } k\}$ . Он принадлежит  $\Sigma_2^P$ , поскольку задаётся формулой  $\exists \psi \forall x (\varphi(x) = \psi(x) \wedge |\psi| \leq k)$ . Заметим, что соответствующая задача точной оптимизации будет лежать в  $\Sigma_3^P \cap \Pi_3^P$ .
- *Задача о рамсеевости частичной раскраски рёбер графа.* Известна теорема Рамсея: для любых  $k$  и  $l$  для всех достаточно больших  $n$  в любой раскраске рёбер полного графа на  $n$  вершинах в красный и синий цвета найдётся либо красная клика на  $k$  вершинах, либо синяя клика на  $l$  вершинах. Более общая задача ставится так: даны три графа,  $F$ ,  $G$  и  $H$ . Верно ли, что для любой раскраски рёбер  $F$  в красный и синий цвета найдётся либо красный подграф, изоморфный  $G$ , либо синий подграф, изоморфный  $H$ . Соответствующий язык будет лежать в  $\Pi_2^P$ , формула будет иметь вид  $\forall f: E(F) \rightarrow \{r, b\} \exists K ((K \simeq G \wedge f|_{E(K)} \equiv r) \vee (K \simeq H \wedge f|_{E(K)} \equiv b))$ , здесь  $E(F)$  обозначает множество рёбер графа  $F$ ,  $f$  — раскраска (отображение в множество из двух цветов),  $f|_{E(K)}$  — ограничение раскраски на подграф  $K$ .
- *Задача о размерности Вапника–Червоненкиса.* Размерностью Вапника–Червоненкиса  $VC(\mathcal{S})$  системы множеств  $\mathcal{S} = (S_1, \dots, S_k)$  называется максимальное количество элементов в множестве  $X$ , таком что среди множеств  $S_i \cap X$  встречаются все подмножества  $X$ . Предположим, что  $\mathcal{S}$  задана в сжатой форме, например, формулой  $\varphi$  полиномиальной длины, такой что для  $i \in \{1, \dots, k\}$  и  $y$  истинность  $\varphi(i, y)$  эквивалентна принадлежности  $y$  к  $S_i$ . Соответствующую систему обозначим через  $\mathcal{S}_\varphi$ . (Можно заменить формулу на схему из функциональных элементов или другое короткое эффективно вычислимое описание). Тогда язык  $\text{VCDIM} = \{(\varphi, m) \mid VC(\mathcal{S}_\varphi) \geq m\}$  лежит в  $\Sigma_3^P$ . Действительно, должно найтись такое  $X$  из  $k$  элементов, что для любого  $A \subset X$  найдётся  $i$ , такое что  $A = \{y \in X \mid \varphi(i, y) = 1\}$ . Если же стоит задача точного подсчёта размерности, то соответствующий язык будет лежать в  $\Sigma_4^P \cap \Pi_4^P$ .
- *Задача о кликовой раскраске.* Пусть дан граф  $G = (V, E)$ . Его кликовой раскраской в  $k$  цветов называется такая функция  $f: V \rightarrow \{1, \dots, k\}$ , что любая максимальная клика в графе содержит вершины хотя бы двух разных цветов. Язык  $\{(G, k) \mid \text{у графа } G \text{ есть кликовая раскраска в } k \text{ цветов}\}$  лежит в  $\Sigma_2^P$ : формула имеет вид  $\exists f \forall X \subset V (\text{если } X \text{ — максимальная клика, то } f(X) \text{ содержит хотя бы два элемента})$ . Действительно, удостовериться, что  $X$  клика можно, проверив, что все пары вершин из  $X$  образуют рёбра. Проверить максимальность тоже легко: любая другая вершина должна быть не соединена хотя бы с одной из вершин  $X$ . Отметим, что в отличие от обычной  $k$ -раскрашиваемости, кликовая  $k$ -раскрашиваемость может не сохраниться при переходе к подграфу. Если потребовать, чтобы все подграфы также допускали кликовую раскраску в  $k$  цветов, то соответствующий язык будет лежать в  $\Pi_3^P$ .

## 6.2 Отделимость классов иерархии друг от друга

### 6.2.1 Коллапсирование иерархии

Мы уже видели, что если  $\mathbf{P} = \mathbf{NP}$ , то  $\mathbf{NP} = \mathbf{coNP}$ . На самом деле можно показать, что в этом случае  $\mathbf{P} = \mathbf{PH}$ . Как говорят, в этом случае полиномиальная иерархия коллапсирует, или схлопывается. Более общее утверждение выглядит так:

**Теорема 6.6.** Если  $\Sigma_k^p = \Sigma_{k+1}^p$ , то  $\mathbf{PH} = \Sigma_k^p$ . Если  $\Sigma_k^p = \Pi_k^p$ , то также  $\mathbf{PH} = \Sigma_k^p$ .

В этом случае говорят, что полиномиальная иерархия схлопывается после  $k$ -го уровня.

*Доказательство.* Вначале докажем, что из  $\Sigma_k^p = \Pi_k^p$  следует  $\Sigma_k^p = \Sigma_{k+1}^p$ , и наоборот. Затем по индукции докажем, что  $\Sigma_k^p = \Sigma_{k+l}^p$ . Отсюда уже будет следовать, что  $\mathbf{PH} = \Sigma_k^p$ .

Действительно, пусть  $A \in \Sigma_{k+1}^p$ . Для определённости, пусть  $k$  нечётно. Значит,  $x \in A \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \forall y_{k+1} V(x, y_1, y_2, y_3, \dots, y_{k+1}) = 1$ . Множество  $\{(x, y_1) \mid \forall y_2 \exists y_3 \dots \forall y_{k+1} V(x, y_1, y_2, y_3, \dots, y_{k+1}) = 1\}$  лежит в  $\Pi_k^p$ . По предположению оно лежит и в  $\Sigma_k^p$ , т.е. принадлежность ему эквивалентна  $\exists z_2 \forall z_3 \dots \exists z_{k+1} W(x, y_1, z_2, z_3, \dots, z_{k+1}) = 1$ . Таким образом,  $x \in A \Leftrightarrow \exists y_1 \exists z_2 \forall z_3 \dots \exists z_{k+1} W(x, y_1, z_2, z_3, \dots, z_{k+1}) = 1$ . А это уже формула из  $\Sigma_k^p$ , поскольку кванторы по  $y_1$  и  $z_2$  можно объединить в один.

Обратно, пусть  $\Sigma_k^p = \Pi_k^p$ . Тогда  $\Pi_k^p \subset \Sigma_{k+1}^p$ , следовательно,  $\Pi_k^p \subset \Sigma_k^p$ . Отсюда верно и обратное включение: если  $B \in \Sigma_k^p$ , то  $\overline{B} \in \Pi_k^p$ , откуда  $\overline{B} \in \Sigma_k^p$ , откуда  $B \in \Pi_k^p$ . Значит,  $\Sigma_k^p = \Pi_k^p$ .

Далее рассуждение проходит похожим образом: если  $B \in \Pi_{k+1}^p$ , то  $\overline{B} \in \Sigma_{k+1}^p$ , откуда по доказанному  $\overline{B} \in \Sigma_k^p$ , откуда  $B \in \Pi_k^p$ . Отсюда  $\Sigma_{k+1}^p = \Pi_{k+1}^p = \Sigma_k^p$ , откуда по индукции  $\Sigma_{k+l}^p = \Pi_{k+l}^p = \Sigma_k^p$  и, как следствие,  $\mathbf{PH} = \Sigma_k^p$ .  $\square$

Возможность коллапсирования — черта, отличающая полиномиальную иерархию от арифметической, для которой доказаны строгие вложения на всех уровнях.

### 6.2.2 Полные задачи на уровнях иерархии

Как обычно, в задачах отделения одного класса от другого важную роль играют полные задачи в большем из классов. Такие задачи есть на каждом из уровней полиномиальной иерархии. Напомним определение.

**Определение 6.7.** Язык  $A$  называется  $\Sigma_k^p$ -полным ( $\Pi_k^p$ -полным), если он лежит в  $\Sigma_k^p$  (соотв.,  $\Pi_k^p$ ) и любой другой язык из  $\Sigma_k^p$  (соотв.,  $\Pi_k^p$ ) сводится к нему по Карпу.

Для всех уровней иерархии есть аналог теоремы Кука–Левина и теоремы о  $\mathbf{PSPACE}$ -полноте  $\mathbf{TQBF}$ .

**Теорема 6.8.** Задача  $\Sigma_k\text{SAT} = \{\varphi \mid \exists y_1 \forall y_2 \exists y_3 \dots \forall y_k \varphi(y_1, y_2, \dots, y_k) = 1\}$  (Здесь  $y_1, \dots, y_k$  обозначают группы булевых переменных) является  $\Sigma_k^p$ -полной. Аналогично задача  $\Pi_k\text{SAT} = \{\varphi \mid \forall y_1 \exists y_2 \forall y_3 \dots \exists y_k \varphi(y_1, y_2, \dots, y_k) = 1\}$  является  $\Pi_k^p$ -полной.



*Доказательство.* Во-первых, докажем эквивалентность утверждений о  $\Sigma_k^p$ -полноте задачи  $\Sigma_k\text{SAT}$  и  $\Pi_k^p$ -полноте задачи  $\Pi_k\text{SAT}$ . Идея состоит в том, что  $A \leq_p \Sigma_k\text{SAT} \Leftrightarrow \bar{A} \leq_p \Pi_k\text{SAT}$ . Пусть  $\varphi_x$  — формула, полученная по первой сводимости. Тогда  $x \in A \Leftrightarrow \exists \mathbf{y}_1 \forall \mathbf{y}_2 \exists \mathbf{y}_3 \dots \forall \mathbf{y}_k \varphi_x(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k) = 1$ . Тогда  $x \in \bar{A}$  эквивалентно  $\neg \exists \mathbf{y}_1 \forall \mathbf{y}_2 \exists \mathbf{y}_3 \dots \forall \mathbf{y}_k \varphi_x(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k) = 1$ , т.е.  $\forall \mathbf{y}_1 \exists \mathbf{y}_2 \forall \mathbf{y}_3 \dots \exists \mathbf{y}_k \varphi_x(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k) = 0$ . Таким образом, взятие  $\neg \varphi_x$  в качестве образа  $x$  задаст сводимость  $\bar{A}$  к  $\Pi_k\text{SAT}$ .

Во-вторых, докажем  $\Sigma_k^p$ -полноту задачи  $\Sigma_k\text{SAT}$  для нечётных  $k$  и  $\Pi_k^p$ -полноту задачи  $\Pi_k\text{SAT}$  для чётных  $k$  (чётность выбирается так, чтобы последний квантор был квантором  $\exists$ ). Пусть, например,  $A \in \Sigma_k^p$  для нечётного  $k$ . Тогда для некоторой полиномиальной машины  $M$  выполнено  $x \in A \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \exists y_k M(x, y_1, y_2, \dots, y_k) = 1$ . Построим по машине  $M$  и слову  $x$  формулу  $\varphi_{M,x}$  той же процедурой, что и в теореме Кука–Левина. Тогда биты слов  $y_1, \dots, y_k$  будут аргументами этой формулы, а истинность  $M(x, y_1, y_2, \dots, y_k) = 1$  будет соответствовать её выполнимости для этих зафиксированных значений. Таким образом, условие  $\exists y_1 \forall y_2 \exists y_3 \dots \exists y_k M(x, y_1, y_2, \dots, y_k) = 1$  эквивалентно  $\exists \mathbf{y}_1 \forall \mathbf{y}_2 \exists \mathbf{y}_3 \dots \exists \mathbf{y}_k \exists \mathbf{z} \varphi_{M,x}(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k, \mathbf{z}) = 1$ . Последние кванторы по  $\mathbf{y}_k$  и  $\mathbf{z}$  можно объединить, таким образом получим экземпляр задачи  $\Sigma_k\text{SAT}$ , что и требовалось.  $\square$

Может ли существовать полная задача во всём классе **РН**? Может, но только если иерархия коллапсирует.

**Утверждение 6.9.** *Если в классе **РН** существует полная задача относительно сводимости по Карпу, то  $\mathbf{РН} = \Sigma_k^p$  для некоторого  $k$ .*

*Доказательство.* Действительно, если язык  $A$  является **РН**-полным, то он, в частности, лежит в **РН** и потому лежит в  $\Sigma_k^p$  для некоторого  $k$ . Но раз любой другой  $B \in \mathbf{РН}$  сводится к  $A$ , то  $B$  также лежит в  $\Sigma_k^p$ , откуда  $\mathbf{РН} = \Sigma_k^p$ , что и требовалось.  $\square$

Среди «естественных» задач известно не так много задач, полных на том или ином уровне полиномиальной иерархии. В подавляющем большинстве случаев они полны на втором или (реже) третьем уровне. Большой список таких задач «в стиле Гэри–Джонсона» можно найти в работе [119].

## 6.3 Альтернирующие машины Тьюринга

Для класса **NP** помимо сертификатного определения имеется эквивалентное определение через недетерминированные машины Тьюринга. Другие классы иерархии также можно определить как классы языков, распознаваемых за полиномиальное время на машинах специального вида. Эти машины называются альтернирующими.

**Определение 6.10.** *Альтернирующей машиной Тьюринга называется машина Тьюринга с многозначной функцией перехода, у которой состояния делятся на два класса:  $\exists$ -состояния и  $\forall$ -состояния. Все конфигурации делятся на принимающие и отвергающие. Определение рекурсивное: во-первых, если конфигурация содержит состояние  $q_a$ , то она принимающая, а если  $q_r$ , то отвергающая. Во-вторых, если конфигурация содержит  $\exists$ -состояние, то она принимающая, если хотя бы одна из конфигураций, в которые можно перейти из неё, принимающая. В-третьих, если конфигурация содержит  $\forall$ -состояние, то она принимающая, если все конфигурации, в которые из неё можно перейти, принимающие.*

Естественным образом определяются классы языков, распознаваемых на альтернирующих машинах за ограниченное время.

**Определение 6.11.** Классом  $\mathbf{ATIME}(t(n))$  называется класс языков  $A$ , для которых существует альтернирующая машина  $M$ , такая что все её ветви имеют длину  $O(t(|x|))$  и  $x \in A$  тогда и только тогда, когда начальная конфигурация для входа  $x$  принимающая. Классами  $\Sigma_k \mathbf{TIME}(t(n))$  и  $\Pi_k \mathbf{TIME}(t(n))$  называются подклассы  $\mathbf{ATIME}(t(n))$ , в которых начальное состояние соответствующих машин является  $\exists$ -состоянием (соответственно,  $\forall$ -состоянием), а количество перемен вида состояний вдоль любой ветви вычислений не превышает  $k - 1$ .

Мы покажем, что альтернирующие машины позволяют охарактеризовать уровни полиномиальной иерархии и класс  $\mathbf{PSPACE}$ .

**Теорема 6.12.** *Класс  $\Sigma_k^p$  совпадает с  $\bigcup_{c=1}^{\infty} \Sigma_k \mathbf{TIME}(n^c)$ . Класс  $\Pi_k^p$  совпадает с  $\bigcup_{c=1}^{\infty} \Pi_k \mathbf{TIME}(n^c)$ . (Эти равенства мотивируют обозначения  $\Sigma_k \mathbf{P}$  и  $\Pi_k \mathbf{P}$ ). Класс  $\mathbf{PSPACE}$  совпадает с  $\mathbf{AP} = \bigcup_{c=1}^{\infty} \mathbf{ATIME}(n^c)$ .*

*Доказательство.* Пусть язык  $A$  распознаётся полиномиальной альтернирующей машиной  $T$ , в которой число перемен вида состояний не больше  $k - 1$ . Тогда он лежит в  $\Sigma_k^p$  (или  $\Pi_k^p$ , в зависимости от вида начального состояния): соответствующими  $y_1, \dots, y_k$  будут последовательности выбора следующей конфигурации в каждой ветви. Машина  $M$  на входе  $(x, y_1, \dots, y_k)$  будет симулировать действие машины  $T$  на входе  $x$  вдоль ветви, описанной  $y_1, \dots, y_k$ . В соответствии с определением альтернирующей машины вход  $x$  будет принят тогда и только тогда, когда верна формула с чередованием кванторов для  $M$ .

Обратно, пусть  $x \in A \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \forall y_k M(x, y_1, \dots, y_k) = 1$ . Тогда альтернирующая машина будет устроена так: сначала в  $\exists$ -состояниях пишет биты  $y_1$ , потом в  $\forall$ -состояниях пишет биты  $y_2$ , и так далее до  $y_k$ , а затем запускает детерминированное вычисление  $M(x, y_1, \dots, y_k)$ .

Осталось доказать, что  $\mathbf{AP} = \mathbf{PSPACE}$ . Вложение  $\mathbf{AP} \subset \mathbf{PSPACE}$  доказывается обычным образом: рекурсивный алгоритм будет работать на полиномиальной памяти. Обратное следует из теоремы о  $\mathbf{PSPACE}$ -полноте языка  $\mathbf{TQBF}$ : машина сначала вычислит формулу, которая получается в результате сводимости, а затем будет выбирать набор её аргументов, чередуя  $\exists$ - и  $\forall$ -состояния.  $\square$

Равенство  $\mathbf{AP} = \mathbf{PSPACE}$  позволяет интерпретировать любой язык из  $\mathbf{PSPACE}$  как множество выигрышных позиций в некоторой полиномиальной игре. По сути, игра идёт на конфигурационном графе альтернирующей машины. В  $\exists$ -состояниях ход делает первый игрок, а в  $\forall$ -состояниях — второй. Если игра пришла в принимающее состояние, то победил первый игрок, а если в отвергающее, то второй. Полиномиальность означает, что игра в любом случае продолжается полиномиальное число ходов, а после этого можно определить победителя полиномиальным по времени вычислением. В частности, в графе нет циклов. Тогда наличие выигрышной стратегии у первого игрока эквивалентно тому, что машина принимает исходное слово.

Аналогично любой язык из класса  $\Sigma_k^p$  ( $\Pi_k^p$ ) можно интерпретировать как множество выигрышных позиций для первого (второго) игрока в игре, длящейся не больше  $k$  ходов и имеющей полиномиальное условие выигрыша.

## 6.4 Определение иерархии при помощи оракулов

*Верь и дерзай, возвести нам оракул, какой бы он ни был!*

Гомер, *Илиада*

Ещё одним способом описания классов полиномиальной иерархии является использование вычислений с оракулом. Напомним, что если  $\mathbf{C}$  — некоторый класс языков, а  $A$  — некоторый язык, то  $\mathbf{C}^A$  — аналог класса  $\mathbf{C}$ , в котором вычисления могут использовать  $A$  в качестве оракула, иначе говоря, могут устанавливать принадлежность к  $A$  за один шаг. Аналогично, если  $\mathbf{D}$  — тоже некоторый класс языков, то  $\mathbf{C}^{\mathbf{D}}$  — аналог класса  $\mathbf{C}$ , в котором в качестве оракула можно использовать любой язык из  $\mathbf{D}$ . Если в  $\mathbf{D}$  есть полный язык  $B$  в смысле сводимости, которую можно реализовать в  $\mathbf{C}$ , то  $\mathbf{C}^{\mathbf{D}} = \mathbf{C}^B$ . Действительно, запрос к любому другому языку из  $\mathbf{D}$  можно реализовать через сводимость и запрос к  $B$ . Теперь всё готово к новой характеристизации полиномиальной иерархии.

**Теорема 6.13.** *Для всех  $k$  выполнено, что*

- $\Sigma_k^p = \mathbf{NP}^{\Sigma_{k-1}^p} = \mathbf{NP}^{\Pi_{k-1}^p}$ ;
- $\Pi_k^p = \mathbf{coNP}^{\Sigma_{k-1}^p} = \mathbf{coNP}^{\Pi_{k-1}^p}$ .

*Доказательство.* Второе равенство в каждом пункте доказывается легко: если  $A \in \Sigma_{k-1}^p$ , то  $\bar{A} \in \Pi_{k-1}^p$ , а если можно устанавливать принадлежность к  $A$ , то можно устанавливать и принадлежность к  $\bar{A}$ , и наоборот. Поэтому  $\mathbf{C}^A = \mathbf{C}^{\bar{A}}$ , откуда  $\mathbf{C}^{\Sigma_{k-1}^p} = \mathbf{C}^{\Pi_{k-1}^p}$ .

Покажем, что  $\Sigma_k^p \subset \mathbf{NP}^{\Pi_{k-1}^p}$ . Для определённости, пусть  $k$  нечётно. Пусть  $B \in \Sigma_k^p$ . Тогда для некоторой полиномиальной машины  $M$  выполнено  $x \in B \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \dots \exists y_k M(x, y_1, y_2, y_3, \dots, y_k) = 1$ . Заметим, что множество  $D = \{(x, y_1) \mid \forall y_2 \exists y_3 \dots \exists y_k M(x, y_1, y_2, y_3, \dots, y_k) = 1\}$  лежит в  $\Pi_{k-1}^p$ , при этом  $B = \{x \mid \exists y_1 (x, y_1) \in D\}$ . Поэтому  $B \in \mathbf{NP}^{\Pi_{k-1}^p}$ , что и требовалось. Аналогично  $\Pi_k^p \subset \mathbf{coNP}^{\Sigma_{k-1}^p}$ .

Вложение в обратную сторону доказывается сложнее, поскольку запросы к оракулу могут делаться многократно и, более того, адаптивно (т.е. новые запросы могут зависеть от ответов на старые). Для того чтобы основная идея была более ясна, изложим вначале, почему  $\mathbf{NP}^{\mathbf{NP}} \subset \Sigma_2^p$ . Пусть  $A \in \mathbf{NP}^{\mathbf{NP}}$ , а  $M$  — соответствующая недетерминированная машина. Как мы уже отмечали, можно считать, что все запросы делаются только к одному  $\mathbf{NP}$ -полному языку  $B$ , распознаваемому верификатором  $V$ . Если сделан запрос  $y$  и получен положительный ответ, это означает, что  $\exists z V(y, z) = 1$ , а если отрицательный, то  $\forall z V(y, z) = 0$ . Тогда принадлежность  $x$  к  $A$  задаётся такой формулой: существуют  $s$  (сертификат для  $M$ ) и  $y_1, \dots, y_m$  (последовательность запросов к оракулу), такие что существуют  $z_{i_1}, \dots, z_{i_p}$ , при которых выполнено  $V(y_{i_1}, z_{i_1}) = 1$ , а для любых  $z_j$ , где  $j \notin \{i_1, \dots, i_p\}$  выполнено  $V(y_j, z_j) = 0$ , при этом выбор  $y_1, \dots, y_m$  согласован с работой  $M(x, s)$  и ответами оракула  $y_j \in B$  для  $j \in \{i_1, \dots, i_p\}$  и  $y_j \notin B$  для остальных  $j$ , а в итоге  $M(x, s) = 1$ . Часть формулы, идущая после кванторов, проверяется за полиномиальное время, что доказывает  $A \in \Sigma_2^p$ .

Теперь разберёмся, почему  $\mathbf{coNP}^{\mathbf{NP}} \subset \Pi_2^p$ . Если повторить изложенное рассуждение, то получится доказать только принадлежность  $\Pi_4^p$ : первый квантор по  $s$  сменится на  $\forall$ . Чтобы доказать принадлежность  $\Pi_2^p$ , нужно заменить кванторы по  $y_1, \dots, y_m$  с существования на всеобщность и немного видоизменить

последующую формулу: если последовательность запросов  $y_1, \dots, y_m$  согласована с работой  $M(x, s)$  и предыдущими ответами оракула, то  $M(x, s) = 1$ .

В общем случае конструкция похожа: отличие состоит в том, что вместо одного  $z_j$  имеется цепочка из  $k - 1$  слова, по которым стоят чередующиеся кванторы. Если выносить кванторы от всех слов параллельно, начиная с самых внешних, то получим формулу вида  $\Sigma_k^p$  или  $\Pi_k^p$ , что и требовалось.  $\square$

Возникает вопрос: а что будет, если в качестве «основания» взять не **NP** или **coNP**, а просто **P**? Оказывается, возникает ещё одна цепочка классов, тоже относящаяся к полиномиальной иерархии.

**Определение 6.14.** Классом  $\Delta_k^p$  называется  $\mathbf{P}^{\Sigma_{k-1}^p}$ .

Как и прежде, он же будет равен  $\mathbf{P}^{\Pi_{k-1}^p}$ . Поскольку  $\mathbf{P} \subset \mathbf{NP} \cap \mathbf{coNP}$ , и это вложение релятивизируется, сразу получаем  $\Delta_k^p \subset \Sigma_k^p \cap \Pi_k^p$ . С другой стороны, поскольку  $A$  и  $\bar{A}$  всегда лежат в  $\mathbf{P}^A$ , получаем  $\Sigma_{k-1}^p \cap \Pi_{k-1}^p \subset \Delta_k^p$ .

Ясно, что  $\Delta_1^p = \mathbf{P}^{\mathbf{P}} = \mathbf{P}$ . А вот место  $\Delta_2^p$  уже не так понятно. Можно лишь заметить, что он включает в себя **DP**. Дело в том, что все классы  $\Delta_k^p$  замкнуты относительно пересечения, объединения и разности. В частности, если  $A \in \mathbf{NP} \subset \Delta_2^p$  и  $B \in \mathbf{coNP} \subset \Delta_2^p$ , то  $A \cap B$  также лежит в  $\Delta_2^p$ .

## 6.5 Другие иерархии: обзор

Понятие той или иной иерархии множеств возникает в различных разделах логики, теории алгоритмов и теории множеств. Например, известно понятие арифметической иерархии, которая получается аналогично полиномиальной, только начинается не с полиномиально разрешимых множеств, а просто с разрешимых. Другой известный пример — иерархия борелевских множеств. Здесь мы приведём краткий обзор иерархий, возникающих в теории сложности вычислений.

**Определение 6.15.** Классом **ВН** (булевой иерархией) называется минимальный класс языков, который включает в себя **NP**, но при этом замкнут относительно пересечения, объединения и дополнения. Конкретными уровнями булевой иерархией называются такие классы:

- $\mathbf{ВН}_1 = \mathbf{NP}$
- $\mathbf{ВН}_{2k}$  — класс всех языков, являющихся пересечением некоторого языка из  $\mathbf{ВН}_{2k-1}$  и некоторого языка из **coNP**.
- $\mathbf{ВН}_{2k+1}$  — класс всех языков, являющихся объединением некоторого языка из  $\mathbf{ВН}_{2k}$  и некоторого языка из **NP**.

Ясно, что  $\mathbf{ВН}_2 = \mathbf{DP}$  (по определению), а также что  $\mathbf{ВН} \subset \Delta_2^p$  (поскольку  $\Delta_2^p$  замкнуто относительно пересечения, объединения и дополнения). Известен результат, что если булева иерархия схлопывается на каком-либо уровне, то  $\mathbf{PH} = \Sigma_3^p$ . Таким образом, булева иерархия уточняет полиномиальную на участке между **NP** и  $\Delta_2^p$ . Интересно, что булеву иерархию можно охарактеризовать иначе, двинувшись не со стороны **NP**, а со стороны  $\Delta_2^p$ : булева иерархия содержит языки, для распознавания которых нужна константное число запросов к оракулу на полиномиальной машине. И здесь есть своя иерархия по точному количеству этих запросов.

Вместо полиномиального ограничения на время можно взять какое-нибудь другое. Таким образом можно получить полилогарифмическую, линейно-экспоненциальную, экспоненциальную, дважды экспоненциальную и другие иерархии. В общем виде удобно использовать определение через альтернирующие машины:

**Определение 6.16.** Полилогарифмической, линейно-экспоненциальной, экспоненциальной, дважды экспоненциальной иерархиями называются следующие классы:

- $\text{PLH} = \bigcup_{k=1}^{\infty} \bigcup_{c=1}^{\infty} \Sigma_k \text{TIME}(2^{\log^c n})$ ;
- $\text{EH} = \bigcup_{k=1}^{\infty} \bigcup_{c=1}^{\infty} \Sigma_k \text{TIME}(2^{cn})$ ;
- $\text{EXPH} = \bigcup_{k=1}^{\infty} \bigcup_{c=1}^{\infty} \Sigma_k \text{TIME}(2^{n^c})$ ;
- $\text{EEN} = \bigcup_{k=1}^{\infty} \bigcup_{c=1}^{\infty} \Sigma_k \text{TIME}(2^{2^{cn}})$ ;

Отдельными уровнями иерархий будут  $\Sigma_k \text{TIME}$  и  $\Pi_k \text{TIME}$  для соответствующих временных ограничений.

Наконец, некоторые иерархии мы изучим подробно на следующих лекциях. Это **НС**-иерархия для языков, распознаваемых схемами константной глубины, (раздел 7.4) и иерархия задач подсчёта, которая похожа на полиномиальную, но вместо кванторов  $\exists$  и  $\forall$  используются кванторы «для большинства» (раздел 9.7).

## 6.6 Исторические замечания и рекомендации по литературе

### 6.7 Задачи и упражнения

**6.1.** . Докажите, что если существует  $B \in \text{PH}$ , такой что  $\text{P}^B \neq \text{NP}^B$ , то  $\text{P} \neq \text{NP}$ .

**6.2.** . Докажите, что если какой-нибудь язык из  $\text{NP} \cap \text{coNP}$  является  $\text{NP}$ -полным, то  $\text{PH} = \text{NP}$ .



# Глава 7

## Схемная сложность

Схемы из функциональных элементов (boolean circuits) — вычислительная модель, существенно отличная от машин Тьюринга, и потому стоящая обособленно в линии нашего повествования. Основное отличие состоит в том, что заранее фиксируется длина входа, поэтому в некотором смысле все функции становятся вычислимыми. Исторически сложность схем начали изучать ещё до появления структурной теории сложности, а впоследствии между сложностью схем и сложностью алгоритмов установлено множество связей в обе стороны. Именно поэтому изучение схем крайне важно для понимания основной теории.

### 7.1 Определения

Неформально говоря, схема из функциональных элементов — это модель настоящей микросхемы, которая получает на вход какие-то сигналы, перерабатывает их и посылает результат на выход. Микросхема спаяна из некоторого количества базовых элементов при помощи проводов, передающих сигналы. Каждый элемент рассматривается как «чёрный ящик», вычисляющий некоторую заданную функцию. Подавляющее большинство современных микросхем построено на двоичной логике: сигнал либо есть, либо отсутствует. Такой логики будем придерживаться и мы.<sup>1</sup>

С формальной точки зрения схемы можно рассматривать как обобщение булевых формул, в которых каждую подформулу можно использовать многократно. Это может быть записано в виде ориентированного графа, либо в виде «прямолинейной программы» (straight-line program): простейшей инструкции без циклов и условных переходов, состоящей лишь из списка присваиваний значений новым переменным, в котором каждая переменная зависит только от ранее встретившихся.

#### 7.1.1 Схемы как графы

**Определение 7.1.** *Схемой из функциональных элементов* называется ориентированный граф без циклов, в котором каждая вершина помечена одной из пяти меток: in, out,  $\neg$ ,  $\vee$ ,  $\wedge$ . При этом соблюдаются следующие условия на входящие и исходящие степени:

---

<sup>1</sup>В 1950-70-х годах в СССР и США пытались построить ЭВМ на троичной логике, однако широкого распространения они не получили. Дональд Кнут предполагает [155, раздел 4.1], что в будущем компьютеры могут перейти на троичную логику в силу её симметрии и красоты. Возможный путь реализации такого перехода — фотонные вычисления. В любом случае наша теория от числа логических значений почти не зависит.

Тип метки	Входящая степень	Исходящая степень
in	0	любая
out	1	0
$\neg$	1	любая
$\wedge$	2 (вар. любая)	любая
$\vee$	2 (вар. любая)	любая

Вершины с пометкой in мы будем называть *входными*, а вершины с пометкой out — *выходными*.

Два варианта допустимой входящей степени для элементов  $\wedge$  и  $\vee$  впоследствии будут использованы для определения разных сложностных классов.

**Определение 7.2.** *Высотой* вершины будем называть длину самого длинного пути от некоторой входной вершины до данной. Поскольку в графе нет циклов, этот максимум корректно определён.

**Определение 7.3.** Пусть всем входным вершинам сопоставлены некоторые булевы значения. Тогда значения всех остальных вершин можно определить индукцией по высоте: значения в вершинах с метками  $\neg$ ,  $\vee$  и  $\wedge$  будут вычисляться как соответствующие функции от значений предшествующих вершин, а значения в вершинах с меткой out будет получено переносом значения в предшествующей вершине. Таким образом, схема с  $n$  входными и  $k$  выходными вершинами задаёт функцию из  $\{0, 1\}^n$  в  $\{0, 1\}^k$ .

Если у схемы только одна выходная вершина, то она вычисляет характеристическую функцию некоторого подмножества  $\{0, 1\}^n$  и, таким образом, его распознаёт. Если рассмотреть семейство схем с  $n$  входами для каждого  $n$ , то можно говорить о распознавании языка  $A \subset \{0, 1\}^*$ .<sup>2</sup> Это позволяет определять различные сложностные классы языков, однако сначала нужно определить размеры ресурсов.

**Определение 7.4.** *Размером* схемы называется количество вершин в графе. *Глубиной* схемы называется максимальная высота вершины.

**Определение 7.5.** Классом  $\mathbf{SIZE}(s(n))$  называется множество языков  $A$ , для которых существует семейство схем  $\{C_n\}_{n=1}^\infty$  размера  $O(s(n))$ , такое что  $x \in A$  тогда и только тогда, когда  $C_{|x|}(x) = 1$ . Классом  $\mathbf{DEPTH}(d(n))$  называется множество языков, распознающихся семейством схем глубины  $O(d(n))$ , а классом  $\mathbf{SIZEDEPTH}(s(n), d(n))$  — множество языков, распознающихся семейством схем одновременно размера  $O(s(n))$  и глубины  $O(d(n))$ .

Определения классов  $\mathbf{DEPTH}(\cdot)$  и  $\mathbf{SIZEDEPTH}(\cdot)$  зависят от того, какая входящая степень допускается у элементов  $\wedge$  и  $\vee$ . Однако различие невелико: элемент со входящей степенью  $k$  можно смоделировать двоичным деревом глубины  $\lceil \log k \rceil$  из элементов со входящей степенью 2.

Поскольку на распознающее семейство не накладывается никаких ограничений, а любая булева функция выражается через  $\neg$ ,  $\wedge$  и  $\vee$ , каким-то семейством схем можно распознать абсолютно любой язык.

<sup>2</sup>Данное нами определение схемы не подходит для случая 0 входов. С одной стороны, этот случай можно не рассматривать: речь идёт всего лишь об одном (пустом) слове. С другой стороны, можно рассмотреть дополнительные элементы — логические константы с нулевой входящей степенью. Если такие элементы есть, то у схемы может и не быть входов.



**Теорема 7.6.** *Для любого языка существует распознающее его семейство схем размера  $O(n2^n)$  и глубиной 4, в которых элементы  $\wedge$  и  $\vee$  имеют произвольную входящую степень.*

*Доказательство.* Для каждой фиксированной длины входа можно представить характеристическую функцию распознаваемого языка в виде КНФ (или в виде ДНФ). Соответствующая формула есть конъюнкция не более чем  $2^n$  дизъюнктов, каждый из которых объединяет не больше  $n$  литералов. Общее количество связей как раз будет иметь порядок  $O(n2^n)$ . А самый длинный путь от входной вершины до выходной будет состоять из 5 вершин и 4 рёбер:  $\text{in} \rightarrow \neg \rightarrow \vee \rightarrow \wedge \rightarrow \text{out}$ .  $\square$

Эта теорема показывает, что рассматривать классы  $\mathbf{DEPTH}(\cdot)$  для произвольной входящей степени большого смысла не имеет. А вот для фиксированной входящей степени конструкция с КНФ/ДНФ даёт только линейную глубину.

Известны оценки Шеннона [123]: почти любая функция требует схемы размера  $\frac{2^n}{n}(1 - o(1))$ , но при этом любую функцию можно представить схемой размера  $O\left(\frac{2^n}{n}\right)$  — и уточняющая верхнюю оценку теорема Лупанова [163]: любую функцию можно представить схемой размера не больше  $\frac{2^n}{n}(1 + o(1))$  и глубиной  $O(n)$ . Нижняя оценка доказывается элементарными комбинаторными методами, фактически принципом Дирихле. Для получения верхних оценок требуется довольно хитрое представление любой функции.

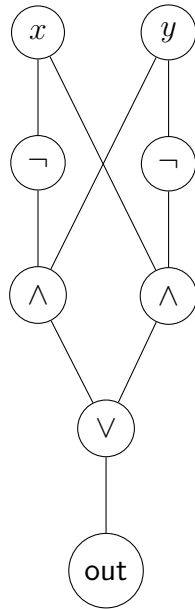
Выбор функций отрицания, конъюнкции и дизъюнкции в качестве базовых достаточно произволен. Ничто не мешает использовать другие элементы, например  $\oplus$ , функцию большинства или пороговые функции. Однако использование другого конечного базиса изменит сложность и глубину схемы для любой функции не более чем в константу раз: через новые базисные элементы можно фиксированным образом выразить старые, и наоборот. При замене базисных элементов на эти выражения размер и глубина всей схемы изменятся не более чем в константу раз. При использовании другого бесконечного базиса изменения могут быть более значительными, как уже произошло с элементами  $\wedge$  и  $\vee$  произвольной входящей степени.

### 7.1.2 Схемы как программы

**Определение 7.7.** *Прямолинейной программой называется последовательность, состоящая из списка исходных переменных  $x_1, \dots, x_n$  и команд-присваиваний вида  $z := \neg y$ ,  $z := y_1 \wedge y_2$ ,  $z := y_1 \vee y_2$ , где в качестве  $y$ ,  $y_1$ ,  $y_2$  могут выступать либо исходные переменные, либо переменные, ранее встретившиеся в левой части присваивания. Некоторые переменные помечены как выходные. Если всем исходным переменным присвоены некоторые булевы значения, то и всем остальным переменным поочерёдно присваиваются значения в соответствии с функциями, указанными в присваивании.*

Как и в случае графического представления схем можно рассмотреть присваивания с функциями  $\wedge$  и  $\vee$ , зависящими от произвольного числа аргументов, или рассмотреть другие базисные функции. Схемы и программы естественным образом соответствуют друг другу.

**Утверждение 7.8.** *Для каждой схемы из функциональных элементов существует прямолинейная программа, вычисляющая ту же функцию, и наоборот. Количество входных вершин схемы равно количеству исходных перемен-*



(а) Граф

**Вход:** Булевы переменные  $x$   
и  $y$

**Результат:**  $x \oplus y$

1  $z_1 := \neg x;$

2  $z_2 := \neg y;$

3  $z_3 := z_1 \wedge y;$

4  $z_4 := x \wedge z_2;$

5  $z_5 := z_3 \vee z_4;$

6 **возвратить**  $z_5$

(б) Программа

Рис. 7.1: Представление схемы для функции чётности в виде графа и в виде программы

ных программы, количество выходных вершин равно количеству выходных переменных, а количество остальных вершин равно числу присваиваний в программе.

*Доказательство.* Соответствие строится естественным образом, проиллюстрированным на рис. 7.1. Если вначале дана схема, то каждой вершине сопоставляется некоторая переменная. Далее вершины перебираются в порядке неубывания глубины и для каждой пишется присваивание, в котором функция соответствует метке, а переменные — предыдущим вершинам. Если из вершины  $z$  идёт ребро в выходную вершину, то переменная  $z$  назначается выходной.

Если же вначале дана программа, то каждому присваиванию сопоставляется вершина, помеченная той же функцией, которая использована в присваивании. В неё направляются рёбра из вершин, соответствующих переменным из правой части. Если же переменная выходная, то от её вершины проводится ребро в ещё одну выходную вершину.  $\square$

## 7.2 Класс $P_{\text{poly}}$ и машины с подсказками

Как обычно, самым главным ограничением для нас является полиномиальное. А именно, изучается такой класс:

**Определение 7.9.**  $P_{\text{poly}} = \bigcup_{c=1}^{\infty} \text{SIZE}(n^c)$ .

Сразу заметим, что верно такое вложение:

**Утверждение 7.10.**  $P \subset P_{\text{poly}}$ .

*Доказательство.* Доказательство в целом повторяет доказательство теоремы Кука–Левина. А именно, если есть машина Тьюринга, работающая за  $p(n)$  шагов, то строится схема размера  $O(p(n)^2)$ , моделирующая работу этой машины. Схема будет вычислять символы, стоящие в протоколе работы этой машины.

Поскольку каждый символ зависит только от 4 символов на предыдущем этапе, эту зависимость можно выразить булевой функцией фиксированного размера, которая записывается в виде схемы. Таким образом, можно вычислить элементы всех ячеек протокола, а затем проверить, что в последней строке есть принимающее состояние.  $\square$

Обратное включение неверно.

**Утверждение 7.11.**  $\mathbf{P} \neq \mathbf{P}/_{\text{poly}}$ .

*Доказательство.* Например, в  $\mathbf{P}/_{\text{poly}}$  будет лежать любой унарный язык  $U$ , т.е. язык, содержащий только слова вида  $1^k$ . Действительно, для каждой длины  $U \cap \{0, 1\}^k$  будет либо пустым множеством, либо состоять только из  $1^k$ . В первом случае язык распознаётся тождественно ложной схемой, во втором — конъюнкцией.

А унарные языки могут быть даже неразрешимыми, например  $\text{UNHALT} = \{1^k \mid \text{машина Тьюринга } M_k \text{ останавливается на входе } k\}$ . Также они могут быть разрешимыми, но не лежать в  $\mathbf{P}$ . Например, можно взять любой язык  $A \in \mathbf{EEXP} \setminus \mathbf{EXP}$  (такой есть по теореме об иерархии) и рассмотреть  $\{1^k \mid k \in A\}$ . Если его можно распознать за полиномиальное время, то  $A$  можно распознать за экспоненциальное время, что противоречит выбору  $A$ .  $\square$

Пример с унарными языками показывает, в чём сила вычислений при помощи схем: для разных длин можно брать разные схемы, при этом схема «сваливается с неба», а не вычисляется по аргументу. Именно поэтому семейства схем могут распознавать даже неразрешимые языки. Концепция «подсказки с неба» позволяет дать эквивалентное определение класса  $\mathbf{P}/_{\text{poly}}$ , которое заодно объясняет выбор обозначения.

**Определение 7.12.** Классом  $\mathbf{DTIME}(t(n))/_{a(n)}$  называется множество языков  $A$ , для которых существует машина Тьюринга  $M$ , и последовательность «подсказок» (advice), т.е. слов  $\alpha_n$  длины  $O(a(n))$ , такие что  $M(x, \alpha_{|x|}) = 1 \Leftrightarrow x \in A$ , и при этом  $M$  работает не больше  $O(t(n))$  шагов.

**Утверждение 7.13.**  $\mathbf{P}/_{\text{poly}} = \bigcup_{c,d=1}^{\infty} \mathbf{DTIME}(n^c)/_{n^d}$ .

Иными словами,  $\mathbf{P}/_{\text{poly}}$  состоит из языков, которые распознаются за полиномиальное время с полиномиальной подсказкой.

*Доказательство.* Как обычно, равенство двух множеств доказывается через два включения. Пусть вначале  $A \in \mathbf{P}/_{\text{poly}}$ . Построим машину с подсказкой, которая будет распознавать  $A$ . В качестве подсказки будем использовать саму схему  $C_{|x|}$ . По определению она будет полиномиального размера от  $|x|$ , а машина будет вычислять значение этой схемы на  $x$ . (Это действие полиномиально и даже линейно от размера схемы, т.е. полиномиально от длины  $x$ ).

В обратную сторону рассуждение проводится так: сначала нужно переделать машину  $M$  в схему обычным образом. У этой схемы будет две группы входов: для битов  $x$  и для битов  $\alpha$ . Поскольку  $\alpha$  было полиномиальной длины от  $n$ , то и вся схема будет полиномиального размера. Окончательная схема  $C_n$  получается фиксацией  $\alpha_n$  в качестве второго аргумента. Так можно сделать, поскольку  $\alpha_n$  одно и то же для всех  $x$  длины  $n$ .  $\square$

Легко заметить, что любой унарный язык лежит не просто в  $\mathbf{P}/_{\text{poly}}$ , а в  $\mathbf{DTIME}(n)/_1$ . Видно также, что если бы подсказка была быстро вычислима, то она была бы не нужна.

**Утверждение 7.14.** *Если язык  $A$  распознаётся семейством схем  $C_n$ , таким что некоторым полиномиальным алгоритмом можно по  $1^n$  построить  $C_n$ , то  $A \in \mathbf{P}$ .*

*Доказательство.* Во-первых, поскольку схемы строятся полиномиальным алгоритмом, они будут полиномиального размера, поэтому  $A \in \mathbf{P}/_{\text{poly}}$ . Во-вторых, взяв композицию алгоритма, строящего схему, и алгоритма, вычисляющего значение схемы, получим алгоритм, распознающий язык. Значит, язык лежит в  $\mathbf{P}$ .  $\square$

Семейства схем, упомянутые в теореме, также называют *равномерными* (uniform). Поэтому языки из  $\mathbf{P}/_{\text{poly}}$  иногда называют *неравномерно вычислимыми* за полиномиальное время (nonuniformly polynomially computable).

## 7.3 Соотношения $\mathbf{P}/_{\text{poly}}$ с другими классами

Мы выяснили, что  $\mathbf{P}/_{\text{poly}}$  строго шире  $\mathbf{P}$ . Но насколько шире? Разумеется, точного ответа на этот вопрос нет, но есть некоторое количество условных теорем. В этом разделе мы познакомимся с двумя из них.

### 7.3.1 Теорема Карпа–Липтона

Если бы удалось построить семейство схем полиномиального размера для  $\mathbf{NP}$ -полной задачи, то это существенно упростило бы решение таких задач на практике. Для каждого конкретного размера задачи было бы достаточно найти одну конкретную схему, а потом её эксплуатировать. Следующая теорема показывает, что это невозможно без коллапсирования полиномиальной иерархии.

**Теорема 7.15** (Карпа–Липтона, [81]). *Если  $\mathbf{NP} \subset \mathbf{P}/_{\text{poly}}$ , то  $\mathbf{PH} = \Sigma_2^p$ .*

*Доказательство.* Как известно, для схлопывания полиномиальной иерархии достаточно доказать, что  $\Pi_2^p = \Sigma_2^p$ . А для этого достаточно доказать, что задача  $\Pi_2\text{SAT}$  лежит в  $\Sigma_2^p$ . Этим мы и займёмся.

Напомним, что язык  $\Pi_2\text{SAT}$  состоит из таких формул  $\varphi$  от двух групп переменных, что для любых значений  $x$  найдутся значения  $y$ , такие что  $\varphi(x, y) = 1$ . Заметим, что язык  $\{(\varphi, x) \mid \exists y \varphi(x, y) = 1\}$  лежит в  $\mathbf{NP}$  и, по предположению, в  $\mathbf{P}/_{\text{poly}}$ . Таким образом, есть полиномиальное семейство схем  $D_n$ , которые получают на вход пару  $(\varphi, x)$  и возвращают  $y$ , такое что  $\varphi(x, y) = 1$ . Далее стандартным образом переделаем алгоритм распознавания в алгоритм поиска и получим другое полиномиальное семейство схем  $S_n$  с несколькими выходами, которые получают на вход пару  $(\varphi, x)$  и возвращают такой  $y$ , что  $\varphi(x, y) = 1$ .<sup>3</sup> Теперь принадлежность  $\varphi$  к  $\Pi_2\text{SAT}$  можно задать такой  $\Sigma_2$ -формулой:  $\exists S_n \forall x \varphi(x, S_n(\varphi, x)) = 1$ . Действительно, если  $\varphi \in \Pi_2\text{SAT}$ , то подойдёт построенная схема  $S_n$ . Более того, она по построению даже не будет зависеть от  $\varphi$ , хотя из формулы это не следует. Если же верна формула, то даже если  $S_n$  зависит от  $\varphi$ , всё равно  $S_n(\varphi, x)$  будет таким  $y$ , что  $\varphi(x, y) = 1$ . Значит, формула  $\forall x \exists y \varphi(x, y)$  истинна, т.е.  $\varphi \in \Pi_2\text{SAT}$ . Итак, принадлежность к  $\Pi_2\text{SAT}$  задана  $\Sigma_2$ -формулой, откуда  $\Pi_2^p = \Sigma_2^p$  и  $\mathbf{PH} = \Sigma_2^p$ .  $\square$

<sup>3</sup>Напомним, как это делается. Для простоты будем считать, что аргументы  $\varphi$  заранее не разделены на  $x$  и  $y$ , а схема  $C_n$  считает переменными  $x$  те, значения которых получила на вход. Тогда первый бит  $y$  определяется так: если  $C_n(\varphi, x1) = 1$ , то он равен единице, иначе нулю. То есть он попросту равен  $y_1 = C_n(\varphi, x1)$ . Второй бит будет равен  $C_n(\varphi, xy_11)$ , и т.д. Ясно, что результирующая схема будет полиномиального размера.

### 7.3.2 Теорема Мейера

В следующей теореме, по сравнению с предыдущей, более сильное условие и более сильное заключение. Удивительно, что ни первое, ни второе не противоречат современным знаниям.

**Теорема 7.16** ([81]).<sup>4</sup> Если  $\mathbf{EXP} \subset \mathbf{P}/_{\text{poly}}$ , то  $\mathbf{EXP} = \Sigma_2^p$ .

*Доказательство.* Поскольку полиномиальная иерархия вложена в  $\mathbf{PSPACE}$  и тем более в  $\mathbf{EXP}$ , достаточно доказать  $\mathbf{EXP} \subset \Sigma_2^p$ . Рассмотрим некоторый язык  $A \in \mathbf{EXP}$ . Пусть он распознаётся машиной  $M$  за время  $e(n) = 2^{en^c}$ . Рассмотрим протокол работы машины  $M$  на входе  $x$  — таблицу размером  $e(n) \times e(n)$ . За экспоненциальное время можно вычислить, какой символ лежит в фиксированной ячейке этой таблицы. По предположению это же можно сделать схемой  $P_n$  полиномиального размера, получающей на вход координаты  $(i, j)$ . Напомним, что для корректности протокола достаточно проверить корректность всех прямоугольников  $2 \times 3$ . Пусть это делается схемой  $C$  (константного размера). Если к тому же протокол начинается с правильной начальной конфигурации для слова  $x$  и заканчивается принимающим состоянием, то машина принимает  $x$ . Таким образом, принадлежность  $x$  к  $A$  эквивалентна следующему факту:

$$\begin{aligned} \exists P_n(\forall i \forall j C(P_n(i, j), P_n(i+1, j), \dots, P_n(i+2, j+1)) \wedge \\ P_n(0, 0) = q_0 \wedge \forall i(i \leq n \Rightarrow P_n(i, 0) = x_i \wedge (i > n \Rightarrow P_n(i, 0) = \#) \wedge \\ \exists i P_n(i, e(n)) = q_a). \end{aligned}$$

Здесь первая строчка означает корректность протокола, вторая строчка — корректность первой его строки, третья — корректность последней. После приведения к предварённой нормальной форме получается  $\Sigma_2$ -формула. Остаётся заметить, что поскольку размер таблицы экспоненциальный, все переменные занимают полиномиальное число битов, что и требуется для принадлежности к  $\Sigma_2^p$ .  $\square$

В частности, если  $\mathbf{EXP} \subset \mathbf{P}/_{\text{poly}}$ , то  $\mathbf{P} \neq \mathbf{NP}$ , т.к. иначе  $\Sigma_2^p = \mathbf{P}$  и в силу доказанной теоремы  $\mathbf{EXP} = \mathbf{P}$ , что противоречит теореме об иерархии. Это один из примеров, когда из верхней оценки (на схемную сложность любого языка из  $\mathbf{EXP}$ ) вытекает нижняя оценка (на сложность решения SAT на детерминированной машине Тьюринга).

## 7.4 NC-иерархия

### 7.4.1 Определения классов $\mathbf{NC}^d$ и $\mathbf{AC}^d$

До сих пор мы говорили только о классах, связанных с размером схемы. Однако глубина — не менее важная её характеристика, поскольку именно она отвечает за быстроедействие. Действительно, если на прохождение сигнала по проводу и, главное, вычисление значения в элементе схемы затрачивается некоторой время (один тик), но вычисления могут идти параллельно, то значение схемы глубины  $d$  будет вычислено за  $d$  тиков. Для расчёта глубины важна точная модель, а именно входная степень элементов  $\wedge$  и  $\vee$ . Поэтому вводится два семейства классов:

<sup>4</sup>Теорема впервые появилась в статье Карпа и Липтона, но была атрибутирована Мейеру.

**Определение 7.17.** Классом  $\mathbf{NC}^d$  называется класс языков, которые распознаются семейством схем полиномиального размера и глубины  $O(\log^d n)$ , в которых элементы  $\wedge$  и  $\vee$  имеют входную степень ровно 2. Классом  $\mathbf{NC}$  называется объединение  $\mathbf{NC}^d$  для всех натуральных  $d$ .

Класс  $\mathbf{NC}$  — пример класса, названного в честь конкретного человека — Николаса Пишпенджера ( $\mathbf{NC}$  — Nick's Class).<sup>5</sup>

**Определение 7.18.** Классом  $\mathbf{AC}^d$  называется класс языков, которые распознаются семейством схем полиномиального размера и глубины  $O(\log^d n)$ , в которых элементы  $\wedge$  и  $\vee$  имеют любую входную степень.

Разные авторы накладывают разные ограничения на *равномерность* соответствующих семейств схем. Есть три основных варианта: никаких ограничений кроме размера и глубины; полиномиальная равномерность (схема для размера  $n$  строится за полиномиальное время) и логарифмическая равномерность (схема для размера  $n$  строится на логарифмической памяти). Также ограничения могут зависеть от глубины: например, схему для  $\mathbf{NC}^d$  должна строиться на памяти  $O(\log^d n)$ . Мы по умолчанию не будем налагать никаких условий, но иногда будем задавать спецификацию явно.

Нетрудно доказать такое соотношение между введёнными классами:

**Утверждение 7.19.** Для всех  $d$  выполнено  $\mathbf{NC}^d \subset \mathbf{AC}^d \subset \mathbf{NC}^{d+1}$ .

*Доказательство.* Действительно, схемы для  $\mathbf{NC}^d$  — это частный случай схем для  $\mathbf{AC}^d$ , отсюда берётся первое включение. Если же элемент  $\wedge$  или  $\vee$  имеет входную степень  $k$ , то можно его заменить двоичным деревом глубины  $\lceil \log k \rceil$  из элементов входной степени 2. Поскольку размер всей схемы был полиномиальным, такой же будет и максимальная входная степень каждого элемента, откуда  $\lceil \log k \rceil = O(\log n)$ . Таким образом, замена на двоичные деревья увеличит общую глубину не больше, чем в  $O(\log n)$  раз, т.е. из  $O(\log^d n)$  она превратится в  $O(\log^{d+1} n)$ . Таким образом,  $\mathbf{AC}^d \subset \mathbf{NC}^{d+1}$ , как и было сказано.  $\square$

Отсюда получается, что  $\bigcup_{d=1}^{\infty} \mathbf{AC}^d = \bigcup_{d=1}^{\infty} \mathbf{NC}^{d+1} = \mathbf{NC}$ , поэтому отдельного обозначения  $\mathbf{AC}$  не вводят. Касательно строгости вложений известно немного.

**Утверждение 7.20.**  $\mathbf{NC}^0 \neq \mathbf{AC}^0$ .

*Доказательство.* В классе  $\mathbf{NC}^0$  лежат языки, которые вычисляются схемами некоторой константной глубины  $c$ . Поскольку у каждого элемента не больше двух входов, число изначальных входов, от которых зависит значение, будет не больше  $2^c$ . Поэтому такой схемой нельзя вычислить функцию, которая зависит от всех входов, например, их конъюнкцию. В то же время конъюнкция тривиальным образом лежит в  $\mathbf{AC}^0$ , откуда  $\mathbf{NC}^0 \neq \mathbf{AC}^0$ .  $\square$

**Теорема 7.21.**  $\mathbf{AC}^0 \neq \mathbf{NC}^1$ .

*Идея доказательства.* Примером языка, лежащего в  $\mathbf{NC}^1$ , но не в  $\mathbf{AC}^0$ , является язык  $\mathbf{PARITY} = \{(x_1, \dots, x_n) : \bigoplus_{i=1}^n x_i = 1\}$ . Принадлежность к  $\mathbf{NC}^1$  доказывается просто: нужно зафиксировать схему для  $\oplus$  и собрать из таких схем двоичное дерево. Непринадлежность к  $\mathbf{AC}^0$  доказывается алгебраической техникой. Идея состоит в том, что любой язык из  $\mathbf{AC}^0$  можно хорошо приблизить некоторым многочленом, а  $\mathbf{PARITY}$  — нельзя.  $\square$

<sup>5</sup>Другим примером является «Стивов класс»  $\mathbf{SC}$ , названный в честь Стивена Кука. Это класс языков, распознаваемых за полиномиальное время и на полилогарифмической памяти.



О строгости вложений на более высоких уровнях NC-иерархии пока что ничего неизвестно.

### 7.4.2 Семейства схем для сложения и умножения

Классы NC-иерархии легко можно расширить с задач распознавания на задачи вычисления функций: отличие только в количестве выходов у схемы. Многие стандартные функции лежат в NC-иерархии. Мы посмотрим на простейшие арифметические операции, начнём со сложения. Везде будем использовать двоичную запись.

**Теорема 7.22.** *Функцию сложения можно вычислить схемой размера  $O(n \log n)$  и глубины  $O(\log n)$ .*

Стандартный школьный алгоритм сложения «в столбик» тут не подойдёт, поскольку в нём последовательно вычисляются биты переноса с младших разрядов к старшим. Поэтому глубина получится порядка  $n$ . Идея состоит в том, чтобы некоторым образом получить биты переноса заранее и потом сложить их с битами чисел параллельно в каждом разряде. Подобные схемы называются сумматорами с ускоренным переносом и широко используются в вычислительных машинах, начиная с механических арифмометров. Известно много различных реализаций идеи ускоренного переноса, мы изложим одну из них.

*Доказательство теоремы 7.22.* Пусть даны два числа в двоичной записи:  $a = \overline{a_1 \dots a_n}$  и  $b = \overline{b_1 \dots b_n}$ , для простоты будем считать, что  $n$  — степень двойки. Цель состоит в том, чтобы для каждого разряда найти бит переноса  $c_i$ , чтобы затем получить в каждом разряде результат  $s_i = a_i \oplus b_i \oplus c_i$  (таким образом,  $c_i = 1$ , если есть перенос из  $(i+1)$ -го разряда в  $i$ -ый, при этом нумерация начинается с нуля, поскольку в сумме на один разряд больше). Но в процессе нужно найти промежуточные биты *генерации* и *проталкивания*. Начнём с определений.

Рассмотрим группу подряд идущих разрядов  $i \dots j$ . Будем говорить, что  $g_{i \dots j} = 1$  (блок  $i \dots j$  генерирует перенос), если при сложении  $\overline{a_i \dots a_j}$  и  $\overline{b_i \dots b_j}$  образуется перенос. Будем говорить, что  $p_{i \dots j} = 1$  (блок  $i \dots j$  проталкивает перенос), если при сложении  $\overline{a_i \dots a_j}$  и  $\overline{b_i \dots b_j}$  образуется число из одних единиц: блок сам по себе не образует переноса, но если придёт перенос из младших разрядов, то он «протолкнётся» в более старшие. Далее мы последовательно вычислим биты генерации и переноса для всех «выровненных» блоков с длиной — степенями двойки, т.е. блоков вида  $l2^m + 1 \dots (l+1)2^m$ .

В качестве базы мы возьмём блоки из одного бита, для них биты определяются так:  $g_{i \dots i} = a_i \wedge b_i$  (перенос появляется, если оба слагаемых — единицы), а  $p_{i \dots i} = a_i \oplus b_i$  (по определению). Далее переход осуществляется следующим образом: разобьём блок  $i \dots j$  на два подблока  $i \dots k$  и  $k+1 \dots j$  и посчитаем для него биты генерации и проталкивания, используя биты для подблоков. В большом блоке генерируется перенос, либо если он генерируется в старшем подблоке, либо если генерируется в младшем и проталкивается старшим:  $g_{i \dots j} = g_{i \dots k} \vee (g_{k+1 \dots j} \wedge p_{i \dots k})$ . Большой блок проталкивает бит, если его проталкивают оба подблока:  $p_{i \dots j} = p_{i \dots k} \wedge p_{k+1 \dots j}$ . Поскольку каждый блок добавляет константу новых элементов, а всего рассматривается  $O(n)$  блоков, общий размер построенной схемы равен также  $O(n)$ . Глубина же будет логарифмической, поскольку каждый блок использует результаты двух блоков вдвое короче.

Теперь посчитаем биты переноса. Для каждой позиции  $i$  разобьём весь «хвост»  $i \dots n$  на блоки, для которых посчитаны биты генерации и переноса. Если действовать «жадным» алгоритмом, начиная с конца, то для каждой длины будет

взято не больше одного блока, поэтому общее число блоков не превысит  $\log n$ . Обозначим эти блоки (т.е. диапазоны индексов) цифрами  $1, \dots, q$ . Тогда бит переноса можно найти по формуле  $c_i = g_1 \vee (g_2 \wedge p_1) \vee (g_3 \wedge p_2 \wedge p_1) \vee \dots \vee (g_q \wedge p_{q-1} \wedge \dots \wedge p_1)$ . Действительно, перенос либо возник в первом блоке, либо возник во втором и протолкнулся через первый, либо возник в третьем и протолкнулся через первые два, и так далее. Поскольку  $q = O(\log n)$ , размер схемы для вычисления одного бита переноса будет также  $O(\log n)$ , а для вычисления всех битов переноса —  $O(n \log n)$ .<sup>6</sup> Глубина же добавится константная (если разрешить только элементы с двумя входами, то повторный логарифм).

Таким образом, схема состоит из трёх этапов: вычисление битов генерации и проталкивания для блоков, затем вычисление битов переноса для разрядов и, наконец, вычисление результатов. Общий размер составляет  $O(n \log n)$ , а глубина —  $O(\log n)$ , как и было заявлено. Таким образом, сложение лежит в  $\mathbf{NC}^1$ .  $\square$

Теперь перейдём к операции умножения. Стандартный алгоритм умножения  $n$ -битовых чисел  $a$  и  $b$  «в столбик» требует сложения не более чем  $n$  двоичных чисел, каждое из которых есть либо сдвиг  $b$  на  $j$  битов, если  $a_j = 1$ . Если эти числа сложить при помощи двоичного дерева, получится схема размера  $O(n^2)$  и глубины  $O(\log^2 n)$ : дерево имеет глубину  $O(\log n)$ , и глубина на каждом уровне тоже  $O(\log n)$ . Это даёт принадлежность умножения только к  $\mathbf{NC}^2$ . Покажем, что оно также лежит в  $\mathbf{NC}^1$ .

**Теорема 7.23.** *Функцию умножения можно вычислить схемой размера  $O(n^2)$  и глубины  $O(\log n)$ .*

*Доказательство.* Мы построим схему константной глубины, которая на вход получает три числа  $x, y, z$  и возвращает два числа  $u, v$ , такие что  $u + v = x + y + z$ . После применения этой схемы  $\log \frac{2}{3}n = O(\log n)$  раз останется 2 числа, которые сложим обычным образом. Таким образом, общая глубина составит те же  $O(\log n)$ .

Осталось построить саму схему. Идея очень проста: сложим  $x, y$  и  $z$  без переносов, например в десятичной системе. В каждом разряде будут числа 0, 1, 2 или 3. Далее представим эти числа в двухбитной двоичной записи: 00, 01, 10, 11. Младшие разряды всех этих чисел образуют число  $u$ , а старшие (со сдвигом на 1 влево) — число  $v$ . Глубина будет константной, ведь применяется просто одна и та же схема. Корректность конструкции почти очевидна: в число  $u$  собраны все результаты сложения без переносов, а в число  $v$  — все биты переносов.  $\square$

### 7.4.3 Связь с классами $\mathbf{L}$ и $\mathbf{NL}$

Мы доказали, что сложение и умножение лежат в  $\mathbf{NC}^1$ . Также известно, что эти функции можно вычислить на логарифмической памяти. Оказывается, можно доказать общую теорему:

**Теорема 7.24.** *Языки из  $\mathbf{NC}^1$ , которые распознаются логарифмически равномерно схемой, лежат в  $\mathbf{L}$ .*

*Доказательство.* Пусть  $A \in \mathbf{NC}^1$ . Логарифмическая равномерность означает, что есть программа, работающая на логарифмической памяти, которая по  $1^n$

<sup>6</sup>Более аккуратным анализом, повторно используя промежуточные данные, можно получить размер  $O(n)$ .



строит схему  $C_n$  полиномиального размера и логарифмической глубины, распознающую  $A^n$ . Требуется показать, что значение полученной схемы на конкретном входе также можно вычислить на логарифмической памяти, после этого теорема будет следовать из теоремы о логарифмической вычислимости композиции двух логарифмически вычислимых функций.

Если задана и схема логарифмической глубины, и её вход, то результат можно вычислить рекурсивно, используя логарифмическую память. Действительно, начнём с последнего элемента. У него либо один аргумент (если это  $\neg$ ), либо два (если это  $\wedge$  или  $\vee$ , ведь схема из  $\mathbf{NC}^1$ ). Нужно вычислить значение первого аргумента, затем на той же памяти — второго (если он есть), а затем вычислить функцию. Вычисление делается рекурсивно, вплоть до входных битов. На хранение промежуточных результатов на каждом уровне нужна константная память, поэтому общая память будет такого же порядка, как и глубина схемы, то есть логарифмической.  $\square$

Похожим способом можно доказать и более общее утверждение:

**Теорема 7.25.** *Если язык  $A$  лежит в  $\mathbf{DEPTH}(d(n))$ , где  $d(n) \geq \log n$ , в схеме используются только элементы со входящей степенью 2, а саму схему можно построить на памяти  $O(d(n))$ , то  $A \in \mathbf{DSPACE}(d(n))$ .*

*Идея доказательства.* Все этапы доказательства предыдущей теоремы, а именно лемма о композиции и рекурсивный алгоритм вычисления значения схемы, работают не только для логарифмического ограничения, но и для любого большего.  $\square$

Теперь перейдём к верхней оценке на глубину схемы.

**Теорема 7.26.**  $\mathbf{NL} \subset \mathbf{AC}^1$ .

Как следствие,  $\mathbf{L}$  тоже включено в  $\mathbf{AC}^1$ .

*Доказательство.* Достаточно доказать, что  $\mathbf{PATH} \in \mathbf{AC}^1$ , а логарифмическая сводимость не выводит за пределы этого класса.

Начнём с того, почему  $\mathbf{PATH} \in \mathbf{AC}^1$ . Пусть ориентированный граф с  $N$  вершинами задан матрицей смежности  $A$ . Будем считать, что на диагонали стоят единицы, т.е.  $A_{ij} = 1$ , если из  $i$  в  $j$  есть путь длины не больше 1. Рассмотрим  $\vee$ -умножение булевых матриц, отличающееся от обычного умножения тем, что вместо сложения используется дизъюнкция:  $(A \cdot B)_{ij} = \bigvee_{k=1}^N (A_{ik} \wedge B_{kj})$ .<sup>7</sup> Тогда  $(A^m)_{ij} = 1$ , если из  $i$  в  $j$  есть путь длины не больше  $m$ . (Возведение в степень, разумеется, понимается как итерированное  $\vee$ -умножение). Соответственно,  $(A^N)_{ij} = 1$ , если есть хоть какой-то путь из  $i$  в  $j$ : если путь есть, то есть и путь не длиннее  $N$ . Таким образом, для решения задачи о наличии пути нужно возвести булеву матрицу в  $N$ -ую  $\vee$ -степень. Это нужно сделать быстрым возведением: посчитать  $A^2$ , затем  $A^4$ ,  $A^8$  и т.д., всего  $\log N$  возведений в квадрат. Каждый этап вычисляется схемой константной глубины (с элементами произвольной входящей степени) по определению. Таким образом, общая глубина схемы составит  $O(\log N)$ .

Теперь докажем, что логарифмическая сводимость не выводит из класса  $\mathbf{AC}^1$ . На самом деле достаточно доказать это не для произвольной сводимости, а для конструкции, которая используется для сведения к  $\mathbf{PATH}$ . Иными

<sup>7</sup>В схемотехнике порой дизъюнкцию и обозначают через  $+$ , тогда запись будет стандартной, поменяется только смысл.

словами, нужно установить, соответствует ли переход от одной конфигурации к другой командам некоторой недетерминированной машины Тьюринга. Это делается уже известным из теоремы Кука–Левина методом: нужно, чтобы во всех малых окрестностях переход был допустимым. Такая формула будет конъюнкцией формул фиксированного размера, т.е. лежать в  $\mathbf{AC}^0$ . Таким образом, мы получили даже  $\mathbf{AC}^0$ -сводимость, которая по доказанному ранее будет логарифмической.

Осталось заметить, что размер полученного графа будет полиномиальным от размера исходного входа:  $N = \text{poly}(n)$ . Таким образом, глубина итоговой схемы составляет  $O(\log N) = O(\log n)$ , что и требовалось.  $\square$

Таким образом, мы доказали  $\mathbf{L} \subset \mathbf{NL} \subset \mathbf{AC}^1$ , и теперь можем доказать общее утверждение о том, что логарифмическая сводимость сохраняет класс  $\mathbf{AC}^1$ . Действительно, при логарифмической сводимости каждый бит результата вычисляется на логарифмической памяти и, как следствие,  $\mathbf{AC}^1$ -схемой. Значит, и все биты будут вычислены  $\mathbf{AC}^1$ -схемой.

## 7.5 Обзор других классов

Если рассмотреть другой набор базовых элементов, то множество языков, распознаваемых схемой той или иной глубины, изменится.

## 7.6 Исторические замечания и рекомендации по литературе

## 7.7 Задачи и упражнения

**7.1.** Докажите, что  $\mathbf{EXPSPACE} \not\subset \mathbf{P}/_{\text{poly}}$ .

## Глава 8

# Вероятностная сложность

*Учёные подсчитали, что шансы реального  
существования столь откровенно абсурдного мира  
равны одному на миллион.  
Однако волшебники подсчитали, что шанс «Один на  
миллион» выпадает в девяти случаев из десяти.*

Терри Пратчетт, *Мор, ученик Смерти*

Вероятностные вычисления широко используются на практике. Пожалуй, самый известный алгоритм — метод Монте-Карло вычисления объёма многомерной фигуры: выберем случайно несколько точек внутри параллелепипеда, в который фигура заведомо заключена, и посчитаем, какая доля из них попала внутрь фигуры. Часто понятие «алгоритм Монте-Карло» толкуют расширительно: так называют любой вероятностный алгоритм, который работает заданное время и даёт ответ с некоторой ошибкой. Также рассматривают «алгоритмы Лас-Вегаса», т.е. вероятностные алгоритмы, которые всегда дают правильный ответ, но работают быстро только в среднем, и «алгоритмы Атлантик-Сити», которые и могут ошибаться, и не всегда работают быстро.

Если речь идёт о задаче распознавания, то вероятностный алгоритм может с некоторой вероятностью ошибиться. Цель разработки алгоритмов состоит в том, чтобы сделать вероятность ошибки маленькой, при этом чтобы ответ был получен достаточно быстро и с помощью небольшого числа случайных битов. В этой главе мы изучим возникающие сложностные классы.

### 8.1 Что такое вероятностные вычисления

В обычной жизни, как и в математике за пределами сложности вычислений, слова «недетерминированный» и «вероятностный» часто используют как синонимы. Когда речь идёт о вычислениях, эти слова приобретают существенно разный смысл. Точнее, недетерминированная машина и вероятностная машина — одинаковые в комбинаторном смысле объекты, но вот вычисления на них ведутся по-разному. Основная идея заключается в том, что недетерминированная машина «перебирает параллельно» все возможные варианты, а вероятностная — выбирает один из них случайно. Как и раньше, одну и ту же идею можно формализовать разными способами.

**Определение 8.1.** Под вероятностной машиной Тьюринга мы будем понимать детерминированную машину Тьюринга  $M$  с двумя аргументами  $x$  (собственно аргумент) и  $r$  (случайные биты), где длина  $r$  есть некоторая функция от длины

$x$ . Результатом работы  $M(x, r)$  будет вероятностное распределение, индуцированное конкретным выбором  $x$  и равномерным распределением на всех значениях  $r$ . Временем работы машины  $M$  на данном  $x$  мы будем считать максимальное время работы  $M(x, r)$  для всех  $r$  указанной длины. Так же определяется и используемая память.

Удобно иметь в виду следующие эквивалентные вариации определения:

- У машины есть специальная (бесконечная) лента только для чтения, полностью заполненная независимыми равномерно распределёнными случайными битами. Время работы считается как функция от длины основного входа  $x$ . Это определение эквивалентно исходному, потому что машина всё равно прочтёт только конечную область случайных битов.
- У машины есть две функции перехода, каждый раз машина выбирает случайную из них равномерно и независимо от предыдущих выборов. Это определение эквивалентно исходному, поскольку машина может сначала сгенерировать все случайные биты, а потом работать как раньше (здесь две функции перехода начнут совпадать).
- Наконец, у машины может быть специальное состояние, после прихода в которое машина получает новый случайный бит (например, равновероятно и независимо от предыстории перехода в одно из ещё двух специальных состояний).

## 8.2 Примеры вероятностных алгоритмов

Вероятностные алгоритмы часто используются на практике. В этом разделе мы приводим два классических примера.

### 8.2.1 Проверка двух многочленов на равенство

Если два многочлена заданы в виде списка одночленов, то проверить их на равенство легко: достаточно проверить, что для каждого одночлена в одном из них есть равный ему одночлен в другом. (Имеется в виду, что можно переставлять порядок одночленов и переменных в них). Ситуация усложняется, если многочлены зависят от многих переменных и заданы арифметическими выражениями или, шире, арифметическими схемами. Такие схемы выглядят так же, как и булевы, но вместо логических операций стоят арифметические. Раскрытие всех скобок может занять слишком долгое время: например, произведение  $n$  скобок по 2 слагаемых после раскрытия скобок в худшем случае имеет экспоненциальную длину. Вместо этого используется следующая идея: посчитаем значения многочленов в случайной точке. Если значения разные, то многочлены тоже заведомо разные. Если же значения одинаковые, то с высокой вероятностью и многочлены одинаковые. Ниже приводится точный алгоритм и оценка вероятности его ошибки.

**Теорема 8.2.** Алгоритм 1 работает правильно.

*Доказательство.* Если  $p = q$ , то для любого  $x$  выполнено и  $p(x) = q(x)$ , тем более  $p(x) \equiv q(x) \pmod{m}$ , поэтому алгоритм выдаст 1. Если же  $p \neq q$ , то нужно доказать, что доля таких  $x$ , что  $p(x) \equiv q(x) \pmod{m}$ , не превышает  $\varepsilon$ . Мы

**Вход:** Две арифметические схемы  $P$  и  $Q$  от  $n$  переменных, задающие многочлены  $p$  и  $q$  степени не выше  $d$

**Результат:** 1, если  $p = q$ ;  
0 с вероятностью  $\geq 1 - \varepsilon$ , если  $p \neq q$

- 1  $N := \frac{nd}{\varepsilon}$ ;
- 2  $m$  — простое число больше  $N$ ; // Можно выбрать детерминированным алгоритмом или вероятностным из следующего раздела
- 3  $x$  — случайный элемент  $(\mathbb{Z}_m)^n$ ;
- 4 если  $p(x) \equiv q(x) \pmod{m}$  то вернуть 1;
- 5 иначе вернуть 0;

**Алгоритм 1:** Вероятностный алгоритм проверки двух многочленов на совпадение

докажем, что доля таких  $x$ , что  $p(x) \not\equiv q(x) \pmod{m}$ , не меньше  $(1 - \frac{d}{m})^n$ , что больше  $1 - \varepsilon$  в силу выбора параметров и неравенства Бернулли.

Доказывать будем индукцией по  $n$ . При  $n = 1$  утверждение следует из основной теоремы алгебры: у ненулевого многочлена  $p - q$  степени не выше  $d$  над полем  $\mathbb{Z}_m$  не больше  $d$  корней. Поэтому доля таких  $x$ , что  $p(x) \equiv q(x) \pmod{m}$ , не превышает  $\frac{d}{m}$ . Далее, пусть для некоторого  $n$  утверждение уже доказано. Докажем для  $n + 1$ . Пусть  $x = (x_0, y) \in (\mathbb{Z}^m)^{n+1}$ . Разложим многочлены  $p$  и  $q$  по степеням  $x_0$ . Хотя бы при одной степени коэффициенты будут различными, поэтому по предположению индукции с вероятностью не меньше  $(1 - \frac{d}{m})^n$  они останутся разными при подстановке случайного  $y$ . Таким образом, после подстановки  $p(x_0, y)$  и  $q(x_0, y)$  останутся разными многочленами от  $x_0$  и с условной вероятностью не меньше  $1 - \frac{d}{m}$  при подстановке случайного  $x_0$  дадут разные значения. Таким образом, полная вероятность будет не меньше  $(1 - \frac{d}{m})^{n+1}$ , что и требовалось.  $\square$

## 8.2.2 Проверка числа на простоту

*Если бы результат [Агравала–Каяла–Саксены] был известен в 1970-х, изучение вероятностных алгоритмов, возможно, не развивалось бы так быстро.*

Ланс Фортноу и Стив Гомер, *Краткая история сложности вычислений*, [48]

Проверка числа на простоту — известнейшая алгоритмическая задача, для которой разработано множество различных методов. Высшим теоретическим достижением стало построение детерминированного полиномиального AKS-алгоритма [8], однако на практике по-прежнему широко используются вероятностные тесты (для которых степень полинома ниже). Разумеется, полином считается не от самого числа, простота которого проверяется, а от длины его записи, т.е. от логарифма. Наиболее популярными алгоритмами являются тест Миллера–Рабина [97, 108] и тест Соловея–Штрассена [129]. Мы изложим основные идеи обоих алгоритмов, не вдаваясь в теоретико-числовые детали.

Прежде всего, напомним малую теорему Ферма: если  $p$  простое, а  $a$  не делится на  $p$ , то  $a^{p-1}$  сравнимо с 1 по модулю  $p$ . На этом факте построен тест Ферма: взять случайное  $a$  и проверить сравнение  $a^{p-1} \equiv 1$ . К сожалению, этот тест проходят и некоторые составные числа, а именно числа Кармайкла, причём таких чисел бесконечно много.

Тест Соловея–Штрассена использует ещё одно свойство простых чисел, связанное с квадратичными вычетами, а именно критерий Эйлера:  $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}}$ . Напомним, что  $\left(\frac{a}{p}\right)$  — это символ Лежандра, т.е. 1, если  $a$  — квадратичный вычет по модулю  $p$ ;  $-1$ , если  $a$  — квадратичный невычет по модулю  $p$ ; и 0, если  $a$  и  $p$  не взаимно просты. Если  $p$  не простое, та же величина называется символом Якоби, и критерий Эйлера выполнен не более, чем для половины остатков. При этом символ Якоби можно быстро посчитать, используя мультипликативность и квадратичный закон взаимности. Таким образом, тест Соловея–Штрассена заключается в том, чтобы для случайного  $a$  проверить равенство  $\left(\frac{a}{m}\right) = a^{\frac{m-1}{2}}$ . Если  $m$  простое, то тест точно будет пройден, а если  $m$  нечётное составное, то тест будет пройден с вероятностью не больше  $\frac{1}{2}$  (с чётными и так всё ясно). Многократное повторение теста уменьшит вероятность ошибки (подробнее об этом ниже).

Тест Миллера–Рабина проверяет ещё одно свойство, выполненное для всех простых чисел  $p > 2$ : если  $p - 1 = 2^k r$ , где  $r$  нечётно, то для любого  $a$ , взаимно простого с  $p$ , либо  $a^r \equiv 1 \pmod{p}$ , либо для некоторого  $l \in [0, k - 1]$  выполнено  $a^{2^l r} \equiv -1 \pmod{p}$ . Это утверждение следует из малой теоремы Ферма и того факта, что в поле  $\mathbb{Z}_p$  есть только два квадратных корня из единицы: 1 и  $-1$ : последовательно извлекая корни, мы либо получим на каком-то этапе  $-1$ , либо до последнего этапа останется 1. Рабин показал, что для если  $p$  нечётное составное, то это утверждение выполнено не для всех  $a$ , а лишь не более, чем для четверти. Алгоритм Миллера–Рабина состоит в проверке этого утверждения для случайно выбранного  $a$ . Простое число проходит его всегда, а нечётное составное — с вероятностью не больше 25%. Многократное повторение вновь позволяет уменьшить ошибку.

### 8.3 Вероятностные сложностные классы и связи между ними

Этот раздел — основной для понимания структуры вероятностных сложностных классов и их связей между собой и с другими классами.

#### 8.3.1 Классы BPP, RP, coRP и ZPP

Ошибка при использовании вероятностного алгоритма распознавания языка  $A$  может быть двух родов: первого, когда  $x \notin A$ , но алгоритм выдаёт единицу, и второго, когда  $x \in A$ , но алгоритм выдаёт ноль. Ошибки первого рода также называют  $\alpha$ -ошибками или ложноположительным срабатыванием (false positive), а вероятность такой ошибки — параметром точности или просто точностью (precision или soundness). Ошибки второго рода, соответственно, —  $\beta$ -ошибками или ложноотрицательным срабатыванием (false negative), а вероятность того, что ошибки нет — параметром полноты или просто полнотой (recall или completeness).<sup>1</sup> Основным критерием, разделяющим классы, является спектр возникающих ошибок: обоих родов, одного из них или никакого.

<sup>1</sup>Логика этих названий такая: точность говорит, насколько точно можно верить сработавшей системе, а полнота — насколько полно она выявляет все случаи. В английском языке термины precision и recall употребляются в информационном поиске, а soundness и completeness пришли из математической логики, где в непротиворечивой (sound) теории нельзя доказать неверные утверждения, а в полной можно доказать все верные.

Величина ошибки может быть уменьшена за счёт многократного повторения, поэтому в определениях она выбирается константой согласно традиции.

**Определение 8.3.** Классом **BPP** называется класс языков  $A$ , для которых существует полиномиальный в худшем случае вероятностный алгоритм  $V$ , такой что:

- если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] \geq \frac{2}{3}$ ;
- если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] \leq \frac{1}{3}$ .

Иными словами, вероятности ошибок и первого, и второго рода должны быть меньше  $\frac{1}{3}$ . Аббревиатура **BPP** расшифровывается как “bounded-error probabilistic polynomial” — полиномиальный вероятностный алгоритм с ограниченной ошибкой. Ограниченность понимается как отделённость от  $\frac{1}{2}$ : если поставить  $> \frac{1}{2}$  и  $< \frac{1}{2}$  вместо  $\geq \frac{2}{3}$  и  $\leq \frac{1}{3}$  соответственно, получится определение класса **RP** (probabilistic polynomial), который мы подробнее изучим в разделе про сложность задач подсчёта.

**Определение 8.4.** Классом **RP** называется класс языков  $A$ , для которых существует полиномиальный в худшем случае вероятностный алгоритм  $V$ , такой что:

- если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] \geq \frac{1}{2}$ ;
- если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] = 0$ .

**Определение 8.5.** Классом **coRP** называется класс языков  $A$ , для которых существует полиномиальный в худшем случае вероятностный алгоритм  $V$ , такой что:

- если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] = 1$ ;
- если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] \leq \frac{1}{2}$ .

Таким образом, для класса **RP** (randomized polynomial) равна нулю вероятность ошибки первого рода, а для **coRP** — второго. Ясно, что  $A \in \mathbf{RP}$  эквивалентно  $\bar{A} \in \mathbf{coRP}$ , что оправдывает обозначение **coRP**. Наконец, есть и класс с нулевой ошибкой. Если время работы алгоритма полиномиально в худшем случае, то нулевая ошибка гарантировала бы существование полиномиального алгоритма. Для определения нового класса используется ожидаемое время работы.

**Определение 8.6.** Классом **ZPP** называется языков языков  $A$ , для которых существует вероятностный алгоритм  $V$ , такой что  $x \in A$  при всех  $r$  эквивалентно  $V(x, r) = 1$ , а для каждого  $x$  ожидаемое по  $r$  время работы полиномиально.

Обозначение **ZPP** расшифровывается как “zero-error probabilistic polynomial”. Ясно, что  $\mathbf{P} \subset \mathbf{ZPP}$ ,  $\mathbf{P} \subset \mathbf{RP} \subset \mathbf{BPP}$  и  $\mathbf{P} \subset \mathbf{coRP} \subset \mathbf{BPP}$ . Также ясно, что если  $A \in \mathbf{BPP}$ , то и  $\bar{A} \in \mathbf{BPP}$ , ведь определение симметрично. Поэтому  $\mathbf{BPP} = \mathbf{coBPP}$ . Вот ещё несколько несложных соотношений:

**Утверждение 8.7.**  $\mathbf{RP} \subset \mathbf{NP}$ .

*Доказательство.* Действительно, любое значение  $r$ , при котором  $V(x, r) = 1$ , будет доказательством того, что  $x \in A$ . Можно сказать, что **NP** тоже определяется вероятностным способом: если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] > 0$ , а если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] = 0$ .  $\square$

Можно и на **NP** посмотреть как на вероятностный класс:  $A \in \mathbf{NP}$ , если существует полиномиальный в худшем случае вероятностный алгоритм  $V$ , такой что:

- если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] > 0$ ;
- если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] = 0$ .

**Утверждение 8.8.  $\mathbf{BPP} \subset \mathbf{PSPACE}$ .**

*Доказательство.* Действительно, на полиномиальной памяти можно найти количество таких  $r$ , что  $V(x, r) = 1$ , посчитать вероятность и сравнить её с  $\frac{1}{3}$  и  $\frac{2}{3}$ .  $\square$

Аналогично доказывается и более сильное включение:

**Утверждение 8.9.  $\mathbf{RP} \subset \mathbf{PSPACE}$ .**

*Доказательство.* Действительно, долю  $r$  можно найти точно и сравнить её с  $\frac{1}{2}$ .  $\square$

Теперь докажем, что «нульсторонняя ошибка» эквивалентна двум односторонним с разных сторон.

**Утверждение 8.10.  $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$ .**

*Доказательство.* Пусть  $A \in \mathbf{ZPP}$ . Докажем, что  $A \in \mathbf{RP}$ . Пусть  $V$  — алгоритм, распознающий  $A$  и работающий в среднем за  $p(n)$ . Рассмотрим такой алгоритм: запустить  $A$  на  $2p(n)$  шагов. Если он закончил работу, вернуть тот же ответ, иначе вернуть 0. Если  $x \notin A$ , то в любом случае будет возвращён 0. Если же  $x \in A$ , то по неравенству Маркова с вероятностью не меньше  $\frac{1}{2}$  алгоритм закончит работу и потому ответ будет 1. Значит,  $A \in \mathbf{RP}$ . Вложение  $A \in \mathbf{coRP}$  доказывается аналогично, нужно только при отсутствии ответа вместо 1 возвращать 0.

Обратно, пусть  $A \in \mathbf{RP} \cap \mathbf{coRP}$ . Пусть  $V$  — алгоритм, не допускающий ошибок первого рода (такой есть, т.к.  $A \in \mathbf{RP}$ ), а  $W$  — не допускающий ошибок второго рода (такой есть, т.к.  $A \in \mathbf{coRP}$ ). Запустим алгоритм  $V$  на входе  $x$ . Если он вернул 1, то заведомо  $x \in A$ . Иначе запустим  $W(x)$ . Если он вернул 0, то заведомо  $x \notin A$ . Так будем повторять со свежими случайными битами, пока не получим  $V(x) = 1$  или  $W(x) = 0$ . Вероятность получения ответа на каждой стадии не меньше  $\frac{1}{2}$ , поэтому ожидаемое число стадий не больше 2. Значит,  $A \in \mathbf{ZPP}$ . Заметим, что при таком алгоритме не исключены бесконечные ветви, хотя они и возникают с нулевой вероятностью. Хотя мы и не запрещали их в определении, их можно избежать: после полиномиального числа стадий вероятность ошибки останется экспоненциально маленькой, и можно в этом случае запустить экспоненциальный алгоритм.  $\square$

### 8.3.2 Амплификация (уменьшение ошибки)

Константы  $\frac{1}{3}$ ,  $\frac{2}{3}$  и  $\frac{1}{2}$  в наших определениях были выбраны произвольно. Оказывается, что определение не зависит от того, какие именно константы выбрать. Более того, можно выбрать вероятности как функции от  $n$ , главное, чтобы разрыв оставался не меньше обратного полинома. Идея состоит в том, чтобы запускать один и тот же алгоритм много раз со свежими случайными битами и выбирать ответ по большинству. Такая процедура называется амплификацией алгоритма. Начнём с определения вероятностных классов для разных ошибок.



**Определение 8.11.** Пусть  $\alpha(n)$  и  $\beta(n)$  — две функции из  $\mathbb{N}$  в  $[0, 1]$ , такие что  $0 \leq \alpha(n) < \beta(n) \leq 1$  при всех  $n$ . Классом  $\mathbf{BPP}_{\alpha(n), \beta(n)}$  назовём множество таких языков  $A$ , для которых существует полиномиальный в худшем случае вероятностный алгоритм  $V$ , такой что:

- если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] \geq \beta(|x|)$ ;
- если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] \leq \alpha(|x|)$ .

Таким образом,  $\alpha(|x|)$  является верхней оценкой на вероятность  $\alpha$ -ошибки, а  $1 - \beta(x)$  — верхней оценкой на вероятность  $\beta$ -ошибки.

Все рассмотренные классы (кроме  $\mathbf{ZPP}$ ) можно записать в новых обозначениях:  $\mathbf{BPP} = \mathbf{BPP}_{\frac{1}{3}, \frac{2}{3}}$ ,  $\mathbf{RP} = \mathbf{BPP}_{0, \frac{1}{2}}$ ,  $\mathbf{coRP} = \mathbf{BPP}_{\frac{1}{2}, 1}$ , и даже  $\mathbf{NP} = \mathbf{BPP}_{0, \frac{1}{2^{\text{poly}(n)}}}$  (эта запись не совсем строгая, но отражает суть). Ясно, что классы  $\mathbf{BPP}_{\alpha, \beta}$  монотонны в следующем смысле: если  $\gamma(n) < \alpha(n)$ , а  $\delta(n) > \beta(n)$ , то  $\mathbf{BPP}_{\alpha(n), \beta(n)} \supset \mathbf{BPP}_{\gamma(n), \delta(n)}$ . Мы докажем, что в широком диапазоне это соотношение выполняется как равенство. Вначале проведём рассуждение для односторонней ошибки:

**Теорема 8.12.** Пусть  $c$  и  $d$  — целые числа. Тогда  $\mathbf{BPP}_{0, \frac{1}{n^c}} = \mathbf{BPP}_{0, 1 - \frac{1}{2^{n^d}}} = \mathbf{RP}$  и  $\mathbf{BPP}_{1 - \frac{1}{n^c}, 1} = \mathbf{BPP}_{\frac{1}{2^{n^d}}, 1} = \mathbf{coRP}$ .

*Доказательство.* Два утверждения доказываются полностью аналогично. Для примера докажем первое. Новый алгоритм будет таким: запустим старый алгоритм  $K$  раз с независимыми случайными битами. Если хотя бы один раз получена единица, возвратим единицу, иначе ноль. (Т.е. возвратим дизъюнкцию результатов). Если на самом деле ответ «да», то вероятность того, что все  $K$  раз алгоритм вернёт 0, равна  $(1 - \frac{1}{n^c})^K$ . В силу второго замечательного предела это примерно равно  $e^{-\frac{K}{n^c}}$ . Таким образом, если взять  $K = n^{c+d}$ , то вероятность ошибки не превысит  $\frac{1}{2^{n^d}}$ , что и требовалось. Равенство  $\mathbf{RP}$  доказывается вложениями  $\mathbf{BPP}_{0, 1 - \frac{1}{2^{n^d}}} \subset \mathbf{RP} \subset \mathbf{BPP}_{0, \frac{1}{n^c}}$ .  $\square$

Для двусторонней ошибки мы докажем вот такое утверждение:

**Теорема 8.13.** Пусть  $c$  и  $d$  — целые числа, а  $\tau \in (0, 1)$ . Тогда  $\mathbf{BPP}_{\tau - \frac{1}{n^c}, \tau + \frac{1}{n^c}} = \mathbf{BPP}_{\frac{1}{2^{n^d}}, 1 - \frac{1}{2^{n^d}}}$ . В частности, для любой константы  $\varepsilon \in (0, \frac{1}{2})$  выполнено  $\mathbf{BPP}_{\varepsilon, 1 - \varepsilon} = \mathbf{BPP}$ .

*Доказательство.* Идея доказательства очень проста: запустим алгоритм независимо много раз и сравним долю успехов с  $\tau$ . Если она была больше, вернём 1, если меньше, то 0. Для доказательства теоремы нужно показать, что полиномиального числа запусков хватит.

Для примера, пусть вероятность успеха при одном запуске равна  $\tau + \frac{1}{n^c}$ . Обозначим через  $\xi_i$  случайную величину, которую возвращает алгоритм на  $i$ -ом запуске. Эта величина бернуллиевская с вероятностью успеха  $\tau + \frac{1}{n^c}$ . Требуется оценить вероятность того, что после  $K$  запусков будет больше, чем  $\tau K$  успехов, т.е.  $\xi_1 + \dots + \xi_K > \tau K$ . А именно, при полиномиальном  $K$  эта вероятность должна быть экспоненциально близка к единице. Интуитивно это объясняется так: по центральной предельной теореме случайная величина  $S_K = \frac{\xi_1 + \dots + \xi_K}{K}$  стремится по распределению к нормальной случайной величине со средним  $\tau + \frac{1}{n^c}$  и дисперсией порядка  $\frac{1}{K}$ . Если  $K = n^{2c} \cdot L$ , то  $\tau$  отстоит от среднего примерно на

$\sqrt{L}$  стандартных отклонений, и потому вероятность, что  $S_K$  окажется меньше  $\tau$ , имеет порядок  $e^{-L}$ , что и требовалось.

Для точного рассуждения нужны оценки вероятности больших отклонений, например, неравенство Чернова. □

ДОБАВИТЬ  
ТОЧНЫЕ  
ОЦЕНКИ!

### 8.3.3 Роль вероятностного распределения

*Наш лектор намеревался пройтись по списку студентов и, останавливаясь на каждой фамилии, бросать монетку: орёл означал бы «отлично», а решка — «хорошо». Но прежде чем он приступил к бросанию монеты, его пронзила ужасная мысль: что, если монета слегка несимметрична?*

Г. Гамов, М. Стерн, *Занимательная математика*

Возникает естественный вопрос: как изменится эффективность вероятностных алгоритмов, если вместо равномерного распределения на  $\{0, 1\}^{|r|}$  взять какое-нибудь ещё? Ведь источник случайных битов на практике может работать как угодно! В общем случае на этот вопрос отвечает глубокая и сложная теория экстракторов или, шире, теория псевдослучайности. Мы рассмотрим частный случай, когда биты  $r$  остаются независимы и одинаково распределены, но их распределение смещено: каждый из них равен единице с вероятностью  $p$  и нулю с вероятностью  $1 - p$ .

**Определение 8.14.** Обозначим через  $\mathbf{BPP}_p$  аналог  $\mathbf{BPP}$ , где все вероятности берутся по распределению на  $r$ , такому что его биты независимы и равны нулю с вероятностью  $p$ , а единице с вероятностью  $1 - p$ .

**Теорема 8.15.** Для любого  $p \in (0, 1)$  выполнено  $\mathbf{BPP} \subset \mathbf{BPP}_p$ .

*Доказательство.* Для доказательства требуется смоделировать равномерные случайные биты при помощи неравномерных с не более чем полиномиальным замедлением. Это делается так: возьмём два неравномерных бита. Если они не равны друг другу, то возьмём первый из них в качестве равномерного. Если же равны, возьмём ещё два неравномерных бита, и так далее.

Вероятность того, что выпали разные биты, равна  $2p(1 - p)$ , причём условные вероятности комбинаций 01 и 10 равны по  $\frac{1}{2}$ . Если провести  $\frac{n}{p(1-p)}$  попыток, то вероятность неудачи во всех будет экспоненциально малой. Таким образом, алгоритм будет таким: попробуем получить  $|r|$  равномерных случайных битов вышеуказанным образом. Если хотя бы один получить не удалось, завершим работу с произвольным ответом. Иначе запустим прежний алгоритм. Вероятность того, что какой-то бит не получен, будет экспоненциально мала. Условно на то, что все биты получены, их распределение будет действительно равномерным. Значит, и вероятности успеха и неудачи будут такими же с экспоненциальной точностью. □

В обратную сторону вложение выполнено не всегда.

**Теорема 8.16.** Если  $p \in (0, 1)$  — полиномиально вероятностно вычислимое число, т.е.  $i$ -ый знак его двоичного разложения можно получить вероятностным алгоритмом за полиномиальное от  $i$  время, то  $\mathbf{BPP}_p = \mathbf{BPP}$ . Для некоторых невычислимых  $p$  класс  $\mathbf{BPP}_p$  шире  $\mathbf{BPP}$ . Такие  $p$  образуют всюду плотное множество на  $(0, 1)$ .

Начнём с простого примера: как получить вероятность  $\frac{1}{3}$ , имея только симметричную монетку? Нужно бросить её дважды. Если выпало 00, вернуть 0; если выпало 10 или 11, то вернуть 1, а если выпало 01, повторить процедуру. Количество итераций следует установить достаточно большим, чтобы вероятностью неполучения ответа можно было пренебречь. Такую процедуру можно обобщить на все полиномиально вероятностно вычислимые числа.

*Доказательство.* Докажем, что для вычислимого  $p$  выполнено  $\mathbf{BPP}_p \subset \mathbf{BPP}$ . Для этого нужно смоделировать монетку с вероятностью успеха  $p$  при помощи симметричной. Пусть алгоритм использует  $m$  случайных битов. Мы будем делать так: для всех  $i$  от 1 до  $K = m^2$  будем вычислять  $i$ -ый знак и кидать монетку. Если монетка оказалась больше (т.е. знак 0, а выпала 1), то возьмём в качестве результата 1, если монетка меньше (т.е. знак 1, а выпал 0), то выдадим 0, иначе перейдём к следующему  $i$ . Если на всех  $K$  этапах монетка совпала со знаком, то тоже выдадим 0. Таким образом, вероятность выпадения 1 будет равна минимальному двоично-рациональному числу со знаменателем не больше  $2^K$ , которое не меньше  $p$ , т.е.  $\hat{p} = \frac{\lfloor 2^K p \rfloor}{2^K}$ . Поскольку вероятность успеха для одной монетки квадратично-экспоненциально близка к  $p$ , то и вероятность выпадения конкретной полиномиальной последовательности будет квадратично-экспоненциально близка к вероятности с монеткой  $p$ . Поэтому и вероятность успеха всего алгоритма изменится экспоненциально мало (тут будет умножение на обычную экспоненту). Ещё некоторый разрыв нужно оставить на ошибку при вычислении битов  $p$ , но это также не повлияет.<sup>2</sup>

Перейдём к рассмотрению невычислимых  $p$ . Первая идея заключается в том, чтобы вычислять знаки самого  $p$  путём кидания большого числа монеток и подсчёта средней доли единиц. Эта доля будет близка к  $p$ , что и позволит вычислять знаки  $p$ . К сожалению, кидать монетки нужно в экспоненциальном количестве: чтобы узнать  $n$ -й знак после запятой, необходимо вычислить  $p$  с точностью  $2^{-n}$ , а для этого по центральной предельной теореме нужно порядка  $2^{2n}$  бросаний монетки. Чтобы сделать время вычисления полиномиальным, мы будем вычислять экспоненциальное разрежение знаков  $p$ , т.е. распознавать язык  $\{1^{2^n} \mid p_n = 1\}$ . Однако возникает другая проблема: если  $p$  расположено очень близко к некоторой последовательности двоично-рациональных чисел, то и точности  $2^{-n}$  не хватит: если  $|p - \frac{x}{2^n}| < \varepsilon$ , то для определения  $n$ -го знака нужно вычислить  $p$  с точностью  $\varepsilon$ . Эта разность может быть сколь угодно малой, в т.ч. сверхэкспоненциально малой, и даже меньше обратной функции Аккермана (тогда никакое вычислимое разрежение не подойдёт). Поэтому мы построим явным образом  $p$ , для которого таких проблем заведомо нет.

<sup>2</sup>Точные оценки такие: пусть биты  $p$  вычисляются с точностью  $\frac{1}{2^K}$ . Тогда все  $K$  битов будут вычислены верно с вероятностью не меньше  $1 - \frac{K}{2^K}$ . Тогда вероятность выпадения 1, полученная в алгоритме, отличается от  $p$  не больше, чем на  $\frac{K+1}{2^K}$ . Обозначим величину отклонения за  $\varepsilon$ . Тогда вероятность выпадения конкретной последовательности с  $k$  единицами и  $l$  нулями вместо  $p^k(1-p)^l$  будет равна  $(p+\varepsilon)^k(1-p-\varepsilon)^l$ , что равно  $p^k(1-p)^l(1+\frac{\varepsilon}{p})^k(1-\frac{\varepsilon}{1-p})^l$ . По формуле Тейлора это примерно равно  $p^k(1-p)^l(1+\frac{k\varepsilon}{p})(1-\frac{l\varepsilon}{1-p})$  и отличается от  $p^k(1-p)^l$  меньше, чем на  $cn\varepsilon$ , где  $c = \max\{\frac{1}{p}, \frac{1}{1-p}\}$ , а  $n = k+l$ . Поскольку всего есть  $2^m$  последовательностей, вероятность любого множества изменится меньше, чем на  $2^m \cdot cn\varepsilon$ . Поскольку  $\varepsilon$  имеет порядок  $\frac{1}{2^{m^2}}$ , изменение вероятности всё равно будет экспоненциально маленьким. В частности, так изменится вероятность того, что алгоритм выдаст единицу, что и требовалось.

Пусть двоичное разложение числа  $p$  задано следующим образом:

$$p_i = \begin{cases} 0, & i = 3k + 1; \\ 1, & i = 3k + 2; \\ 0, & i = 3k \text{ и } M_k(k) \text{ не определена;} \\ 1, & i = 3k \text{ и } M_k(k) \text{ определена,} \end{cases}$$

где под  $M_k$  понимается  $k$ -я машина Тьюринга. Ясно, что число  $p$  кодирует проблему самоприменимости и потому невычислимо ни за какое время, но при этом в нём нигде не идёт больше двух нулей или единиц подряд.

Теперь покажем, что при помощи  $p$ -монетки можно за полиномиальное время с малой ошибкой решать унарную экспоненциально разреженную проблему самоприменимости, т.е. распознавать язык  $\{1^{2^k} \mid M_k(k) \text{ определена}\}$ . Ясно, что если в слове есть нули или число единиц не является степенью двойки, то его можно отвергнуть без всякой монетки. Если же слово состоит из  $2^k$  единиц, то нужно подбросить монетку  $2^{6k+4}$  раз и посчитать долю единиц. Тогда  $3k$ -ый бит этой доли и будет с высокой вероятностью правильным ответом. Поскольку этот язык неразрешим, он заведомо не лежит в **ВРР**, т.е. **ВРР** <sub>$p$</sub>  и правда шире.

Ясно, что кодирование проблемы самоприменимости можно начинать не с первого символа, а с любого другого, а начало заполнить любыми цифрами. Это гарантирует всюду плотность таких примеров на  $(0, 1)$ .

ДОБАВИТЬ СНОСКУ С ОЦЕНКАМИ ИЗ ЦПТ

□

Заметим, что такой конструкцией можно построить не только абсолютно невычислимые  $p$ , но и  $p$ , невычислимые за линейно-экспоненциальное время, для которых **ВРР** <sub>$p$</sub>  будет шире **ВРР**. При этом можно показать диагональной техникой, что существует и невычислимое  $p$ , для которого **ВРР** <sub>$p$</sub>  = **ВРР**.

Записать это доказательство хоть куда-нибудь

### 8.3.4 Связь со схемами из функциональных элементов

**Теорема 8.17** (Эйдельман<sup>3</sup>, [6]). **ВРР**  $\subset$  **P**/<sub>poly</sub>.

*Доказательство.* Идея доказательства основана на следующем: уменьшение ошибки позволяет выбрать одно значение случайных битов, годящееся для всех входов. Это значение можно «запаять» в схему.

Более подробно: пусть  $A \in \mathbf{ВРР}$ . Рассмотрим алгоритм  $V$ , дающий для слов длины  $n$  правильный ответ с вероятностью  $1 - \varepsilon$ , где  $\varepsilon < \frac{1}{2^n}$ . Тогда вероятность того, что  $V(x, r)$  даёт неправильный ответ хотя бы для одного  $x$ , не больше  $2^n \varepsilon < 1$ . Значит, с положительной вероятностью результат  $V(x, r)$  даёт правильный ответ для всех  $x$ . Значит, найдётся некоторое  $r_0$ , такое что для всех  $x$  длины  $n$  выполнено  $V(x, r_0) = 1$  тогда и только тогда, когда  $x \in A$ . Поскольку  $V$  работает за полиномиальное время, стандартное преобразование  $V(x, r_0)$  в схему со входом  $x$  даёт схему полиномиального размера, распознающую  $A^{=n}$ . □

<sup>3</sup>На русский фамилию этого учёного чаще транскрибируют как Адлеман, мы предпочитаем традиционную русскую транскрипцию этой еврейской фамилии

### 8.3.5 Связь с полиномиальной иерархией

Мы уже выяснили, что  $\mathbf{BPP} \subset \mathbf{PSPACE}$ . Использование более продвинутых методов позволяет включить  $\mathbf{BPP}$  в полиномиальную иерархию. А именно, верна такая теорема:

**Теорема 8.18** (Сипсер–Гач–Лаутеман, [89, 125]).  $\mathbf{BPP} \subset \Sigma_2^p \cap \Pi_2^p$ .

*Доказательство.* Как мы уже отметили,  $\mathbf{BPP}$  совпадает с  $\mathbf{coBPP}$ . Поэтому достаточно доказать только  $\mathbf{BPP} \subset \Sigma_2^p$ : из этого будет следовать, что  $\mathbf{coBPP} \subset \Pi_2^p$ , т.е.  $\mathbf{BPP} \subset \Pi_2^p$ . Идея доказательства заключается в следующем: сначала сделаем ошибку достаточно маленькой, для каждого  $x$  множество подходящих  $r$  будет либо очень большим, либо очень маленьким. Если оно большое, то полиномиальным количеством его сдвигов можно покрыть всё пространство, а если оно маленькое, то нельзя. Тот факт, что сдвиги покрывают всё пространство, можно записать  $\Sigma_2$ -формулой.

Изложим этот план более детально. Пусть  $A \in \mathbf{BPP}$ . Рассмотрим алгоритм  $V$ , дающий для слов длины  $n$  правильный ответ с вероятностью  $1 - \varepsilon$ , где  $\varepsilon < \frac{1}{2^n}$ . Пусть он требует  $m = \text{poly}(n)$  случайных битов. Для каждого  $x$  обозначим через  $R_x$  множество таких  $r$ , что  $V(x, r) = 1$ . Если  $x \in A$ , то  $|R_x| > 2^m(1 - \varepsilon)$ , а если  $x \notin A$ , то  $|R_x| < 2^m\varepsilon$ . Будем называть *сдвигом* строки  $r$  из  $\{0, 1\}^m$  на вектор  $s$  побитовое сложение  $r \oplus s$  (можно это рассматривать как сдвиг в векторном пространстве размерности  $m$  над полем из двух элементов). Также будем говорить о сдвиге множества  $R_x$ :  $R_x \oplus s = \{r \oplus s \mid r \in R_x\}$ . Докажем, что для некоторого  $k$  при  $|R_x| > 2^m(1 - \varepsilon)$  найдутся сдвиги  $s_1, \dots, s_k$ , такие что  $R_x \oplus s_i$  в объединении дают всё  $\{0, 1\}^m$ , а при  $|R_x| < 2^m\varepsilon$  таких сдвигов не найдётся.

Для большого  $R_x$  воспользуемся простейшим вероятностным методом. Пусть  $s_1, \dots, s_k$  выбраны случайно и независимо. Если сдвиги  $R_x$  не покрывают всё пространство, то найдётся  $y$ , не принадлежащий ни одному из множеств  $R_x \oplus s_i$ . Это эквивалентно тому, что  $R_x$  не содержит ни один из векторов  $y \oplus s_1, \dots, y \oplus s_k$ . Эти вектора тоже распределены равномерно и независимо, поэтому для фиксированного  $y$  вероятность того, что все они оказались вне  $R_x$ , равна  $\varepsilon^k$ . Вероятность же того, что такое случится хотя бы для одного  $y$ , не превосходит  $2^m\varepsilon^k$ . Если  $k > \frac{m}{\varepsilon}$ , то эта вероятность будет меньше 1, то есть с положительной вероятностью такого  $y$  не найдётся, т.е. для некоторого набора сдвигов образы  $R_x$  покроют всё пространство, что и требовалось.

Если же размер  $R_x$  меньше  $2^m\varepsilon$ , то  $k$  его сдвигов покроют меньше  $2^m\varepsilon k$  элементов, что меньше  $2^m$  при  $k < 2^n$ . Таким образом, при  $k = m$  всё пространство быть покрытым не может.

Осталось показать, как указанное свойство можно задать  $\Sigma_2$ -формулой. Это делается просто:  $\exists s_1 \dots \exists s_k \forall y (V(x, y \oplus s_1) \vee \dots \vee V(x, y \oplus s_k))$ . Поскольку  $k$  полиномиально, а  $V$  работает за полиномиальное время, выражение в скобках также вычисляется за полиномиальное время. Таким образом,  $\mathbf{BPP} \subset \Sigma_2^p$ , а значит, и  $\mathbf{BPP} \subset \Pi_2^p$ , что и требовалось.  $\square$

Сразу получаем такое следствие:

**Следствие 8.19.** Если  $\mathbf{P} = \mathbf{NP}$ , то  $\mathbf{P} = \mathbf{BPP}$ .

*Доказательство.* Действительно, если  $\mathbf{P} = \mathbf{NP}$ , то  $\mathbf{P} = \mathbf{PH}$ , а  $\mathbf{BPP} \subset \mathbf{PH}$ , как мы только что выяснили.  $\square$

В этом смысле недетерминизм могущественнее случайности, хотя текущий объём знаний не исключает даже равенства  $\mathbf{BPP} = \mathbf{NEXP}$ .

### 8.3.6 ВРР-полные задачи?

Многие сложностные классы структурируются за счёт наличия полных задач. Возникает естественный вопрос: существуют ли такие в классе **ВРР**? Если бы они существовали, то проблема  $\mathbf{P} = \mathbf{ВРР}$  потенциально могла бы быть решена построением полиномиального алгоритма для одной из них. К сожалению, с классом **ВРР** имеется проблема: не всякая вероятностная машина вообще задаёт какой-либо язык. Ведь если вероятность принять хоть один  $x$  попала внутрь интервала  $(\frac{1}{3}, \frac{2}{3})$ , то интерпретировать результат уже не получится. Про этот эффект говорят, что определение **ВРР** *семантическое*, а не синтаксическое.

Рассмотрим естественный кандидат на **ВРР**-полноту: язык  $\{(M, x, 1^t) \mid M(x) = 1 \text{ с вероятностью не меньше } \frac{2}{3} \text{ и не более, чем за } t \text{ шагов}\}$ . Разумеется, он будет **ВРР**-труден: с его помощью можно распознать любой язык из **ВРР**. Однако сам он в **ВРР** скорее всего не лежит, на самом деле он полон в классе **РР**. Разумеется, полностью исключать наличие **ВРР**-полных языков нельзя: если  $\mathbf{ВРР} = \mathbf{P}$ , то раз они есть в **Р**, то будут и в **ВРР**.

Возможно, более верный взгляд на задачи из **ВРР** (и вообще на задачи распознавания) — это задачи с предусловием (promise problems). Условие такой задачи состоит из двух непересекающихся языков  $A_y$  и  $A_n$ . Решающий эту задачу алгоритм должен выдать положительный ответ на элементах  $A_y$ , отрицательный на элементах  $A_n$ , а на элементах не из  $A_y \cup A_n$  он может выдавать что угодно. Ясно, что обычная задача распознавания является частным случаем задачи с предусловием: достаточно положить  $A_n = \bar{A}_y$ . При этом предусловие часто возникает само собой: например, в задаче о раскраске графа нас интересовало, есть раскраска или нет, а не задаёт ли вообще поданное на вход слово некоторый граф. (Разумеется, в этом случае предусловие легко проверяется).

Практически к любому из уже изученных классов можно приставить приставку **Promise**-. Например, **Promise-P** — класс задач с предусловием, которые решаются полиномиальным алгоритмом, **Promise-NP** — класс задач с предусловием, которые решаются полиномиальным недетерминированным алгоритмом. Точно также и **Promise-BPP** — класс задач с предусловием, которые решаются полиномиальным вероятностным алгоритмом. Это означает, что на элементах  $A_y$  алгоритм возвращает 1 с вероятностью не меньше  $\frac{2}{3}$ , а на элементах  $A_n$  он возвращает 0 с вероятностью не меньше  $\frac{1}{3}$ . И вот для **Promise-BPP** стандартная полная задача есть. Она определяется так:  $A_y = \{(M, x, 1^t) \mid M(x) = 1 \text{ с вероятностью не меньше } \frac{2}{3} \text{ и не более, чем за } t \text{ шагов}\}$ ,  $A_n = \{(M, x, 1^t) \mid M(x) = 1 \text{ с вероятностью не больше } \frac{1}{3} \text{ и не более, чем за } t \text{ шагов}\}$ .

Более подробно о задачах с предусловием можно почитать в обзоре Голдрайха [63], см. также [93, гл. 25].

## 8.4 Вероятностные алгоритмы на логарифмической памяти

### 8.4.1 Определения и простые включения

Как и раньше, для вероятностных машин можно ограничивать не только время работы, но и использованную память. Если при полиномиальном ограничении на память случайность не добавит вычислительной силы, то при ло-



гарифмическом может добавить. Кратко изучим соответствующие классы:

**Определение 8.20.** Классом **BPL** называется класс языков, которые распознаются вероятностной машиной с двусторонней ошибкой на логарифмической памяти за полиномиальное время. Иными словами,  $A \in \mathbf{BPL}$ , если существует машина  $V$  с двумя аргументами, такая что  $V(x, r)$  вычисляется на логарифмической памяти и за полиномиальное время от  $|x|$ , при этом если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] > \frac{2}{3}$ , а если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] < \frac{1}{3}$ .

**Определение 8.21.** Классом **RL** называется класс языков, которые распознаются вероятностной машиной с односторонней ошибкой на логарифмической памяти за полиномиальное время. Иными словами,  $A \in \mathbf{RL}$ , если существует машина  $V$  с двумя аргументами, такая что  $V(x, r)$  вычисляется на логарифмической памяти и за полиномиальное время от  $|x|$ , при этом если  $x \in A$ , то  $\Pr_r [V(x, r) = 1] > \frac{1}{2}$ , а если  $x \notin A$ , то  $\Pr_r [V(x, r) = 1] = 0$ .

Некоторые авторы обозначения **RL** и **BPL** используют для классов только с логарифмическим ограничением на память, а если добавляется полиномиальное время, обозначают классы как **RLP** и **BPLP**.

Поясним, зачем дополнительно накладывается требование полиномиального времени. Для детерминированных машин это было не нужно, поскольку следовало из логарифмической памяти. Для вероятностных машин это уже зависит от деталей модели. Например, если требовать, чтобы  $r$  было полиномиальной от  $x$  длины, то дополнительно требовать полиномиального времени не нужно. А вот если в качестве  $r$  брать бесконечную ленту, заполненную случайно, то можно составить логарифмический по памяти (но экспоненциальный по времени, причём только в среднем) алгоритм для любого языка из **NL**. Для этого нужно воспользоваться сертификатным определением **NL**, а случайную ленту воспринимать как цепочку сертификатов. Каждый из них нужно проверять, до тех пор пока один не подойдёт или не встретится сертификат из одних нулей. В первом случае слово следует принять, во втором — отвергнуть.<sup>4</sup> Если правильного сертификата нет, то рано или поздно сертификат из одних нулей встретится, и вход будет отвергнут. Если правильные сертификаты есть, то с вероятностью хотя бы  $\frac{1}{2}$  один из них встретится раньше, чем сертификат из одних нулей, и тогда вход будет принят. Дополнительная память нужна только на проверку, что есть единицы, а это даже не логарифм, а константа.

Отдельно встаёт вопрос о том, может ли машина двигаться по ленте случайных битов в обратном направлении. Раньше это было неважно, потому что все выпавшие биты и так можно было запоминать при необходимости. Теперь так делать уже нельзя, поскольку памяти для этого недостаточно, а при отдельной ленте они записаны автоматически. [И ЧТО ДЕЛАТЬ?]

Как и для полиномиального времени, для логарифмической памяти верны теоремы об амплификации и простые вложения  $\mathbf{L} \subset \mathbf{RL} \subset \mathbf{BPL}$  и  $\mathbf{RL} \subset \mathbf{NL}$ . Как следствие,  $\mathbf{RL} \subset \mathbf{P}$ . Верно и более сильное утверждение.

**Теорема 8.22.**  $\mathbf{BPL} \subset \mathbf{P}$ .

*Доказательство.* Идея доказательства заключается в том, чтобы посчитать вероятность прихода в принимающее состояние и сравнить её с  $\frac{1}{3}$  и  $\frac{2}{3}$ . Для доказательства будем использовать модель с двумя функциями перехода, из

<sup>4</sup>Нужно ещё позаботиться, чтобы сертификат из одних нулей точно не был правильным, но это несложно. Например, можно ко всем правильным сертификатам приписать в начало единицу.

которых каждый раз выбирается случайная. В таком случае конфигурационный граф имеет полиномиальный размер. Будем для каждой вершины считать вероятность, что из неё машина попадёт в принимающее состояние. Поскольку машина всегда заканчивает свою работу, циклов в конфигурационном графе быть не может. Поэтому вероятности можно вычислять динамическим программированием, начиная с конца. А именно, в принимающих состояниях вероятность будет равна 1, в отвергающих — 0. Далее, если из данной конфигурации ведут переходы в конфигурации с вероятностями успеха  $p$  и  $q$ , то для данной конфигурации запишем вероятность успеха  $\frac{p+q}{2}$ . Поскольку циклов нет, последовательным применением таких операций вычисляется вероятность для начального состояния, остаётся сравнить её с  $\frac{1}{3}$  и  $\frac{2}{3}$ .

Поскольку граф полиномиален, всего нужно вычислить полиномиальное количество вероятностей. Кроме того, все они будут двоично-рациональными со знаменателем  $2^{\text{poly}(n)}$ , где полином равен числу вершин графа, поэтому каждая из них будет записана полиномиальным числом битов, и все операции можно будет совершить за полином. Значит, описанная процедура будет полиномиальной и **VPL** действительно включено в **P**.  $\square$

### 8.4.2 Случайные блуждания в графах

*Прошло ещё минут десять, и компания очутилась в центре лабиринта. Гаррис хотел сначала сделать вид, будто он именно к этому и стремился, но его свита имела довольно угрожающий вид, и он решил расценить это как случайность.*

Джером К. Джером, *Трое в лодке, не считая собаки*

**Теорема 8.23.** Язык  $\text{UPATH} = \{(G, s, t) \mid \text{в неориентированном графе } G \text{ есть путь из } s \text{ в } t\}$  лежит в **RL**.

Известен более сильный результат (теорема Рейнгольда, [110]):  $\text{UPATH} \in \mathbf{L}$ , однако он доказывается очень сложной комбинаторной техникой.

*Доказательство.* Идея доказательства проста: нужно запустить достаточно длинное, но полиномиальное случайное блуждание из  $s$ . Если во время этого блуждания  $t$  хотя бы один раз посещено, то ответ положительный, иначе отрицательный. Если  $s$  и  $t$  находятся в разных компонентах связности, то, очевидно,  $t$  посещено не будет. Также ясно, что требуется лишь логарифмическая память на хранение текущей вершины. Однако нужно объяснить, почему если  $t$  достижимо, то полиномиального блуждания будет достаточно.

В дальнейшем будем предполагать, что  $G$  связан, состоит из  $n$  вершин и  $m$  (неориентированных) рёбер и не имеет петель и кратных рёбер.<sup>5</sup> Рассмотрим (бесконечное) случайное блуждание с началом в  $s$ . Через  $s_i$  обозначим  $i$ -ю посещённую вершину ( $s_0 = s$ ). Мы докажем, что в пределе все рёбра (с добавленной ориентацией, т.е. порядком посещения) встречаются в нём с одинаковой частотой. Для каждого ребра  $e = (y, z)$  определим его частоту как

<sup>5</sup>Случай петель и кратных рёбер можно свести к стандартному, поставив по вершине в середине каждого ребра. Тогда блуждание в новом графе будет выглядеть как блуждание в старом, где в каждый момент с вероятностью  $\frac{1}{2}$  вершина не изменяется. После удаления таких стояний на месте получится обычное блуждание.



предел  $P_{yz} = \lim_{t \rightarrow \infty} \frac{1}{t} \mathbb{E} \# \{i \in [0, t-1] : (s_i, s_{i+1}) = (y, z)\}$ . (Этот предел существует из-за марковского свойства: будущее поведение блуждания не зависит от истории, поэтому если ребро встретилось хотя бы один раз, то после каждой следующей встречи распределение будет такое же, как после первой. Можно было бы его определить как обратное к среднему времени до первого прохода  $(y, z)$  после старта из  $z$ ). Аналогично определим частоты вершин  $P_y = \lim_{t \rightarrow \infty} \frac{1}{t} \mathbb{E} \# \{i \in [0, t] : s_i = y\}$ . Поскольку блуждание случайно, после посещения вершины  $y$  может последовать любая из соседних вершин равновероятно, откуда  $P_{yz} = \frac{1}{\deg y} P_y$ . С другой стороны, вершина  $y$  может возникнуть только в результате прохождения по какому-то ребру, ведущему в неё, откуда  $P_y = \sum_{x: (x,y) \in E} P_{xy}$ . Совмещая два равенства, имеем  $P_{yz} = \frac{1}{\deg y} \sum_{x: (x,y) \in E} P_{xy}$ . Из этого равенства уже легко следует, что все  $P_{yz}$  равны: иначе бы для некоторого максимального  $P_{yz}$  некоторые из  $P_{xy}$  не были бы максимальными, и равенство бы не выполнялось.<sup>6</sup> Поскольку сумма всех частот равна единице, каждая из них равна  $\frac{1}{2m}$ , а  $P_y = \frac{\deg y}{2m}$ .

К сожалению, дальше нельзя просто сказать, что в среднем через  $2m$  шагов встретится ребро, ведущее в  $t$ . Дело в том, что  $\frac{1}{2m}$  — только предельная частота встречи каждого ребра, а сами эти встречи могут быть распределены неравномерно: в одних фрагментах блуждания существенно чаще, в других — реже. Поэтому нужно непосредственное рассуждение.

Обозначим через  $E(y, z)$  среднее число шагов, после которого случайное блуждание из вершины  $y$  впервые придёт в  $z$ . Из  $P_y = \frac{\deg y}{2m}$  следует, что  $E(y, y) = \frac{2m}{\deg y}$ . Отсюда, если  $y$  и  $z$  связаны ребром, то  $E(y, z) \leq 2m$ : после каждого посещения  $y$  блуждание с вероятностью  $\frac{1}{\deg y}$  пойдёт в  $z$ , значит в среднем через  $\deg y$  раз оно туда и пойдёт. Кроме того, можно посетить  $z$  и по другому маршруту,<sup>7</sup> отсюда знак  $\leq$ .

Далее по индукции докажем, что  $E(y, z) \leq 2kt$ , если от  $y$  до  $z$  идёт путь длины  $k$ . Действительно, база при  $k = 1$  уже доказана. Если же мы продлеваем путь на ещё одно ребро  $(z, w)$ , то сначала в среднем не больше чем за  $2kt$  шагов блуждание доходит до  $z$ , а потом ещё не больше чем за  $2m$  — до  $w$ , что и даёт нужную оценку. Поскольку какой-то путь от  $s$  до  $t$  короче  $n$ , то среднее время достижения  $t$  составит меньше  $2mn$ . По неравенству Маркова если запустить блуждание на  $4mn$  шагов, то с вероятностью больше  $\frac{1}{2}$  оно достигнет  $t$ , что и требовалось.  $\square$

Мы получили, что требуется запустить блуждание длины  $O(n^3)$ . Эта оценка точна для графа-«леденца», состоящего из клики и цепочки по  $\frac{n}{2}$  вершин (см. рис. 8.1). Действительно, блужданию из  $s$  нужно сначала попасть в вершину  $g$ , потом перейти в вершину  $h$ , а потом пройти по всей цепочке до  $t$ , не вернувшись в клику. Вершина  $g$  будет посещаться в среднем раз в  $\frac{n}{2}$  шагов внутри клики, т.к. все  $\frac{n}{2}$  вершин должны встречаться с одинаковой частотой. Блуждание будет переходить из  $g$  в  $h$  также раз в  $\frac{n}{2}$  посещений, т.е. среднее время до попадания в  $h$  равно  $\frac{n^2}{4}$ . После этого нужно, чтобы случайное блуждание по цепочке сдвинулось на  $\frac{n}{2}$  шагов вправо раньше, чем на 1 шаг влево. При помощи рекуррентной формулы легко показать, что это также произойдёт в среднем раз в  $\frac{n}{2}$  попыток. Таким образом, среднее время прихода в  $t$  составит  $\Omega(n^3)$ , что с точностью до

<sup>6</sup>В этом месте используется неориентированность  $G$ : для этого вывода нужно, чтобы входящая и исходящая степень у вершины  $y$  совпадали. Таким образом, если  $u$  ориентированного графа в каждой вершине входящая и исходящая степени совпадают, то для него теорема тоже будет верна.

<sup>7</sup>Если  $(y, z)$  не мост.

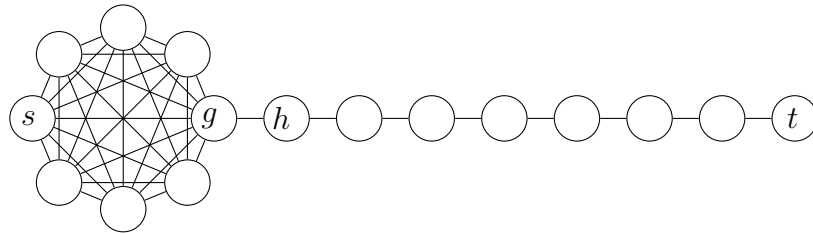


Рис. 8.1: Граф-«леденец», на котором случайное блуждание дольше всего добирается из  $s$  в  $t$ , показан для 16 вершин.

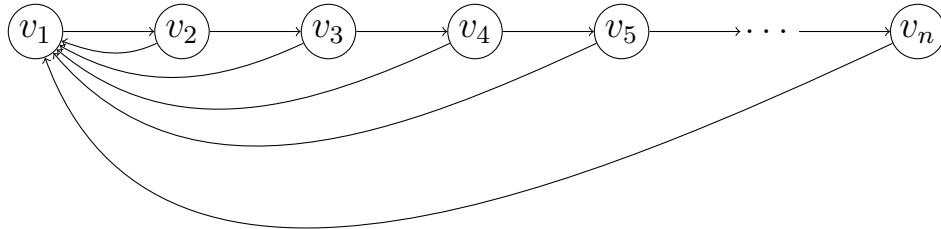


Рис. 8.2: Ориентированный граф, на котором случайное блуждание не срабатывает.

константного множителя совпадает с верхней оценкой.

Заметим, что для ориентированных графов простейший алгоритм случайного блуждания не подойдёт. Например, можно рассмотреть цепочку из  $n$  вершин, из каждой из которых есть рёбра в следующую и в первую (см. рис. 8.2). Если каждый раз идти случайно вперёд или назад, то для прохода из первой вершины до последней нужно  $n$  раз подряд пойти вперёд, что случится в среднем через  $2^n$  шагов. С другой стороны, использованная память останется логарифмической, что лишний раз иллюстрирует необходимость ограничения на время.

Отметим ещё одно красивое свойство блужданий по неориентированным графам: существует универсальное блуждание полиномиальной длины, посещающее все вершины любого связного графа. Разумеется, во всех вершинах должен быть одинаковый набор возможных направлений, поэтому мы будем считать граф  $d$ -регулярным, т.е. степени всех вершин будут равны  $d$ . Это не слишком обременительное условие: можно дополнить степени всех вершин до  $d$  кратными петлями.

**Теорема 8.24.** *Существует полином  $p(\cdot)$ , такой что для любого  $n$  существует последовательность универсального блуждания  $\mathbf{z} \in \{1, \dots, d\}^{p(n)}$ . Универсальность означает, что в любом связном  $d$ -регулярном графе  $G$  при любой начальной вершине  $v_0$  блуждание, определяемое по правилу « $v_{i+1}$  есть  $z_i$ -ый сосед вершины  $v_i$ », посещает все вершины не более чем за  $p(n)$  шагов.*

*Доказательство.* Как известно, в связном графе есть цикл не длиннее  $2n$ , проходящий через все вершины. (Нужно взять остовное дерево и обойти его по правилу левой руки. Вообще-то получится длина  $2n - 2$ , но не будем обращать внимание на мелочи). Обозначим этот цикл через  $w_1, \dots, w_{2n}$ . Аналогично доказательству предыдущей теоремы получаем, что среднее время, за которое случай-

ное блуждание, начав с  $w_i$ , посетит последовательно  $w_{i+1}, \dots, w_{2n}, w_1, \dots, w_{i-1}$ , не превосходит  $4mn$ . По неравенству Маркова, после  $8mn$  шагов блуждание, начав с любой точки, посетит все вершины с вероятностью не меньше  $\frac{1}{2}$ . Поскольку начальная точка не важна, то после  $8kmn$  шагов блуждание посетит все вершины с вероятностью не меньше  $1 - (\frac{1}{2})^k$ . Если  $(\frac{1}{2})^k$  будет меньше чем единица, делённая на общее число графов, то некоторая последовательность подойдёт сразу ко всем графам. Общее число  $d$ -регулярных графов на  $n$  вершинах меньше  $n^{dn}$ . Поэтому для существования универсальной последовательности достаточно взять  $k = dn \lceil \log n \rceil$ . Таким образом, последовательность будет иметь длину  $p(n) = 8dmn^2 \lceil \log n \rceil$ , то есть полиномиальную, как и было заявлено.  $\square$

## 8.5 О числе затраченных случайных битов

Рассмотрение вероятностных алгоритмов вводит новый вид вычислительного ресурса — число случайных битов. Возникает естественный вопрос: сколько битов реально необходимо для проведения вычислений? Если это число можно сократить до логарифма, сохранив полиномиальное время работы, то **P = BPP**: можно просто перебрать все возможные наборы и посчитать вероятность. Многие исследователи верят, что в конце концов так и будет сделано, но существенные продвижения известны уже сейчас. Мы приведём один пример: для уменьшения ошибки вовсе не обязательно для каждого запуска брать новые случайные биты. Можно хитрым образом переиспользовать старые. Сначала мы подробно разберём подход с попарно независимыми наборами случайных битов, который экономит случайные биты ценой экспоненциального увеличения числа запусков. Затем мы кратко опишем подход при помощи экспандеров, который не требует так много запусков.

### 8.5.1 Использование попарно независимых наборов случайных битов

Идея первого подхода заключается в следующем: ошибка будет уменьшаться при многократном повторении алгоритма, даже если случайные биты независимы между запусками лишь попарно, а не в совокупности. Поэтому сначала нужно научиться генерировать много попарно независимых битов, а потом оценить, как уменьшается ошибка.

**Лемма 8.25.** Пусть  $b_1, \dots, b_q$  — независимые случайные биты, а  $\alpha \in \{0, 1\}^q \setminus \{0^q\}$ . Определим  $x_\alpha$  как  $\bigoplus_{i=1}^q \alpha_i b_i$ . Тогда каждое  $x_\alpha$  распределено равномерно, а при  $\alpha \neq \beta$  величины  $x_\alpha$  и  $x_\beta$  независимы.

Заметим, что  $x_\alpha \oplus x_\beta = x_{\alpha \oplus \beta}$ , поэтому тройки независимыми уже не будут.

*Доказательство.* Поскольку  $\alpha \neq 0^q$ , то для некоторого  $j$  верно  $\alpha_j = 1$ . Тогда для любой фиксированной суммы  $\bigoplus_{i \neq j} \alpha_i b_i$  изменение  $b_j$  меняет итоговое  $x_\alpha$ . Поскольку  $b_j$  не зависит от  $\bigoplus_{i \neq j} \alpha_i b_i$ , получаем, что  $x_\alpha$  равновероятно принимает оба значения.

Далее, при  $\alpha \neq \beta$  найдётся индекс  $j$ , такой что  $\alpha_j \neq \beta_j$ . Без ограничения общности  $\alpha_j = 1$ , а  $\beta_j = 0$ . Тогда изменение  $b_j$  меняет  $x_\alpha$ , но не меняет  $x_\beta$ . Поскольку значение  $b_i$  не зависит от значений остальных сумм, с вероятностью  $\frac{1}{2}$  значение  $x_\alpha$  совпадает с  $x_\beta$ , а с вероятностью  $\frac{1}{2}$  не совпадает. Вместе с равномерностью каждого распределения это означает независимость.  $\square$

## 8.6 Исторические замечания и рекомендации по литературе

Понятие детерминированного алгоритма было известно ещё Евклиду, но вероятностные алгоритмы — значительно более позднее приобретение. Хотя некоторые традиционные практики гадания могут быть интерпретированы как использование рандомизации (см. два забавных примера в очерке Шалита [121]), первым применением случайности для вычислений следует считать приближённое вычисление числа  $\pi$  при помощи иглы Бюффона (см., например, заметку Холла [69]). Первым «алгоритмом Лас-Вегаса» был, по всей видимости, алгоритм Поклингтона решения квадратичного сравнения, открытый в 1917 году [106]. Описание этого и других теоретико-числовых алгоритмов докомпьютерной эпохи приведён в обзоре Уильямса и Шалита [142].

Следующей вехой на пути развития вероятностных методов стало открытие методов Монте-Карло в рамках работы над манхеттенским проектом. Широкой публике о них стало известно прежде всего из работы Метрополиса и Улама [96]. В качестве подробного обзора можно рекомендовать книгу Фишмана [46].

Вероятностные машины Тьюринга впервые были определены и проанализированы в работе де Леува, Мура, Шеннона и Шапиро [90], там же было доказано, что при монетке с вычислимой вероятностью выпадения орла использование вероятности не расширяет класс вычислимых функций. В этой статье использовалась модель машины с дополнительной лентой, полностью заполненной случайными битами.

Систематическое изучение вероятностных сложных классов началось с диссертации Джилла [56] и его же работы [57]. В них были определены классы **RP**, **BPP**, а также чуть в других терминах **RP**, **coRP** и **ZPP**, доказаны теоремы об амплификации и **ZPP** = **RP** ∩ **coRP**. Джилл использовал модель со специальным состоянием, в котором бросается монетка.

Важной задачей, привлекавшей внимание к вероятностным алгоритмам, была задача проверки числа на простоту. В конце 1970-х были открыты тесты Миллера–Рабина [97, 108] и Соловея–Штрассена [129], показавшие, что множество простых чисел лежит в **coRP**. В 1987 году Эйдельман и Хуан доказали [7], что простые числа лежат в **RP**, а тем самым и в **ZPP**. Наконец, в 2002 году задача о простых числах была окончательно классифицирована [8] как лежащая в **P**.

Связь вероятностных классов со схемами из функциональных элементов была продемонстрирована в статье Эйдельмана [6]. Класс **BPP** был включён в полиномиальную иерархию в работе Сипсера [125]. В той же работе было указано, что Питер Гач усилил утверждение, включив **BPP** на второй уровень. Приведённое нами доказательство принадлежит Лаутеману [89].

Стандартным учебником по вероятностным алгоритмам является книга Мотуани и Рагхавана [100]. Обзор последних достижений в задаче о равенстве двух многочленов приведён в статьях Саксены [117] и [118].

## 8.7 Задачи и упражнения

**8.1. Вероятностные схемы.** Определим класс  $\text{BPP}/_{\text{poly}}$  как множество языков  $A$ , для которых существует схема из функциональных элементов  $C$  с двумя группами входов  $x$  и  $r$  со следующим свойством: если  $x \in A$ , то

$\Pr_r [C(x, r) = 1] > \frac{2}{3}$ , а если  $x \notin A$ , то  $\Pr_r [C(x, r) = 1] < \frac{1}{3}$ . Докажите, что  $\mathbf{BPP}/\text{poly} = \mathbf{P}/\text{poly}$ .

**8.2. Приближённое решение 3-КНФ.** Докажите, что случайный набор значений делает истинными не менее  $\frac{7}{8}$  дизъюнктов в 3-КНФ (если в каждой скобке участвуют три различные переменные). Постройте детерминированный алгоритм поиска такого набора.

**8.3. Альтернативное определение ZPP.** Докажите, что  $A \in \mathbf{ZPP}$  тогда и только тогда, когда существует вероятностная машина  $M$ , возвращающая три возможных значения 0, 1 и \*, такая что для любого  $x$  вероятность, что  $M(x) = *$ , не превосходит  $\frac{1}{2}$  и всегда  $M(x) \in \{A(x), *\}$ . (Здесь  $A(x) = 1$  при  $x \in A$  и  $A(x) = 0$  при  $x \notin A$ ).

**8.4. Альтернативный способ амплификации.** Докажите, что для доказательства теоремы 8.13 сработает такой алгоритм: при доле больше  $\tau + \varepsilon$  выдаём 1, при доле меньше  $\tau - \varepsilon$  выдаём 0, иначе случайно.

**8.5. Вероятность и недетерминизм.** Докажите, что если  $\mathbf{NP} \subset \mathbf{BPP}$ , то  $\mathbf{RP} = \mathbf{NP}$ .

**8.6. Ещё вероятность и недетерминизм.** Докажите, что если  $\mathbf{NP} \subset \mathbf{coRP}$ , то  $\mathbf{ZPP} = \mathbf{NP}$ .

**8.7. Случайный оракул.** Докажите, что для случайного оракула  $B$  равенство  $\mathbf{P}^B = \mathbf{BPP}^B$  выполнено с вероятностью 1.

**8.8.** Докажите, что если у ориентированного графа у каждой вершины входящая степень равна исходящей, то для него связность эквивалентна сильной связности.



# Глава 9

## Сложность задач подсчёта

*Ведь сколько на свете хороших друзей,  
Хороших друзей, хороших друзей,  
Сколько на свете весёлых затей,  
Весёлых затей, весёлых затей.*

*Песенка друзей из м/ф «По дороге с  
облаками»*

Недетерминированные и вероятностные классы имели дело с числом сертификатов: в первом случае нужно было узнать, отлично ли это число от нуля, во втором нужно было сравнить их долю с  $\frac{1}{3}$  или другим порогом. И то, и другое легко сделать, если точное количество подходящих сертификатов уже известно. Возникает вопрос, насколько сложно найти это количество в общем случае. Его изучению и посвящена данная глава. Оказывается, что точный подсчёт может оказаться более сложным, чем сравнение результата с каким-то конкретным числом.

### 9.1 Класс $\#P$

**Определение 9.1.** Пусть  $V: \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}$  есть некоторый алгоритм, при этом для слов  $x$  длины  $n$  он использует второй аргумент  $y$  длины  $h(n)$  для некоторой вычислимой функции  $h$ . Тогда задачей подсчёта  $\#V$  называется задача вычисления функции  $\#_V(x) = \#\{y \mid V(x, y) = 1\}$ .

На любую задачу вычисления (всюду определённой) функции можно посмотреть как на задачу подсчёта: вычислить  $f(x)$  есть то же самое, что посчитать количество таких  $y$ , что  $y < f(x)$ . С другой стороны, задача подсчёта есть частный случай задачи вычисления функции, поэтому в общем случае различие между задачами можно не делать. Ситуация меняется при переходе к полиномиальному времени.

**Определение 9.2.** Классом  $FP$  называется класс функций  $f$ , вычислимых за полиномиальное время. Иными словами, некоторый полиномиальный алгоритм на любом входе  $x$  вернёт  $f(x)$ .

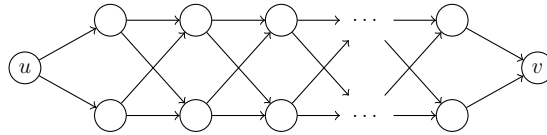
**Определение 9.3.** Классом  $\#P$  (На английском читается “Sharp-P” или “Count-P”) называется класс задач подсчёта, для которых функция длины сертификата  $h$  является полиномом, а алгоритм проверки сертификата  $V$  работает за полиномиальное время.

Таким образом, типичная задача из  $\#P$  — подсчёт числа сертификатов для некоторой задачи из  $NP$ , например числа выполняющих наборов у формулы, числа правильных раскрасок, числа клик данного размера, числа решений задачи целочисленного программирования и т.д. Естественным образом возникают и задачи на базе вероятностных алгоритмов, например подсчёт числа точек, в которых совпадают два полинома. Будет ли задача подсчёта сложнее, чем проверка существования сертификата? Следующий пример показывает, что она может быть сложнее.

**Теорема 9.4.** *Язык  $CYCLE = \{G \mid \text{в ориентированном графе } G \text{ есть ориентированный цикл}\}$  лежит в  $P$ . Однако задача  $\#CYCLE$  подсчёта числа простых (т.е. без повторяющихся вершин) ориентированных циклов в данном графе  $G$  является  $NP$ -трудной. Последнее означает, что существование полиномиального алгоритма для её решения гарантирует  $P = NP$ .*

*Доказательство.* Проверка существования цикла тривиально делается при помощи обхода графа. Например, если обход в глубину из какой-то вершины вышел на  $(n + 1)$ -й уровень, то цикл точно есть, а если цикла нет, то все обходы завершаться не дальше  $n$ -го уровня. Или можно последовательно удалять вершины с нулевой входящей и/или исходящей степенью. Если всё будет удалено, то цикла не будет, если не всё, то цикл будет.

Теперь покажем  $NP$ -трудность задачи  $\#CYCLE$ . Для этого сведём к ней задачу о гамильтоновом цикле в ориентированном графе. Мы предъявим такую сводимость, что в любом графе, построенном по гамильтонову графу, будет больше циклов, чем в любом графе, построенном по негамильтонову графу. Сводимость устроена очень просто: каждое ребро  $(u, v)$  исходного графа заменяется на такую структуру-«конфету»:



Общее количество пар вершин равно  $k$  и будет определено позже. Пока что отметим, что количество способов пройти в новом графе от  $u$  к  $v$  равно  $2^k$ .

Пусть в исходном графе был гамильтонов цикл, т.е. простой цикл из  $n$  вершин. Тогда в новом графе будет цикл из  $n$  конфет. Общее количество простых циклов, которые он породит, равняется  $2^{kn}$ . Это и будет нижней оценкой на число циклов в новом графе.

Если в графе гамильтонова цикла не было, то все циклы были не длиннее  $n - 1$ . Общее количество циклов меньше  $n^{n-1}$ : если цикл разорвать, то получится цепочка без повторяющихся вершин длины не больше  $n - 1$ , а  $n^{n-1}$  — число всех цепочек длины ровно  $n - 1$ . Каждый из них породит не больше  $2^{k(n-1)}$  простых циклов в новом графе, таким образом общее число циклов будет меньше  $n^{n-1} \cdot 2^{k(n-1)}$ . Параметр  $k$  нужно выбрать так, чтобы это число не превысило  $2^{kn}$ . Это легко осуществить: нужно, чтобы  $(n - 1)(k + \log n)$  было меньше  $kn$ . Это будет выполнено при  $k = n \log n$ , что гарантирует полиномиальность сводимости.  $\square$

## 9.2 Полнота в классе $\#P$

Из сказанного выше ясно, что  $FP \subset \#P$ : верификатору достаточно проверить, что его аргумент меньше  $f(x)$ . Вопрос о строгости вложения — аналог



проблемы  $P \stackrel{?}{=} NP$  для задач подсчёта. Как и раньше, важную роль играют полные задачи, к рассмотрению которых мы переходим.

### 9.2.1 Определение и простые примеры

Поскольку мы больше не работаем с задачами распознавания, использовать сводимость по Карпу в общем случае затруднительно: ответ на одну задачу может получиться из ответа на другую довольно замысловатым образом. Поэтому мы будем использовать сводимость по Куку, при которой результат одной задачи используется как оракул. Более формально:

**Определение 9.5.** Пусть  $f$  есть некоторая функция. Тогда  $FP^f$  есть класс функций, которые вычисляются за полиномиальное время с обращениями к  $f$  как к оракулу. (Т.е. значение  $f$  в любой точке можно вычислить за один шаг). Задача  $f$  называется #P-трудной, если  $\#P \subset FP^f$ , и #P-полной, если она #P-трудна и сама лежит в #P.

В тех же терминах удобно сформулировать, что означает NP-трудность задачи #CYCLE.

**Определение 9.6.** Классом  $P^f$  называется класс языков, которые вычисляются за полиномиальное время с оракулом  $f$ . Задача  $f$  называется NP-трудной, если  $NP \subset P^f$ .

**Теорема 9.7.** Задача #CYCLE является NP-трудной.

*Доказательство.* Все ингредиенты у нас уже есть. Во-первых, задача HAMCYCLE является NP-полной, поэтому вначале по экземпляру произвольной NP-задачи за полиномиальное время строится некоторый граф. Затем конструкцией из доказательства 9.4 за полиномиальное время строится новый граф. Наконец, одним запросом к оракулу #CYCLE можно узнать, сколько в новом графе циклов. Если это число не меньше  $2^{kn}$ , то ответ на исходный вопрос — «да», иначе — «нет».  $\square$

Как построение полиномиального алгоритма для NP-полной задачи будет означать, что  $P = NP$ , так и полиномиальный алгоритм для #P-полной задачи приведёт к равенству  $FP = \#P$ . Действительно, если  $f \in FP$ , то  $FP^f = FP$ . Поскольку для #P-полной  $f$  выполнено  $\#P \subset FP^f$ , получаем  $\#P \subset FP$ , а обратное вложение у нас уже есть. Таким образом, мы доказали следующее:

**Утверждение 9.8.** Если #P-полная задача лежит в FP, то  $FP = \#P$ .

Любой NP-полной задаче соответствует #P-полная задача подсчёта числа сертификатов. Для этого достаточно построить не просто сводимость по Карпу, а сводимость по Левину, при которой сертификаты к исходной задаче перейдут в сертификаты к новой взаимно однозначно.<sup>1</sup> Все стандартные сводимости так и строятся, также можно показать, что для любых NP-языков  $A$  и  $B$  можно построить верификаторы  $V_A$  и  $V_B$ , для которых это выполнено.

<sup>1</sup>Сводимость по Левину определяется как сводимость задач поиска: задача  $A$  с верификатором  $V_A$  сводится к задаче  $B$  с верификатором  $V_B$ , если существуют полиномиально вычисляемые функции  $f$ ,  $g$  и  $h$ , такие что  $V_A(x, y) = V_B(f(x), g(x, y))$  и  $V_B(f(x), z) = V_A(x, h(x, z))$ . Таким образом,  $g$  переделывает сертификат принадлежности  $x$  к  $A$  в сертификат принадлежности  $f(x)$  к  $B$  (а несертификат — в несертификат), а  $h$  — наоборот. Здесь мы дополнительно требуем, чтобы соответствие было взаимно однозначным.

Типичной  $\#P$ -полной задачей является задача  $\#SAT$  вычисления числа выполняющих наборов для пропозициональной формулы. Как обычно, удобнее рассматривать вариант со скобками по три литерала  $\#3SAT$ . Разберём чуть подробнее, почему для неё имеет место сводимость по Левину.

**Теорема 9.9.** *Задача  $\#3SAT$  является  $\#P$ -полной.*

*Доказательство.* Вначале докажем  $\#P$ -полноту задачи  $\#SAT$ . Проанализируем конструкцию из доказательства теоремы 3.11. Если биты, соответствующие сертификату, фиксировались, то вся остальная таблица заполнялась автоматически. А значит, и выполняющий набор для формулы строился однозначно. Таким образом, количество сертификатов для исходного входа и количество выполняющих наборов для полученной формулы совпадают, что и требовалось. Осталось заметить, что сводимость из теоремы 3.9 также не меняет число выполняющих наборов: любой набор однозначно определяет значение любой подформулы.  $\square$

### 9.2.2 Класс $RP$

*Один из членов кнессета, известный своей эмоциональностью, однажды закончил выступление словами: «Чего тут вообще можно добиться, когда половина членов кнессета — идиоты!» В прессе поднялась буря возмущения, и на следующий день он смягчился: «Приношу извинения за свою бестактность, на самом деле половина членов кнессета — не идиоты».*

*Еврейский анекдот*

Ясно, что из  $\#P = FP$  следует  $P = NP$ , а также  $P = BPP$ : если число сертификатов подсчитано, то можно потом сравнить его с любым порогом. Однако в обратную сторону импликация вовсе не обязательна: как мы уже видели, задача подсчёта может быть сложнее задачи распознавания. С другой стороны, если  $P = PSPACE$ , то  $\#P = FP$ : на полиномиальной памяти можно перебрать все сертификаты и посчитать число нужных. Но здесь не будет обратной импликации: на полиномиальной памяти можно сделать гораздо больше. Есть ли класс, равенство которого классу  $P$  будет эквивалентно  $\#P = FP$ ? Оказывается, есть! Это класс  $RP$ . Напомним его определение:

**Определение 9.10.** Язык  $A$  лежит в  $RP$ , если существует такой верификатор  $V$ , что  $x \in A$  эквивалентно тому, что  $\Pr_y [V(x, y) = 1] \geq \frac{1}{2}$ .

Это определение очень устойчиво к деталям. Во-первых, знак  $\geq$  можно заменить на  $>$  и даже сделать определение симметричным:  $\Pr_y [V(x, y) = 1] > \frac{1}{2}$  при  $x \in A$  и  $\Pr_y [V(x, y) = 1] < \frac{1}{2}$  при  $x \notin A$ . Во-вторых,  $\frac{1}{2}$  можно заменить на любой другой порог  $\alpha$ , возможно даже зависящий от  $x$ , если только этот порог можно вычислить с точностью  $2^{-|y|}$  за полиномиальное время.<sup>2</sup> Ключевой будет следующая лемма.

**Лемма 9.11.** Пусть  $V$  — полиномиальный верификатор с длиной сертификата  $p(n)$ . Пусть функция  $t$  вычислима за полиномиальное время и принимает

<sup>2</sup>Правда, использовать мы будем лишь константные пороги.

целые значения от 0 до  $2^{p(n)}$ . Тогда существует другой полиномиальный верификатор  $W$  с длиной сертификата  $q(n)$ , такой что если  $\#_V(x) \geq m(x)$ , то  $\#_W(x) > \frac{1}{2}2^{q(n)}$ , а если  $\#_V(x) < m(x)$ , то  $\#_W(x) < \frac{1}{2}2^{q(n)}$ .

*Доказательство.* Доказательство состоит из двух частей: сначала мы сместим порог с  $m(x)$  на половину, а потом сделаем оба неравенства строгими.

Построим новый верификатор  $W_m$ , который принимает на вход  $x$  и сертификат  $z$  длины  $p(n) + 1$ . Если  $z$  имеет вид  $0v$ , то  $W_m$  возвращает  $V(x, v)$ . Если же  $z$  имеет вид  $1v$ , то  $W_m$  возвращает 1, если  $v$  не меньше  $m(x)$ . Поскольку  $V$  и  $m$  вычислимы за полиномиальное время, то же верно и для  $W_m$ . Если  $V(x, \cdot)$  принимал  $q$  сертификатов, то  $W_m(x, \cdot)$  принимает  $q + 2^{p(n)} - m(x)$  сертификатов. Это не меньше  $\frac{1}{2}$  от  $2^{p(n)+1}$ , т.е. не меньше  $2^{p(n)}$ , тогда и только тогда, когда  $q \geq m(x)$ . Таким образом, сравнение доли сертификатов для  $W_m$  с  $\frac{1}{2}$  позволяет сравнить число сертификатов для  $V$  с любым порогом.

Осталось описать, как добиться строгого неравенства. Идея заключается в том, чтобы чуть-чуть сдвинуть порог, так чтобы в него уже нельзя было точно попасть. Для этого нужно вначале добавить фиктивный бит, от которого ничего не зависит. Тогда подходящих сертификатов станет либо  $\geq 2m(x)$ , либо  $\leq 2m(x) - 2$ , т.е. либо  $> 2m(x) - 1$ , либо  $< 2m(x) - 1$ . После этого нужно применить ту же конструкцию, что и раньше.  $\square$

Теперь будем использовать эту лемму. Непосредственным её применением для  $m(x) = \frac{1}{2}2^{p(n)}$  получаем, что определение может быть сделано симметричным: в любом случае вероятность принятия  $x$  будет отлична от  $\frac{1}{2}$ . Отсюда заметим важное

**Следствие 9.12.** *Класс  $\mathbf{RP}$  замкнут относительно дополнения.*

*Доказательство.* После того, как определение стало симметричным, достаточно обратить результат верификатора.  $\square$

Далее покажем, что условие  $\mathbf{P} = \mathbf{RP}$  и правда сильнее, чем  $\mathbf{P} = \mathbf{NP}$ . Это следует из следующего утверждения:

**Утверждение 9.13.**  $\mathbf{NP} \subset \mathbf{RP}$ .

*Доказательство.* Достаточно применить лемму для  $m(x) = 1$ .  $\square$

Наконец, докажем основную теорему.

**Теорема 9.14.**  $\#P = \mathbf{FP}$  тогда и только тогда, когда  $\mathbf{P} = \mathbf{RP}$ .

*Доказательство.* В одну сторону утверждение очевидно: если  $\#P = \mathbf{FP}$ , то долю подходящих  $y$  можно точно подсчитать и затем сравнить с  $\frac{1}{2}$ .

В другую сторону рассуждение основано на идее двоичного поиска: при помощи леммы и предположения  $\mathbf{RP} = \mathbf{P}$  можно полиномиальным алгоритмом сравнить долю сертификатов сначала с  $\frac{1}{2}$ , затем в зависимости от результата с  $\frac{1}{4}$  или  $\frac{3}{4}$  и т.д. Глубина дерева поиска равна  $p(n)$ , а в конце количество сертификатов будет точно известно. Таким образом, полиномиальность сохранится, и мы доказали  $\#P = \mathbf{FP}$ .  $\square$

В классе  $\mathbf{RP}$  определена полнота относительно стандартной полиномиальной сводимости. Легко понять, что классической  $\mathbf{RP}$ -полной задачей будет задача  $\mathbf{MAJSAT} = \{\varphi \mid \text{для формулы } \varphi \text{ хотя бы половина наборов — выполняющие}\}.$

Заметим напоследок, что на класс **РР** удобно смотреть, введя специальный квантор  $\text{Maj } y$  — «для большинства значений  $y$ ». Если определением **НР**-языка была эквивалентность  $x \in A \Leftrightarrow \exists y V(x, y)$ , а определением **coNP**-языка — эквивалентность  $x \in A \Leftrightarrow \forall y V(x, y)$ , то определением **РР**-языку служит эквивалентность  $x \in A \Leftrightarrow \text{Maj } y V(x, y)$ .

### 9.2.3 Задача о подсчёте перманента

Важнейшую роль в линейной алгебре играет понятие определителя, или детерминанта. Напомним, что  $\det A = \sum_{\sigma \in S_n} (-1)^\sigma \prod_{i=1}^n A_{i\sigma(i)}$ , где  $A$  — матрица размера  $n \times n$ ,  $S_n$  — множество всех перестановок на  $n$  элементах, а  $(-1)^\sigma$  — чётность перестановки. Предсказуемое изменение определителя при элементарных преобразованиях позволяет вычислить его за полиномиальное время через приведение к ступенчатому виду. Если избавиться от множителей  $(-1)^\sigma$ , то получится другая характеристика матрицы — перманент. Мы покажем, что эта характеристика тоже осмысленная (а на самом деле очень важная для теории сложности), но её вычисление является  $\#\mathbf{P}$ -полной задачей.

Вначале нужно пояснить, какая тут вообще связь с  $\#\mathbf{P}$ : количеству каких сертификатов равен перманент. Наиболее прозрачен ответ для матрицы из нулей и единиц. Любая такая матрица задаёт некоторый двудольный граф с долями по  $n$  вершин, где ребро между  $i$ -й вершиной левой доли и  $j$ -й вершиной правой доли проводится, если  $A_{ij} = 1$ . (Такая матрица называется матрицей Татта). В таком случае перманент  $A$  равен количеству совершенных паросочетаний, т.е. наборов из  $n$  непересекающихся по вершинам рёбер. Действительно, если  $\prod_{i=1}^n A_{i\sigma(i)} = 1$ , то для каждого  $i$  проведено ребро из  $i$ -й вершины слева в  $\sigma(i)$ -ю вершину справа. Поскольку  $\sigma$  — перестановка, эти рёбра не пересекаются. С другой стороны, каждому набору из  $n$  непересекающихся рёбер соответствует ненулевое слагаемое в перманенте. Если в матрице есть целые числа больше единицы, то перманент будет равен числу совершенных паросочетаний в двудольном графе с рёбрами соответствующих кратностей. Если же есть и отрицательные числа, то интерпретировать задачу как чистую задачу подсчёта уже не получится, хотя бы потому, что ответ может быть отрицательным. Можно сказать, что каждое ребро имеет вес, возможно, отрицательный, вес паросочетания есть произведение весов входящих в него рёбер, а ответ в задаче — сумма весов всех паросочетаний. Позже мы покажем, что задачу с отрицательными весами можно свести к задаче с неотрицательными.

Стоит заметить, что соответствующая задача *распознавания*: существует ли в графе совершенное паросочетание — решается за полиномиальное время. Например, можно применить алгоритм Форда–Фалкерсона поиска максимального потока в графе.<sup>3</sup> Для этого нужно добавить источник, от которого идут рёбра во все вершины левой доли, и сток, в который идут рёбра из всех вершин правой доли. Все рёбра между долями нужно ориентировать слева направо, а веса всех рёбер взять единичными. Тогда совершенное паросочетание существует тогда и только тогда, когда по сети можно пропустить поток размера  $n$ . Таким образом, доказательство  $\#\mathbf{P}$ -полноты даст ещё один пример, когда задача подсчёта сложнее задачи распознавания.

Есть и ещё одна интерпретация задачи о вычислении перманента как задачи подсчёта. На этот раз посмотрим на матрицу  $A$  (вновь из нулей и единиц) как на матрицу смежности ориентированного графа (с возможными петлями). Как из-

<sup>3</sup>Существует также специализированный алгоритм Хопкрофта–Карпа и ряд других.

вестно, любая перестановка раскладывается в произведение циклов. Нетрудно заметить, что  $\prod_{i=1}^n A_{i\sigma(i)} = 1$  тогда и только тогда, когда все эти циклы реально являются циклами в графе. Таким образом, каждое единичное слагаемое в перманенте соответствует покрытию ориентированного графа непересекающимися циклами, а задача вычисления перманента соответствует задаче подсчёта числа таких покрытий. Если в матрице стоят не только нули и единицы, то все рёбра и все покрытия вновь становятся взвешенными, и требуется посчитать их суммарный вес.

Заметим, что соответствующая задача распознавания: можно ли покрыть граф циклами — также решается за полиномиальное время. Более того, она легко сводится к задаче о поиске совершенного паросочетания. Для этого нужно продублировать каждую вершину графа и заменить каждое направленное ребро  $(u, v)$  на ненаправленное  $(u, v')$  ( $v'$  — копия  $v$ ). Легко увидеть, что покрытия циклами в исходном графе и совершенные паросочетания в новом взаимно однозначно соответствуют друг другу.

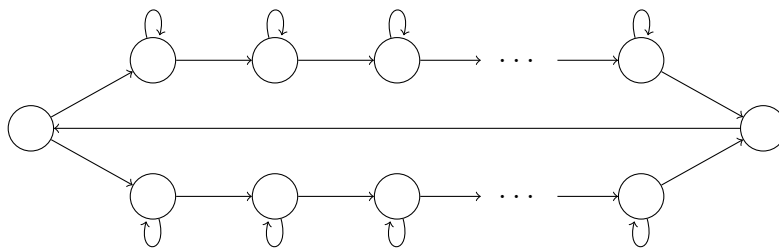
Итак, мы доказываем следующую теорему:

**Теорема 9.15** (Теорема Вэлианта, [138]). *Задача вычисления перманента для матриц из нулей и единиц #P-полна.*

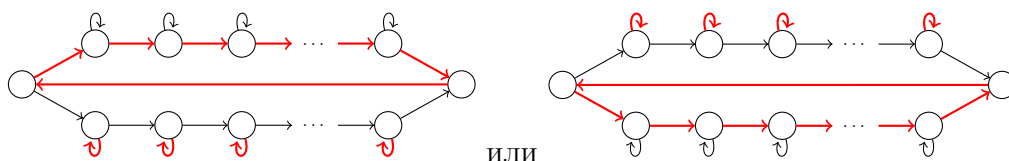
Есть несколько способов доказать эту теорему. Все они используют довольно сложные гаджеты, в которых веса рёбер расставлены некоторым специальным образом, чтобы почти всё сокращалось. Мы будем сводить #3SAT к задаче о подсчёте суммарного веса покрытий циклами, а затем покажем, как избавиться от всех весов, кроме нуля и единицы.

*Доказательство.* Мы построим взвешенный граф по формуле  $\varphi$ , записанной в 3-КНФ, так чтобы каждому выполняющему набору соответствовали покрытия циклами суммарного фиксированного положительного веса, а каждому невыполняющему — нулевого. Как обычно, нужны гаджеты для каждой переменной, каждого дизъюнкта и связей между ними. Всюду мы будем использовать следующие соглашения: если вес ребра нулевой, то оно не проводится, если единичный, то проводится, но вес не подписывается, иначе ребро проводится, а вес подписывается.

Начнём с гаджета для переменной:

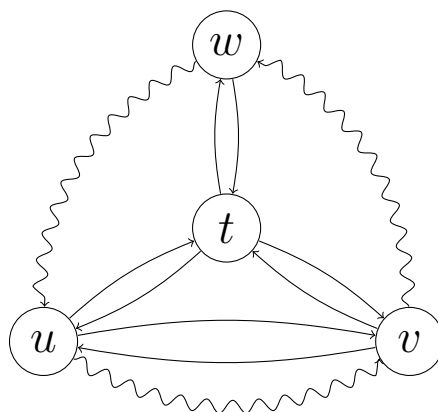


Количество промежуточных вершин мы укажем позже. Ясно, что этот гаджет можно покрыть циклами двумя способами:



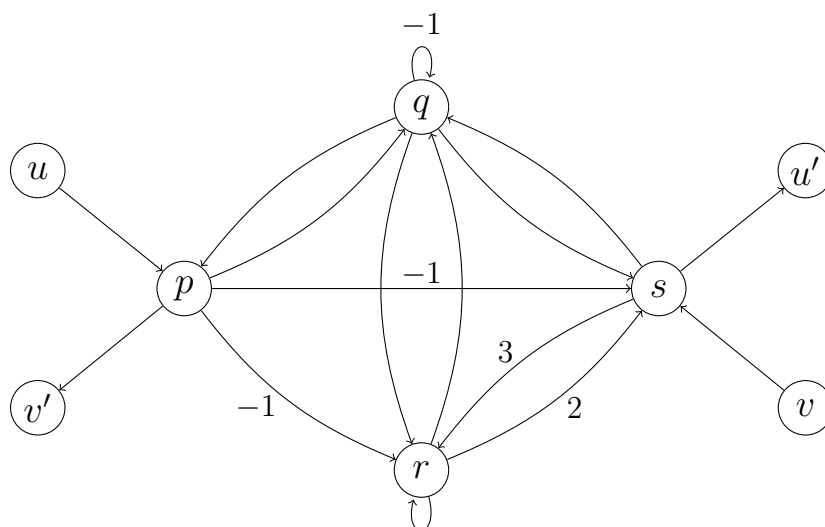
Левый из них будет соответствовать истинной переменной, а правый — ложной.

Теперь опишем гаджет для дизъюнкта. Волнистыми линиями показаны рёбра, которые впоследствии будут переделаны в более сложные конструкции. (В частности, кратного ребра в итоге не будет).



Идея гаджета заключается в следующем: все три волнистых ребра быть в циклах покрытия не могут, иначе вершина  $t$  останется не покрытой. Все остальные наборы волнистых рёбер можно дополнить до покрытия циклами ровно одним способом. Например, если нет ни одного из волнистых рёбер, то единственный вариант — это два цикла  $uv$  и  $tw$ , если есть только ребро  $(v, w)$ , то оно продолжается до цикла  $vwtu$ , два ребра  $(v, w)$  и  $(w, u)$  продолжают до цикла  $vwtu$ , и т.д.

Теперь понятно, как использовать эту идею: нужно, каждому литералу, входящему в дизъюнкт, должно соответствовать одно из волнистых рёбер. При этом если литерал истинен, то соответствующего ребра заведомо не должно быть в покрытии циклами. А истинность литерала определяется способом покрытия гаджета для соответствующей переменной: конкретно мы будем смотреть на петли. Отсюда становится ясной потребность в XOR-гаджете, т.е. некоторой структуре, которая гарантирует наличие в покрытии циклами только одного ребра из трёх. Выглядит этот гаджет так:



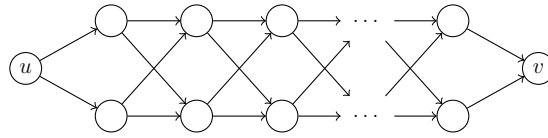
Точное свойство этого гаджета таково: если в некотором графе из рёбер  $uu'$  и  $vv'$  проведено ровно одно, а других рёбер между этими вершинами заведомо нет, то после присоединения XOR-гаджета к этим вершинам вместо этого ребра суммарный вес всех циклов домножится на 4, если же не было ни одного, или были оба, то суммарный вес обнулится. Аккуратно проверим это свойство:

- Пусть не было ни одного из двух рёбер. Тогда все циклы сохранятся, и нужно только дополнительно покрыть вершины  $p, q, r, s$ . Это можно сделать такими наборами циклов:  $(pq, rs)$ ,  $(prsq)$ ,  $(psrq)$ ,  $(psq, r)$ . Их веса составляют, соответственно, 6,  $-2$ ,  $-3$  и  $-1$ , что в сумме даёт ноль, что и требовалось. Заметим, что т.к. рёбра  $uv'$  и  $vu'$  в графе заведомо отсутствуют, то участки  $upv'$  и  $vsu'$  в циклах встретиться не могут.
- Пусть было только ребро  $uu'$ . Тогда в новом графе должен быть путь от  $u$  до  $u'$  и, возможно, какие-то ещё циклы. Для этого есть такие варианты:  $(upsu', q, r)$ ,  $(upsu', qr)$ ,  $(upqsu', r)$ ,  $(uprsu', q)$ ,  $(upqrsu')$  и  $(uprqsu')$ . Их веса равны, соответственно, 1,  $-1$ , 1, 2, 2 и  $-1$ , в сумме 4, как и было заявлено.
- Пусть было только ребро  $vv'$ . Тогда будет всего два варианта  $(vsqpv', r)$  и  $(vsrqpv')$ , суммарный вес вновь 4.
- Пусть были оба ребра. Поскольку одновременно путей из  $u$  в  $u'$  и из  $v$  в  $v'$  быть не может, этот вариант обнулится.

Теперь опишем, как собрать всё вместе. Каждое из волнистых рёбер гаджета дизъюнкта соответствует одному из входящих в него литералов. Если это литерал  $p_i$ , то соединим это ребро при помощи XOR-гаджета с одним из рёбер верхнего пути гаджета  $i$ -й переменной. Если же это литерал  $\bar{p}_i$ , то соединим ребро с одним из рёбер нижнего пути. Каждый раз нужно выбирать новые рёбра, поэтому для каждой переменной нужен гаджет длиной в количество её вхождений.

Проанализируем конструкцию. Каждый набор истинностных значений задаёт одно из двух разбиений на циклы каждого гаджета переменной. Если набор выполняющий, то XOR-гаджеты обеспечивают отсутствие хотя бы одного волнистого ребра в каждом гаджете дизъюнкта, после чего они однозначно покрываются циклами. При этом каждый XOR-гаджет умножает суммарный вес циклов на 4, так что общий вес составит  $4^{3m}$ . Если же набор невыполняющий, то XOR-гаджеты оставят все три волнистых ребра хотя бы в одном гаджете дизъюнкта, поэтому вес обнулится. Значит, итоговый перманент будет равен  $4^{3m} \cdot \#\varphi$ , что и обеспечивает сводимость в #P.

Осталось пояснить, как оставить только нулевые и единичные веса. Во-первых, ребро весом  $-w$  можно заменить на два последовательных веса  $-1$  и  $w$ . Во-вторых, уже знакомая нам конфета



позволяет смоделировать ребро веса  $2^k$  при помощи рёбер веса 1. В-третьих, при проведении параллельных рёбер веса складываются, что позволяет смоделировать и любые другие положительные веса (чтобы не возникало кратных рёбер, можно поставить дополнительные вершины):



Таким образом, у нас остались только веса 0, 1 и  $-1$ , и теперь нужно избавиться от последних. Если у нас к этому моменту оказалось  $n$  вершин, то перманент заведомо лежит в пределах от  $-n!$  до  $n!$ . Это значит, что достаточно посчитать перманент по модулю  $s$ , если только  $s > 2n!$ . А значит, можно и заменять веса по такому модулю, т.е. все веса  $-1$  заменить на  $s - 1$ . Для удобства можно взять подходящую степень двойки, например  $s = 2^{n \lceil \log n \rceil} + 1$ , и после замены этих рёбер на конфеты длины  $n \lceil \log n \rceil$  останутся лишь веса 0 и 1, что и требовалось.  $\square$

## 9.3 Языки с единственным сертификатом

*Подруга дней моих суровых,  
Голубка дряхлая моя!  
Одна в глуши лесов сосновых  
Давно, давно ты ждёшь меня.*

А.С.Пушкин, *Няне*

Пусть заранее известно, что у формулы может быть не более одного выполняющего набора. Может ли это условие помочь отличить выполнимую формулу от невыполнимой? Оказывается, вряд ли: если бы могло, то **NP** равнялось бы **RP**. Формально мы имеем задачу с предусловием (см. раздел 8.3.6)  $\text{USAT}$ , для которой  $\text{USAT}_y = \{\varphi \mid \text{у формулы } \varphi \text{ ровно один выполняющий набор}\}$ , а  $\text{USAT}_n = \{\varphi \mid \text{формула } \varphi \text{ невыполнима}\}$ .

### 9.3.1 Семейства попарно независимых хеш-функций

Нам потребуется такой инструмент, как семейства попарно независимых хеш-функций. Вообще, понятие хеш-функции используется в самых разных разделах теории и практики программирования. Неформально говоря, хеширование означает быстрое вычисление некоторого короткого значения — *хеша*, такого что трудно найти *коллизия*, т.е. два исходных слова, у которых одинаковый хеш. Это понимание может уточняться самыми разными способами, нам потребуется такой:

**Определение 9.16.** Множество  $\mathcal{H}_{n,k}$  функций из  $\{0,1\}^n$  в  $\{0,1\}^k$  называется семейством попарно независимых хеш-функций, если для любых  $x \neq x'$  из  $\{0,1\}^n$  и  $y, y'$  из  $\{0,1\}^k$  выполнено

$$\Pr_{h \in \mathcal{H}_{n,k}} [h(x) = y \wedge h(x') = y'] = \frac{1}{2^{2k}}.$$

Идея заключается в том, что вероятность коллизии (и вообще любой конкретной пары значений на разных входах) для случайной функции из семейства такая же, как для случайной функции вообще. Поэтому интерес представляет случай, когда семейство имеет экспоненциальный размер вместо двойной экспоненты множества всех функций. Тогда номер функции имеет полиномиальную длину, и можно надеяться на полиномиальную вычислимость. Такие семейства и правда можно построить:

**Теорема 9.17.** Для любых  $n$  и  $k$  существует полиномиально вычислимое попарно независимое семейство хеш-функций размера  $2^{k(n+1)}$ .



*Доказательство.* Хеш-функции из нашего семейства будут задаваться словами из нулей и единиц длины  $k(n+1)$ , которые мы будем интерпретировать как матрицу  $A$  размера  $k \times n$  и вектор-столбец  $b$  длины  $k$  над полем  $\mathbb{F}_2$ . Функция, заданная такими параметрами, будет работать как линейное преобразование над этим полем:  $h_{A,b}(x) = Ax + b$ . Размер семейства и полиномиальная вычислимость очевидны из построения, нужно доказать свойство попарной независимости.

Нам пригодится простое, но важное соображение:

**Лемма 9.18** (Принцип суммирования по случайному подмножеству). *Пусть  $x \in \{0,1\}^n$ ,  $x \neq 0$ . Тогда доля таких  $a$ , что  $\langle a, x \rangle = 1$ , равняется  $\frac{1}{2}$ . (Здесь  $\langle \cdot, \cdot \rangle$  — скалярное произведение по модулю 2).*

*Доказательство.* Выражение  $\langle a, x \rangle$  можно интерпретировать как сумму координат  $x$ , для которых  $a_i = 1$ . Если  $x \neq 0$ , то, например,  $x_j = 1$ . Перемена  $a_j$  повлияет на вхождение  $x_j$  в сумму и тем самым изменит результат. Таким образом, все вектора  $a$  разобьются на пары, в каждой из которых один вектор даёт сумму 1, а другой 0. Значит, ровно половина векторов даст сумму 1.  $\square$

Сначала выясним, для какой доли  $A$  может быть выполнено условие  $Ax = y$ . Оно распадается на  $k$  отдельных условий  $\langle a_i, x \rangle = y_i$ . Если  $x = 0$ , то при  $y = 0$  подойдут любые  $A$ , а при  $y \neq 0$  не подойдёт ни одна. Если же  $x \neq 0$ , то каждое из  $k$  условий выполнено с вероятностью  $\frac{1}{2}$  независимо от других. Поэтому все одновременно выполнены с вероятностью  $\frac{1}{2^k}$ .

Добавление  $b$  позволяет сделать вероятность равной  $\frac{1}{2^k}$  независимо от значений  $x$  и  $y$ . Действительно, при  $x \neq 0$  рассуждение не изменится, а при  $x = 0$  подойдёт ровно одно  $b = y$  из  $2^k$  вариантов, а  $A$  по-прежнему любая.

Теперь надо проверить, что равенства  $Ax + b = y$  и  $Ax' + b = y'$  выполняются независимо друг от друга при  $x \neq x'$ . Если одно из  $x$  и  $x'$  равно нулю, то одно условие задаёт только  $b$ , а другое — только  $A$ , поэтому они независимы. Если же оба вектора ненулевые, то найдутся такие координаты  $j$  и  $l$ , что  $x_j = 1$ ,  $x_l = 0$  и  $x'_l = 1$  (при необходимости нужно поменять  $x$  и  $x'$  местами). В таком случае 4 варианта значений  $a_{ij}$  и  $a_{il}$  зададут 4 разных варианта скалярных произведений с  $x$  и  $x'$ , из которых только один подойдёт под оба условия. Во всех строках сразу это произойдёт с вероятностью  $\frac{1}{4^k}$ , что и означает попарную независимость.  $\square$

Заметим, что хотя обычно у хеш-функции длина хеша значительно меньше длины исходного слова, здесь мы таких условий не накладывали. Более того, в наших приложениях  $k$  может быть и слегка больше  $n$ .

### 9.3.2 Теорема Вэлианта–Вазирани

*Ты у меня одна,  
Словно в ночи луна,  
Словно в степи сосна  
Словно в году весна.*

Ю.Визбор

**Теорема 9.19** (Вэлианта–Вазирани, [139]). *Если  $\text{USAT} \in \text{Promise-P}$ , то  $\text{NP} = \text{RP}$ .*

*Доказательство.* Мы уже знаем, что  $\mathbf{RP} \subset \mathbf{NP}$ . Значит, для доказательства равенства достаточно построить вероятностный алгоритм с односторонней ошибкой для задачи о выполнимости. Мы построим специальную рандомизированную сводимость, которая с существенной вероятностью оставит ровно один выполняющий набор. Нам понадобится такая лемма:

**Лемма 9.20.** Пусть  $S \subset \{0,1\}^n$ , а  $k$  выбрано так, что  $2^{k-2} \leq |S| \leq 2^{k-1}$ . Пусть  $\mathcal{H}_{n,k}$  — семейство попарно независимых хеш-функций. Тогда

$$\Pr_{h \in \mathcal{H}_{n,k}} [\exists! x \in S \ h(x) = 0] \geq \frac{1}{8}$$

*Доказательство.* Обозначим через  $E_x$  событие « $h(x) = 0$ ». Из свойства попарной независимости следует, что для любого  $x$  выполнено  $\Pr_h[E_x] = \frac{1}{2^k}$ , а для  $x \neq x'$  верно  $\Pr_h[E_x \cap E_{x'}] = \frac{1}{2^{2k}}$ . Обозначим через  $p$  величину  $\frac{|S|}{2^k}$ . По предположению она лежит на  $[\frac{1}{4}, \frac{1}{2}]$ . Из линейности мат. ожидания она равна среднему числу элементов  $x \in S$ , для которых случилось  $E_x$ . Нам же нужно оценить вероятность того, что такой элемент будет ровно один. Обозначим количество таких элементов (случайную величину) за  $N$ .

Во-первых, оценим величину  $\Pr_h[N \geq 1] = \Pr_h[\cup_{x \in S} E_x]$  по формуле включений-исключений:

$$\Pr_h[\cup_{x \in S} E_x] \geq \sum_{x \in S} \Pr_h[E_x] - \sum_{x, x' \in S, x \neq x'} \Pr_h[E_x \cap E_{x'}] > \frac{|S|}{2^k} - \frac{|S|^2}{2 \cdot 2^{2k}} = p - \frac{p^2}{2}.$$

Во-вторых, оценим сверху величину  $\Pr_h[N \geq 2] = \Pr_h[\cup_{x, x' \in S, x \neq x'} (E_x \cap E_{x'})]$ . Здесь годится самая простая оценка через сумму вероятностей:

$$\Pr_h[N \geq 2] \leq \sum_{x, x' \in S, x \neq x'} \Pr_h[E_x \cap E_{x'}] < \frac{|S|^2}{2 \cdot 2^{2k}} = \frac{p^2}{2}.$$

Соединив две оценки, получаем  $\Pr_h[N = 1] = \Pr_h[N \geq 1] - \Pr_h[N \geq 2] > (p - \frac{p^2}{2}) - \frac{p^2}{2} = p - p^2$ . Поскольку  $p \in [\frac{1}{4}, \frac{1}{2}]$ , имеем  $p - p^2 > \frac{1}{8}$ , что и даёт заявленную оценку.  $\square$

Теперь аккуратно опишем, какую сводимость мы будем использовать. Мы построим полиномиально вычислимую случайную функцию  $f$ , такую что:

- Если  $\varphi$  выполнима, то  $f(\varphi) \in \mathbf{USAT}_y$  с вероятностью не меньше  $\frac{1}{8n}$ ;
- Если  $\varphi$  невыполнима, то и  $f(\varphi)$  невыполнима.

После этого легко доказать  $\mathbf{SAT} \in \mathbf{RP}$  при  $\mathbf{USAT} \in \mathbf{Promise-P}$ : по формуле  $\varphi$  вычислим  $f(\varphi)$ , запустим на ней алгоритм для  $\mathbf{USAT}$  и выдадим результат в качестве ответа. Если  $\varphi$  была выполнима, то  $f(\varphi)$  с вероятностью не меньше  $\frac{1}{8n}$  будет формулой с единственным выполняющим набором, и алгоритм для  $\mathbf{USAT}$  выдаст 1. Возможно, алгоритм для  $\mathbf{USAT}$  выдаст 1 и в другом случае, так что вероятность точно будет не меньше  $\frac{1}{8n}$ . Если же  $\varphi$  не была выполнима, то  $f(\varphi)$  точно не будет выполнимой, и алгоритм для  $\mathbf{USAT}$  точно выдаст 0. Далее стандартной амплификацией  $\frac{1}{8n}$  можно довести до  $\frac{1}{2}$ .

Осталось построить саму сводимость  $f$ . Она будет устроена так: сначала выбирается равномерно случайное  $k$  от 2 до  $n+1$ . Затем выбирается случайная  $h \in \mathcal{H}_{n,k}$  и строится формула, означающая  $f(\varphi) = \varphi(x) \wedge h(x) = 0^k$ . Заметим, что наше семейство  $h$  специально построено таким образом, чтобы равенство

$h(x) = 0^k$  легко записывалось булевой формулой. Покажем, что формула  $f(\varphi)$  подходит под условие. Действительно, если  $\varphi$  невыполнима, то и  $f(\varphi)$  невыполнима как конъюнкция  $\varphi$  и дополнительного условия. Если же  $\varphi$  выполнима, то с вероятностью  $\frac{1}{n}$  параметр  $k$  будет выбран так, что число выполняющих наборов  $\varphi$  будет лежать между  $2^{k-2}$  и  $2^{k-1}$ . В этом случае с условной вероятностью не меньше  $\frac{1}{8}$  условие  $h(x) = 0^k$  оставит ровно один выполняющий набор, а полная вероятность такого события будет не меньше  $\frac{1}{8n}$ , что и требовалось.  $\square$

## 9.4 Класс $\oplus P$

На задачи из класса  $PP$  можно посмотреть и под таким углом: требуется узнать самый старший бит числа подходящих сертификатов. А что, если спросить самый младший бит, т.е. чётность? Тогда тоже получится интересный класс под названием  $\oplus P$  (на английском читается как “Parity- $P$ ”).

**Определение 9.21.** Класс  $\oplus P$  есть множество языков  $A$ , для которых существует полиномиальный верификатор  $V$ , такой что  $x \in A$  эквивалентно нечётности числа  $y$ , таких что  $V(x, y) = 1$ .

Видно, что определение находится в русле определений  $NP$ ,  $coNP$  и  $PP$ , но вместо кванторов  $\exists y$ ,  $\forall y$  и  $\text{Maj } y$  вводится новый квантор  $\oplus y$  («для нечётного числа  $y$ »), формально определяемый как  $\oplus y V(x, y) = \sum_y V(x, y) \bmod 2$ .

Неудивительно, что сводимости, сохраняющие число сертификатов, приводят к  $\oplus P$ -полным языкам. Так, типичной  $\oplus P$ -полной задачей будет задача  $\oplus SAT = \{\varphi \mid \text{у формулы } \varphi \text{ нечётное число выполняющих наборов}\}$ .

Одно из существенных отличий  $\oplus SAT$  от  $SAT$  состоит вот в чём: если бы  $SAT$  сводилась к  $\overline{SAT}$ , то  $NP$  равнялось бы  $coNP$ . Поэтому такой сводимости мы не знаем. А вот для  $\oplus SAT$  такая сводимость легко строится. Этому помогает арифметическая природа  $\oplus SAT$ : логические операции нужно превратить в операции над числами соответствующей чётности. А именно, верны следующие соотношения:

- Утверждение  $\varphi \in \oplus SAT \wedge \psi \in \oplus SAT$  эквивалентно  $(\varphi \wedge \psi) \in \oplus SAT$ , где переменные у формул  $\varphi$  и  $\psi$  переименованы так, чтобы не пересекаться. Действительно, число выполняющих наборов  $\varphi \wedge \psi$  равно произведению чисел выполняющих наборов  $\varphi$  и  $\psi$ , а произведение нечётно тогда и только тогда, когда оба сомножителя нечётны. Как обычно, вместо конъюнкции можно записывать произведение:  $\varphi \cdot \psi$ . (Заметим, что такая эквивалентность могла бы быть сделана и для  $SAT$ ).
- Если  $\varphi$  есть функция от  $n$  аргументов, то  $\varphi \notin \oplus SAT$  тогда и только тогда, когда следующая формула  $\psi$  от  $(n+1)$  аргумента лежит в  $\oplus SAT$ :  $\psi(z, x) = (\neg z \wedge \varphi(x)) \vee (z \wedge \bigwedge_{i=1}^n x_i)$ . Действительно, в выполняющем наборе  $\psi$  либо  $z = 0$ , а  $x$  является выполняющим набором  $\varphi$ , либо все переменные равны единице. Таким образом, у  $\psi$  на один выполняющий набор больше, так что чётность сменится. Естественным образом такая  $\psi$  обозначается как  $\varphi + 1$ .
- Похожим образом по двум формулам  $\varphi$  и  $\varphi'$  можно построить формулу, число выполняющих наборов у которой будет суммой соответствующих чисел у  $\varphi$  и  $\varphi'$ :  $\psi(z, x) = (\neg z \wedge \varphi(x)) \vee (z \wedge \varphi'(x))$ . Здесь мы предположили, что две формулы зависят от одних и тех же переменных. Если это не

так, нужно недостающие добавить конъюнкцией. Естественным образом такая  $\psi$  обозначается как  $\varphi + \varphi'$ . Видно, что предыдущее рассуждение был частным случаем, когда  $\varphi'$  была тождественной истиной. Также видно, что  $\psi \in \oplus\text{SAT}$  тогда и только тогда, когда ровно одна из формул  $\varphi$ ,  $\varphi'$  лежит в  $\oplus\text{SAT}$ .

- Наконец, из всего описанного можно выразить и дизъюнкцию. А именно,  $\varphi \in \oplus\text{SAT}$  или  $\varphi' \in \oplus\text{SAT}$  тогда и только тогда, когда  $(\varphi + 1)(\varphi' + 1) + 1$  лежит в  $\oplus\text{SAT}$ .

Естественным образом из описанных конструкций получается такое

**Утверждение 9.22.** Если  $A$  и  $B$  лежат в  $\oplus\mathbf{P}$ , то  $A \cap B$ ,  $\overline{A}$ ,  $A \triangle B$ ,  $A \cup B$  также лежат в  $\oplus\mathbf{P}$ .

*Идея доказательства.* Нужно свести и  $A$ , и  $B$  к  $\oplus\text{SAT}$ , а затем с двумя полученными формулами произвести вышеописанные операции.  $\square$

## 9.5 Теорема Тоды

Из предыдущего обсуждения мы выяснили, что  $\mathbf{NP} \subset \mathbf{PP} \subset \mathbf{PSPACE}$ . Возникает естественный вопрос: где точно расположен класс  $\mathbf{PP}$  на этой шкале? В частности, как он соотносится с полиномиальной иерархией? Точный ответ на этот вопрос пока неизвестен, зато известно, что если использовать  $\mathbf{PP}$  в качестве оракула, то любой язык из полиномиальной иерархии распознать можно. В этом заключается известная теорема Тоды:

**Теорема 9.23** (Тода, [135]).  $\mathbf{PH} \subset \mathbf{P}^{\mathbf{PP}}$ .

Вначале заметим, что в этой теореме неважно, использовать ли оракул из класса  $\mathbf{PP}$  или из класса  $\#\mathbf{P}$ :

**Утверждение 9.24.**  $\mathbf{P}^{\mathbf{PP}} = \mathbf{P}^{\#\mathbf{P}}$ .

*Доказательство.* Утверждение доказывается так же, как и теорема 9.14: используя оракул для  $\#\mathbf{P}$ , можно посчитать число сертификатов и сравнить его с половиной, тем самым получив ответ оракула из  $\mathbf{PP}$ . А используя оракул для  $\mathbf{PP}$ , можно посчитать ответ оракула  $\#\mathbf{P}$  двоичным поиском.  $\square$

Доказывать теорему будем в 2 этапа: сначала докажем, что  $\mathbf{PH} \subset \mathbf{BPP}^{\oplus\mathbf{P}}$ . Затем мы дерандомизируем эту конструкцию, используя более сильный оракул, и получим  $\mathbf{BPP}^{\oplus\mathbf{P}} \subset \mathbf{P}^{\#\mathbf{P}}$ .

### 9.5.1 Построение вероятностного алгоритма

Начнём с базового случая:

**Лемма 9.25.**  $\mathbf{NP} \subset \mathbf{BPP}^{\oplus\mathbf{P}}$ .

*Доказательство.* Достаточно заметить, что оракул из  $\oplus\mathbf{P}$  позволяет решить задачу  $\text{USAT}$ : если  $\psi \in \text{USAT}_y$ , то  $\psi \in \oplus\text{SAT}$ , а если  $\psi \in \text{USAT}_n$ , то  $\psi \notin \oplus\text{SAT}$ . После этого утверждение следует из конструкции Вэлианта–Вазирани с достаточной амплификацией.  $\square$

**Замечание 9.26.** В предыдущей конструкции достаточно сделать один запрос к оракулу, независимо от степени амплификации, причём уже после запросов к случайным битам. Действительно, в ходе амплификации мы построим несколько формул, из которых хотя бы одна должна принадлежать  $\oplus\text{SAT}$ . Как мы уже знаем, можно построить одну формулу и спросить принадлежность к  $\oplus\text{SAT}$  только про неё. При этом запрос делается уже после выбора всех случайных битов, так что конструкцию можно назвать вероятностным сведением к  $\oplus\text{P}$ .

Заметим, что предыдущая лемма релятивизуется в следующем смысле:

**Лемма 9.27.** Для любого оракула  $C$  выполнено  $\text{NP}^C \subset \text{BPP}^{\oplus\text{P}^C}$ .

*Доказательство.* Для релятивизованных вычислений теорему Кука–Левина напрямую использовать уже нельзя, поэтому мы проведём рассуждение непосредственно для недетерминированной машины, обращающейся к оракулу  $C$ . Пусть язык  $A$  распознаётся недетерминированной машиной  $M$ , обращающейся к оракулу  $C$ . Построим случайным образом другую машину  $M'$ , которая вначале запускает машину  $M$ . Если  $M$  отвергла сертификат, то и  $M'$  отвергает. Если же  $M$  приняла сертификат  $y$ , то  $M'$  дополнительно проверяет, что  $h(y) = 0^k$ , где  $k \in [2, n+1]$  и  $h \in \mathcal{H}_{n,k}$  заранее выбраны случайно.

Аналогично рассуждению в теореме Вэлианта–Вазирани, если  $x \in A$ , то с существенной вероятностью  $M'$  примет ровно один сертификат, и это можно установить оракулом из  $\oplus\text{P}^C$ . Если же  $x \notin A$ , то  $M'$  не примет ничего. После применения амплификации мы получим, что  $A \in \text{BPP}^{\oplus\text{P}^C}$ , что и требовалось.  $\square$

Теперь мы будем доказывать по индукции, что  $\Sigma_k \text{P} \subset \text{BPP}^{\oplus\text{P}}$ . Вспомним, что полиномиальную иерархию можно определять через оракулы:  $\Sigma_{k+1} \text{P} = \text{NP}^{\Sigma_k \text{P}}$ . Если  $\Sigma_k \text{P} \subset \text{BPP}^{\oplus\text{P}}$  уже известно, то  $\Sigma_{k+1} \text{P} \subset \text{NP}^{\text{BPP}^{\oplus\text{P}}} \subset \text{BPP}^{\oplus\text{P}^{\text{BPP}^{\oplus\text{P}}}}$ , и нам достаточно доказать, что эта башня сокращается обратно до  $\text{BPP}^{\oplus\text{P}}$ . План такой: переставим местами второй и третий этажи, получив  $\text{BPP}^{\text{BPP}^{\oplus\text{P}^{\oplus\text{P}}}}$ , а затем уберём повторы. Обоснуем, почему так можно:

**Лемма 9.28.** Для любого оракула  $C$  выполнено  $\text{BPP}^{\text{BPP}^C} \subset \text{BPP}^C$ .

*Доказательство.* Идея доказательства заключается в том, чтобы заменять вызовы оракула из  $\text{BPP}^C$  на вычисления с оракулом  $C$ . Нужно только добиться, чтобы не накопилась слишком большая ошибка. Для этого нужно с самого начала сделать её достаточно маленькой.

А именно, пусть  $B \in \text{BPP}^C$ , а  $A \in \text{BPP}^B$ . При этом ошибку вычисления  $A$  мы сделаем равной  $2^{-a(n)}$ , а вычисления  $B$  — равной  $2^{-b(n)}$ , величины полиномов  $a$  и  $b$  выберем позже. Пусть вычисление  $A$  обращается к оракулу  $B$  не больше  $q(n)$  раз. Будем заменять каждое обращение к оракулу на вычисление вероятностным алгоритмом со свежими случайными битами. Тогда ошибка может возникнуть из двух источников:

- Все ответы оракула  $B$  посчитаны верно, но алгоритм, вычисляющий  $A$ , ошибся;
- Хотя бы один ответ оракула  $B$  посчитан неверно.

Вероятность первого варианта не превышает  $2^{-a(n)}$ , а второго — не больше  $1 - (1 - 2^{-b(n)})^{q(n)}$ . Сумму можно сделать меньше  $\frac{1}{3}$ , например, выбрав  $a(n) = n$ , а  $b(n) = nq(n)$ . (Выбор  $a$  определяет  $q$  и уже на базе этого можно выбрать  $b$ ).  $\square$

**Лемма 9.29.**  $\oplus \mathbf{P}^{\oplus \mathbf{P}} \subset \oplus \mathbf{P}$ .

*Доказательство.* Здесь идея состоит в том, чтобы заменить обращения к оракулу из  $\oplus \mathbf{P}$  на фрагменты вычислений, так чтобы положительный ответ оракула сменил чётность числа принимаемых сертификатов, а отрицательный — нет.

А именно, пусть  $B \in \oplus \mathbf{P}$ ,  $A \in \oplus \mathbf{P}^B$ . Пусть  $V$  — полиномиальный с оракулом  $B$  алгоритм, такой что  $x \in A$  тогда и только тогда, когда  $\bigoplus y V(x, y) = 1$ , а  $W$  — просто полиномиальный алгоритм, такой что  $z \in B$  тогда и только тогда, когда  $\bigoplus t W(z, t) = 1$ . Поскольку  $\oplus \mathbf{P}$  замкнуто относительно дополнения, можно рассмотреть и полиномиальный алгоритм  $\bar{W}$ , такой что  $z \notin B$  тогда и только тогда, когда  $\bigoplus t \bar{W}(z, t) = 1$ .

Дальнейшая идея состоит вот в чём: будем недетерминированно угадывать ответы оракула, а потом проверять положительные ответы при помощи  $W$ , а отрицательные — при помощи  $\bar{W}$ . Если ответ угадан неправильно, то принимающих ветвей при проверке будет чётное число, и они сократятся. Если же ответ угадан верно, то принимающих ветвей будет нечётное число. В результате вдоль единственного пути, в котором все ответы угаданы верно, число принимающих ветвей умножится на некоторое нечётное число, так что его чётность не изменится.

Изложим это план формально. Пусть машина  $S(x, y, a, t_1, \dots, t_m)$ , где  $|a| = m$ , а  $m$  — максимально возможное число вызовов оракула, работает так. Она моделирует вычисление  $V(x, y)$ . Как только оно вызывает оракул  $B$  на очередном входе  $z_i$ , машина в качестве результата берёт  $W(z_i, t_i)$ , если  $a_i = 1$ , и  $\bar{W}(z_i, t_i)$ , если  $a_i = 0$ . Если вычисление закончилось с менее чем  $m$  вызовами оракула, машина дополнительно проверяет, что все оставшиеся  $t_i$  состоят из одних единиц, и все оставшиеся  $a_i$  тоже равны единице.

Проверим, что чётность числа сертификатов у  $V$  и  $S$  одинакова. Вначале рассмотрим случай, когда вектор  $a$  состоит из правильных ответов (включая единицы для незапрошенных позиций). Тогда для каждого  $i$  подойдёт нечётное число разных  $t_i$  (включая незапрошенные, для которых подойдёт ровно один вариант). Все эти числа умножатся на количество подходящих  $y$ , так что чётность сохранится. Если же хотя бы один бит  $a$  неверен, то для соответствующего  $z_i$  подойдёт чётное число разных  $t_i$ , так что они не изменят чётность ответа. Таким образом, чётность числа сертификатов у  $V$  и  $S$  действительно одинаковая, и язык  $A$  лежит в  $\oplus \mathbf{P}$ , что и требовалось.  $\square$

**Лемма 9.30.** Для любого оракула  $C$  выполнено  $\oplus \mathbf{P}^{\mathbf{BPP}^C} \subset \mathbf{BPP}^{\oplus \mathbf{P}^C}$ .

*Доказательство.* Идея заключается в том, что кванторы «для нечётного числа  $y$ » и «для подавляющего большинства  $z$ » можно переставить: если в нечётном числе строк закрашено подавляющее большинство клеток, а в остальных — ничтожное меньшинство, то в большинстве (хоть и не настолько подавляющем) столбцов закрашено нечётное число клеток.

Изложим этот план подробнее. Пусть  $A \in \oplus \mathbf{P}^B$ , а  $B \in \mathbf{BPP}^C$ . Пусть верификатор  $V$  распознаёт  $A$  с оракулом  $B$ , т.е.  $x \in A$  тогда и только тогда, когда для нечётного числа  $y$  выполнено  $V(x, y) = 1$ . Сам алгоритм  $V$  работает за полиномиальное время с обращениями к оракулу  $B$ . Это значит, что соответствующий предикат лежит в  $\mathbf{P}^{\mathbf{BPP}^C}$ , т.е. по лемме 9.28 просто в  $\mathbf{BPP}^C$ . Ошибку сделаем равной  $2^{-b(n)}$ , где полином  $b(n)$  выберем позднее. Обозначим новый верификатор через  $W$ , а его случайные биты через  $z$ . Тогда получается, что при  $x \in A$  имеется нечётное число  $y$ , для которых для доли  $\geq 1 - 2^{-b(n)}$  слов

$z$  выполнено  $V(x, y, z) = 1$ , а для остальных  $y$  это будет выполнено для доли  $\leq 2^{-b(n)}$  слов  $z$ . Если же  $x \notin A$ , то слов  $y$  первого типа будет чётное число. Если взять  $b(n)$  больше, чем  $|y| + 2$ , то в первом случае для доли  $\geq \frac{3}{4}$  слов  $z$  будет нечётное число  $y$ , таких что  $W(x, y, z) = 1$ , а во втором случае доля таких  $z$  не превысит  $\frac{1}{4}$ . Действительно, если бы ошибки не было, то эти доли были бы равны 1 и 0 соответственно, а ошибка для каждого  $y$  изменяет каждую из этих долей не больше, чем на  $2^{-b(n)}$ .

Теперь понятно, как доказать  $A \in \mathbf{BPP}^{\oplus \mathbf{P}^C}$ : получив  $x$  на вход, нужно взять случайное  $z$  и спросить у оракула из  $\oplus \mathbf{P}^C$ , какова чётность числа  $y$ , для которых  $W(x, y, z) = 1$ . Вышеприведённые оценки доказывают корректность такого алгоритма.  $\square$

Итак, мы получили первую часть теоремы Тоды:

**Теорема 9.31.**  $\mathbf{PH} \subset \mathbf{BPP}^{\oplus \mathbf{P}}$ .

*Доказательство.* Напомним основные этапы. Поскольку  $\mathbf{PH} = \bigcup_{k=1}^{\infty} \Sigma_k \mathbf{P}$ , нам достаточно доказать  $\Sigma_k \mathbf{P} \subset \mathbf{BPP}^{\oplus \mathbf{P}}$  для всех  $k$ . Мы доказываем это по индукции. Базой ( $k = 1$ ) служит лемма 9.25. Для перехода мы используем такую цепочку вложений:  $\Sigma_{k+1} \mathbf{P} = \mathbf{NP}^{\Sigma_k \mathbf{P}} \subset \langle \text{по предположению индукции} \rangle \subset \mathbf{NP}^{\mathbf{BPP}^{\oplus \mathbf{P}}} \subset \langle \text{по лемме 9.27} \rangle \subset \mathbf{BPP}^{\oplus \mathbf{BPP}^{\oplus \mathbf{P}}} \subset \langle \text{по лемме 9.30} \rangle \subset \mathbf{BPP}^{\mathbf{BPP}^{\oplus \mathbf{P}}} \subset \langle \text{по лемме 9.29} \rangle \subset \mathbf{BPP}^{\mathbf{BPP}^{\oplus \mathbf{P}}} \subset \langle \text{по лемме 9.28} \rangle \subset \mathbf{BPP}^{\oplus \mathbf{P}}$ , что и требовалось.  $\square$

**Замечание 9.32.** Анализ конструкции показывает, что и тут можно обойтись только одним запросом к оракулу, причём после выбора случайных битов. Действительно, для «нижнего» оракула из  $\oplus \mathbf{P}$  это делается аналогично замечанию 9.26. Перемена местами  $\oplus \mathbf{P}$  и  $\mathbf{BPP}$  сохраняет единственность вызова и порядок запроса. Конструкция из леммы 9.29 также оставляет только один запрос, а из леммы 9.28 его сохраняет.

## 9.5.2 Дерандомизация

Теперь мы докажем вторую часть теоремы Тоды:

**Теорема 9.33.**  $\mathbf{BPP}^{\oplus \mathbf{P}} \subset \mathbf{P}^{\# \mathbf{P}}$ .

*Доказательство.* Мы ограничимся разбором случая, когда алгоритм делает один запрос к оракулу  $\oplus \mathbf{P}$ , а точнее даже к  $\oplus \mathbf{SAT}$ . Идея заключается в том, чтобы по формуле, у числа выполняющих наборов которой мы знаем чётность, построить формулу, у числа выполняющих наборов которой мы знаем остаток по большому модулю, и этот остаток 0 или  $-1$ . Случайные биты будут параметром этого преобразования, и мы просуммируем вышеуказанные остатки для всех вариантов случайных битов. Модуль будет таким большим, чтобы диапазоны сумм для разных ответов алгоритма не пересекались. А саму сумму мы узнаем с помощью оракула  $\# \mathbf{P}$ .

Нам потребуется такая лемма:

**Лемма 9.34.** Пусть формула  $\varphi$  такова, что  $\#\varphi$  сравнимо с нулём или с  $-1$  по модулю  $2^q$ . Тогда для формулы  $\psi = 4 \cdot \varphi^3 + 3 \cdot \varphi^4$  величина  $\#\psi$  сравнима, соответственно, с нулём или с  $-1$  по модулю  $2^{2q}$ . (В записи  $4 \cdot \varphi^3 + 3 \cdot \varphi^4$  сложение и умножение понимаются в смысле операций над формулами из раздела 9.4. Как обычно, возведение в степень понимается как итерированное умножение. Множители 3 и 4 означают некоторые канонические формулы с таким количеством выполняющих наборов).



*Доказательство.* Пусть  $\varphi$  было  $a \cdot 2^q$  выполняющих наборов. Тогда у формулы  $\psi$  число выполняющих наборов равно  $4 \cdot a^3 \cdot 2^{3q} + 3 \cdot a^4 \cdot 2^{4q} = (4a^3 2^q + 3a^4 2^{2q})2^{2q}$ , т.е. сравнимо с нулём по модулю  $2^{2q}$ .

Пусть теперь у  $\varphi$  был  $a \cdot 2^q - 1$  выполняющий набор. Тогда число выполняющих наборов у  $\psi$  равно  $4 \cdot (a \cdot 2^q - 1)^3 + 3(a \cdot 2^q - 1)^4 = 4(a^3 2^{3q} - 3a^2 2^{2q} + 3a 2^q - 1) + 3(a^4 2^{4q} - 4a^3 2^{3q} + 6a^2 2^{2q} - 4a 2^q + 1) = 3a^4 2^{4q} - 8a^3 2^{3q} + 6a^2 2^{2q} - 1$ , т.е. сравнимо с  $-1$  по модулю  $2^{2q}$ .  $\square$

Итак, у нас есть язык из  $\mathbf{BPP}^{\oplus \mathbf{P}}$ , который делает один запрос к оракулу  $\oplus \mathbf{SAT}$  после выбора  $r(n)$  случайных битов и возвращает тот же ответ. Ошибку будем считать стандартной:  $\frac{1}{3}$ . Запрос к  $\oplus \mathbf{SAT}$  означает, что для некоторой формулы  $\varphi_z$ , вычисленной по случайным битам  $z$ , мы узнаём у оракула, какой остаток у числа её выполняющих наборов по модулю 2. Многократным применением леммы 9.34 можно сделать новую формулу  $\psi_z$ , у которой число выполняющих наборов сравнимо с 0 или с  $-1$  по модулю  $2^{r(n)+1}$  в зависимости от исходного ответа. (Непосредственное применение индукции позволит сделать только модуль с показателем — степенью двойки, но достаточно взять степень двойки, превышающую  $r(n) + 1$ . При этом потребуется  $O(\log r(n)) = O(\log n)$  итераций, на каждой итерации формула вырастет в константу раз, так что общий рост формулы будет полиномиальным). Теперь запросом к оракулу  $\# \mathbf{P}$  можно узнать, сколько в сумме выполняющих наборов у  $\psi_z$  для всех возможных значений  $z$ . Если изначально слово  $x$  лежало в языке, то хотя бы для  $\frac{2}{3}$  всех  $z$  у  $\varphi_z$  было нечётное число выполняющих наборов. Значит, хотя бы для  $\frac{2}{3}$  всех  $z$  число выполняющих наборов у  $\psi_z$  даёт остаток  $-1$  по модулю  $2^{r(n)+1}$ , а для остальных — остаток 0. Значит, суммарное число выполняющих наборов будет в интервале от  $-2^{r(n)}$  до  $-\frac{2}{3}2^{r(n)}$  по модулю  $2^{r(n)+1}$ . Если же изначально слово  $x$  не лежало в языке, то этот остаток будет в интервале от  $-\frac{1}{3}2^{r(n)}$  до нуля. Поскольку диапазоны не пересекаются, из ответа  $\# \mathbf{P}$ -оракула мы узнаем, какой из двух случаев имеет место.  $\square$

## 9.6 Подсчёт числа изоморфизмов в графе

Из теоремы Тоды можно сделать вывод, что подсчёт числа выполняющих наборов булевой формулы сложнее, чем определение, существуют ли такие наборы. Действительно, умение решать  $\# \mathbf{SAT}$  позволяет решать все задачи из  $\# \mathbf{P}$ , а значит и все задачи из  $\mathbf{RP}$ . Иными словами,  $\mathbf{P}^{\# \mathbf{SAT}} = \mathbf{P}^{\mathbf{RP}} \supset \mathbf{PH}$ . С другой стороны,  $\mathbf{P}^{\mathbf{SAT}} = \mathbf{P}^{\mathbf{NP}} = \Delta_2^{\mathbf{P}} \subset \Sigma_2^{\mathbf{P}} \cap \Pi_2^{\mathbf{P}}$ . Таким образом, если полиномиальная иерархия не схлопывается, то  $\mathbf{P}^{\mathbf{SAT}} \subsetneq \mathbf{P}^{\# \mathbf{SAT}}$ . Также можно заключить, что в этом случае  $\# \mathbf{SAT} \notin \mathbf{FP}^{\mathbf{SAT}}$ . Такие же соотношения верны и для других  $\mathbf{NP}$ -полных задач. А что будет для не  $\mathbf{NP}$ -полных? Оказывается, для изоморфизма графов задачи подсчёта и распознавания эквивалентны. Напомним определения:

**Определение 9.35.** Языком  $\mathbf{GI}$  называется множество  $\{(G, H) \mid \text{неориентированные графы } G \text{ и } H \text{ изоморфны}\}$ . Задачей  $\# \mathbf{GI}$  называется задача вычисления по графам  $G$  и  $H$  количества изоморфизмов между ними.

**Теорема 9.36.**  $\# \mathbf{GI} \in \mathbf{FP}^{\mathbf{GI}}$ .

*Доказательство.* Прежде всего заметим, что если графы  $G$  и  $H$  изоморфны, то количество изоморфизмов между ними равно количеству автоморфизмов каждого из них. Поэтому достаточно сначала узнать, есть ли хоть один изоморф-



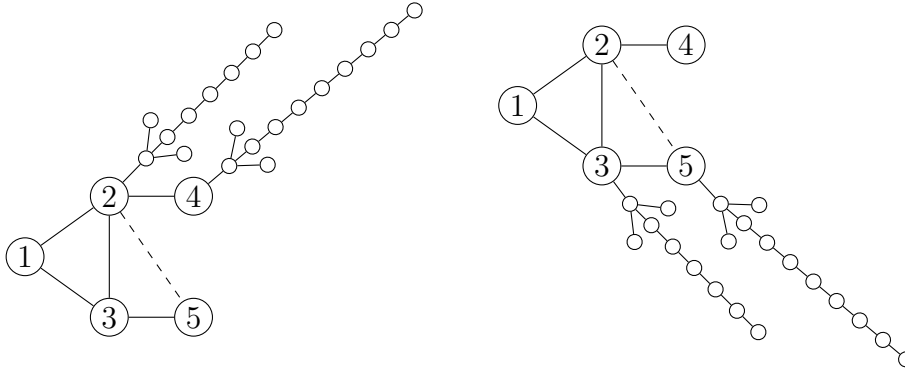


Рис. 9.1: Проверка на существование автоморфизма при помощи GI. Графы с добавленными «стрелками» изоморфны тогда и только тогда, когда в исходном графе есть автоморфизм, переводящий 4 в 5, а 2 в 3. Это верно при отсутствии ребра (2, 5) и неверно при его наличии.

зим, а потом в случае положительного ответа подсчитать число автоморфизмов. Для подсчёта нам потребуется такая лемма.

**Лемма 9.37.** Обозначим через  $A_{i \rightarrow j}^{[i-1]}$  множество всех автоморфизмов графа  $G$ , сохраняющих вершины  $1, \dots, i-1$  и переводящих  $i$  в  $j$ . Тогда для любого  $j > i$  либо  $|A_{i \rightarrow j}^{[i-1]}| = 0$ , либо  $|A_{i \rightarrow j}^{[i-1]}| = |A_{i \rightarrow i}^{[i-1]}|$ .

*Доказательство.* Пусть  $A_{i \rightarrow j}^{[i-1]}$  непусто, зафиксируем некоторый  $\pi \in A_{i \rightarrow j}^{[i-1]}$ . Тогда для любого  $\sigma \in A_{i \rightarrow i}^{[i-1]}$  композиция  $\pi \circ \sigma$  также лежит в  $A_{i \rightarrow j}^{[i-1]}$ . При этом если  $\sigma \neq \sigma'$ , то и  $\pi \circ \sigma \neq \pi \circ \sigma'$ , т.к. автоморфизмы образуют группу. С другой стороны, любой  $\tau \in A_{i \rightarrow j}^{[i-1]}$  представляется в виде  $\pi \circ \sigma$  для  $\sigma = \pi^{-1} \circ \tau$ . Таким образом, умножение на  $\pi$  задаёт биекцию между  $A_{i \rightarrow j}^{[i-1]}$  и  $A_{i \rightarrow i}^{[i-1]}$ , что и требовалось.  $\square$

Теперь нетрудно заметить такое соотношение: количество автоморфизмов, сохраняющих  $1, \dots, i-1$ , равно количеству автоморфизмов, сохраняющих  $1, \dots, i$ , умноженному на число таких  $j \geq i$ , что некоторый автоморфизм, сохраняющий  $1, \dots, i-1$ , переводит  $i$  в  $j$ . Проверить, есть ли такой автоморфизм, можно при помощи GI-оракула согласно следующей лемме.

**Лемма 9.38.** Пусть фиксированы наборы вершин графа  $\{i_1, \dots, i_k\}$  и  $\{j_1, \dots, j_k\}$ . Тогда при помощи одного запроса к оракулу GI можно вычислить, существует ли автоморфизм, переводящий все  $i_r$  в соответствующие  $j_r$ .

*Доказательство.* Возьмём два экземпляра исходного графа. В первом экземпляре для каждого  $r$  приделаем к вершине  $i_r$  «стрелку»: цепочку длины  $n + i_r$  и ещё два отдельных ребра из ближайшей к  $i_r$  вершины этой цепочки. Во втором экземпляре такую же стрелку приделаем к вершине  $j_r$ . Если эти графы изоморфны, то в исходном графе есть искомый автоморфизм, иначе нет. Действительно, длина стрелки гарантирует, что в исходном графе такой конструкции нет, поэтому стрелки переходят в стрелки. Дополнительные рёбра позволяют понять, где кончается стрелка и начинается исходный граф, а различные длины стрелок позволяют однозначно их друг с другом сопоставить: стрелка, приделанная к  $i_r$ , обязательно перейдёт в стрелку, приделанную к  $j_r$ .  $\square$

Теперь всё готово к построению алгоритма, вычисляющего число автоморфизмов. Идея состоит в том, чтобы последовательно вычислять количество автоморфизмов, сохраняющих  $1, \dots, i$  для  $i$ , убывающего от  $n$  до 0. Для  $i = n$  бу-

дет только один автоморфизм. Далее на каждом шаге цикла нужно посчитать, в какое количество разных  $j$  может перейти  $i$ , при том, что  $1, \dots, i-1$  остаются на месте, и увеличить текущее количество автоморфизмов в такое число раз. Число, получившееся после прохода цикла, будет ответом. Ниже приводится псевдокод.

<b>Вход:</b> Граф $G$ , оракул $G^I$	
<b>Результат:</b> Число автоморфизмов графа $G$	
1	$a := 1;$ // Тривиальный автоморфизм всегда есть
2	<b>для</b> $i$ <b>от</b> $n$ <b>до</b> $1$ <b>выполнить</b>
3	$a_i := 1;$ // Всегда есть автоморфизм, сохраняющий $1, \dots, i-1, i$
4	<b>для</b> $j$ <b>от</b> $i+1$ <b>до</b> $n$ <b>выполнить</b>
5	<b>если</b> <i>существует автоморфизм, сохраняющий <math>1, \dots, i-1</math> и переводящий <math>i</math> в <math>j</math></i> <b>то</b> $a_i += 1;$ // Проверка осуществляется по алгоритму из леммы 9.38
6	<b>конец цикла</b> // Теперь $a_i$ равно числу возможных $j$ , в которых переходит $i$ при автоморфизме, сохраняющем $1, \dots, i-1$
7	$a *= a_i;$ // Теперь $a$ равно числу автоморфизмов, сохраняющих $1, \dots, i-1$
8	<b>конец цикла</b> // Теперь $a$ равно числу всех автоморфизмов
9	<b>возвратить</b> $a;$

**Алгоритм 2:** Подсчёт числа автоморфизмов

□

## 9.7 Иерархия подсчёта

Мы выяснили, что  $\mathbf{P}^{\mathbf{H}} \subset \mathbf{P}^{\mathbf{P}^{\mathbf{P}}}$ , однако вопрос о  $\mathbf{P}^{\mathbf{H}} \subset \mathbf{P}^{\mathbf{P}}$  открыт. Значит,  $\mathbf{P}^{\mathbf{P}^{\mathbf{P}}}$  может быть более широким классом, чем сам  $\mathbf{P}^{\mathbf{P}}$ .<sup>4</sup> А что будет, если нарастить башню ещё сильнее?

Другой вопрос: мы видели, что языки из  $\mathbf{P}^{\mathbf{P}}$  получаются навешиванием квантора «для большинства» на полиномиально разрешимые множества пар. А что, если взять множество троек и навесить два квантора подряд? А ещё дальше? Оказывается, и при наращивании башни, и при навешивании кванторов получится один и тот же объект — иерархия задач подсчёта (counting hierarchy).

**Определение 9.39.** Уровни иерархии подсчёта определяются следующим образом:

- $\mathbf{C}_0 \mathbf{P} = \mathbf{P};$
- $\mathbf{C}_{k+1} \mathbf{P} = \mathbf{P}^{\mathbf{C}_k \mathbf{P}}.$

Иерархия подсчёта  $\mathbf{CH}$  определяется как  $\bigcup_{k=1}^{\infty} \mathbf{C}_k \mathbf{P}.$

<sup>4</sup>Известны результаты [18, 49], что  $\mathbf{P}^{\parallel \mathbf{P}^{\mathbf{P}}} = \mathbf{P}^{\mathbf{P}}$ , т.е. неадаптивные обращения к  $\mathbf{P}^{\mathbf{P}}$ -оракулу не добавляют вычислительной силы по сравнению с одним запуском, а также что  $\mathbf{P}_{O(\log n)}^{\mathbf{P}^{\mathbf{P}}} = \mathbf{P}^{\mathbf{P}}$ , т.е.  $O(\log n)$  адаптивных обращений к оракулу также не добавляют вычислительной силы. Однако все использованные для него аргументы рассыпаются при переходе к полиномиальному числу адаптивных обращений. Неудивительно, ведь для двоичного поиска нужны именно адаптивные обращения и именно в таком количестве.

Таким образом,  $C_0 P = P$ ,  $C_1 P = PP^P = PP$ ,  $C_2 P = PP^{PP}$ ,  $C_3 P = PP^{PP^{PP}}$ , и т.д. Альтернативный способ описания использует не оракулы, а кванторы «по большинству».

**Теорема 9.40.** *Язык  $A$  лежит в  $C_{k+1} P$  тогда и только тогда, когда найдётся некоторое множество пар  $V \in C_k P$ , такое что  $x \in A$  тогда и только тогда, когда для большинства  $y$  выполнено  $V(x, y) = 1$ . Иными словами,  $x \in A \Leftrightarrow \text{Maj } yV(x, y)$ .*

*Доказательство.* Как часто бывает, в одну сторону доказательство простое. Если  $x \in A \Leftrightarrow \text{Maj } yV(x, y)$ , то  $A \in PP^V$ : оракул даст значения  $V(x, y)$ , а в  $PP$  можно понять, каких значений большинство. Поскольку  $V \in C_k P$ , имеем  $A \in PP^{C_k P}$ .

В другую сторону нужно доказать, что если  $A \in PP^W$ , а  $W \in C_k P$ , то для некоторого другого  $V \in C_k P$  выполнено  $x \in A \Leftrightarrow \text{Maj } yV(x, y)$ . Трудность состоит в том, что запросов к  $W$  может быть много, и нужно построить  $V$ , который все их объединит в один.  $\square$

## 9.8 Исторические замечания и рекомендации по литературе

## 9.9 Задачи и упражнения



# Глава 10

## Сложность в среднем

*Когда Билл Гейтс заходит в бар, все его посетители  
становятся в среднем миллиардерами*

Интернет-фольклор

В теории алгоритмов известны случаи, когда алгоритм на практике работает существенно быстрее, чем при анализе худшего случая в теории. Так, всем известная «быстрая сортировка» в худшем случае работает за  $\Omega(n^2)$  шагов, но в среднем работает за  $O(n \log n)$  и часто быстрее других методов. Задачи линейного программирования на практике решаются экспоненциальным в худшем случае симплекс-методом Данцига, а не полиномиальным методом эллипсоидов Хачияна. Равновесие Нэша часто успешно ищется алгоритмом Лемке–Хоусона, хотя в худшем случае он также экспоненциальный. Если бы похожие алгоритмы существовали для всех **NP**-полных задач, то вопрос  $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$  не стоял бы так остро: для большинства входов ответ находился бы быстро (с другой стороны, не было бы и сложных задач, которые можно использовать в криптографии).

### 10.1 Роль распределения входов

Если говорить о сложности в среднем, то ключевую роль играет вероятностное распределение. С одной стороны, некоторые **NP**-полные задачи легко решаются для некоторых распределений. Рассмотрим, например, задачу **dist3COL** о раскраске в 3 цвета случайного графа  $G(n, p)$  для  $p = \frac{1}{2}$ . Напомним, что это означает, что каждое ребро проводится с вероятностью  $\frac{1}{2}$  независимо от других.

**Утверждение 10.1.** *Задача **dist3COL** решается в среднем за  $O(n)$  шагов.*

*Идея доказательства.* Дело в том, что в графе  $G(n, \frac{1}{2})$  почти наверняка есть клика на 4 вершинах. Поэтому сначала нужно поискать такую клику, а если не найдётся, запустить полный перебор. Экспоненциально малая вероятность уравнивает экспоненциально длинный перебор, и среднее время составит  $O(n)$ . Надо только объяснить, как искать такую клику: полный перебор хоть и полиномиален, всё же имеет четвертую степень. Нужно делать так: возьмём произвольную вершину  $u$ , оставим в графе только её соседей. Далее возьмём соседа  $v$ , оставим только тех, кто и сосед  $v$  тоже. Далее возьмём  $w$  из оставшихся, и если остались соседи всех трёх, то клика из 4 вершин найдётся. В среднем должно остаться  $\frac{n}{8}$  вершин, так что хотя бы одна будет с подавляющей вероятностью.  $\square$

Похожие результаты есть и для некоторых других задач.

С другой стороны, есть очень хитрое распределение, при котором для любого алгоритма сложность в среднем и сложность в худшем случае совпадают. Идея заключается в том, что вероятности входов, на которых алгоритм работает долго, завышаются как раз так, чтобы эти входы стали типичными. Для доказательства используется техника из теории колмогоровской сложности.

Отсюда видно, что для построения отдельной теории сложности в среднем нужно рассматривать, с одной стороны, не слишком узкий класс распределений, а с другой стороны, не слишком широкий. В некотором смысле нужно рассматривать «естественным образом» возникающие распределения. Ведь когда задачи возникают на практике, никто не пытается специально сделать их труднорешаемыми, условия порождаются некоторыми естественными процессами. Иногда эти процессы сами по себе алгоритмические, как при составлении капчи при входе на сайт. Часто можно надеяться, что их можно алгоритмически смоделировать.

Как известно, непрерывное распределение вероятностей можно задавать двумя способами: через функцию распределения и через плотность. В случае дискретной случайной величины аналогом плотности будут вероятности каждого отдельного слова. Для определения функции распределения нужно ввести некоторый порядок на словах, лучше всего подойдёт стандартный лексикографический. В таком случае мы будем использовать такие определения:

**Определение 10.2.** Пусть  $\{\mu_n\}_{n=1}^\infty$  — семейство случайных величин на  $\{0, 1\}^n$  (вероятностный ансамбль). Тогда плотностью распределения мы будем называть отображение  $p_\mu$ , определяемое как  $p_\mu(x) = \Pr[\mu = x]$ , а функцией распределения — отображение  $F_\mu$ , определяемое как  $F_\mu(x) = \Pr[\mu \leq x]$ .

Из определений прямо вытекают такие соотношения:

**Утверждение 10.3.** •  $F_\mu(x) = \sum_{y \leq x} p_\mu(y)$ .

•  $p_\mu(x) = F_\mu(x) - F_\mu(x-1)$ , где  $x-1$  — предшествующее  $x$  слово, а если  $x$  из одних нулей, то  $F_\mu(x-1) = 0$

Отсюда видно, что полиномиальная вычислимость  $p_\mu$  и  $F_\mu$  это не одно и то же: по  $F_\mu$  можно быстро посчитать  $p_\mu$ , но обратный подсчёт может требовать экспоненциального суммирования. Можно показать, что при  $\mathbf{P} \neq \mathbf{NP}$  существует распределение, для которого  $p_\mu$  вычислимо за полиномиальное время, а  $F_\mu$  — нет. Этот факт мотивирует следующее определение:

**Определение 10.4.** Распределение  $\mu$  называется полиномиально вычислимым, если функция распределения  $F_\mu$  вычислима за полиномиальное время.

Есть более широкая формализация «распределения, возникающего на практике». Действительно, для моделирования случайной величины не нужно знать её функцию распределения. Нужно просто уметь её порождать. Это мотивирует следующее определение:

**Определение 10.5.** Распределение  $\mu$  называется полиномиально порождаемым, если существует полиномиальный вероятностный алгоритм со входом  $1^n$ , такой что его выход распределён на  $\{0, 1\}^n$  так же, как и  $\mu$ .

При помощи двоичного поиска можно показать, что любое полиномиально вычислимое распределение является полиномиально порождаемым. Обратное

может быть неверно, а в предположении  $\mathbf{P} \neq \mathbf{P}^{\# \mathbf{P}}$  можно построить пример полиномиально порождаемого распределения, которое не полиномиально вычислимо. Тем не менее, мы в основном ограничимся полиномиально вычислимыми распределениями.

Завершим раздел формальным определением, какие же задачи мы будем изучать:

**Определение 10.6.** *Задачей с распределением на входах* называется пара  $(A, \mu)$ , где  $A \subset \{0, 1\}^*$  — некоторый язык, а  $\mu$  — вероятностный ансамбль.

## 10.2 Полиномиальность в среднем и класс $\text{distP}$

Теперь поговорим о том, как определять полиномиальность времени работы в среднем. Первая наивная идея — потребовать, чтобы среднее по заданному распределению время работы было ограничено полиномом. К сожалению, эта идея сталкивается с проблемой: ожидание случайной величины может быть полиномом, а её квадрата — экспонентой. Например, такой будет величина, которая во всех точках равна нулю, а в одной равна  $2^n$ . Её ожидание равно 1, а ожидание её квадрата —  $2^n$ . Этот эффект вызывает сразу две проблемы:

- Определение становится зависимым от вычислительной модели: напомним, что смена модели может возвести время вычисления в квадрат. И именно инвариантность класса  $\mathbf{P}$  относительно выбора модели была одной из мотиваций его выбора в качестве базового.
- Хуже того, даже если модель фиксировать, то получившийся класс полиномиальных в среднем алгоритмов не будет замкнут относительно композиции. Это означает, что один алгоритм нельзя использовать как подпрограмму в другом алгоритме.

Значит, указанный негативный эффект нужно каким-то образом нивелировать. Предлагается такой метод: будем возводить время работы в некоторую степень, чтобы оно стало в среднем линейным.

**Определение 10.7.** Обозначим через  $\text{time}_M(x)$  время работы алгоритма  $M$  на входе  $x$ . Классом  $\text{distP}$  называется множество задач с распределением на входах  $(A, \mu)$ , для которых существует алгоритм  $M$  с такими свойствами:

- $M$  распознаёт язык  $A$ ;
- Для некоторых констант  $\varepsilon > 0$  и  $C$  при любом  $n$  выполнено

$$\mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon}{n} < C. \quad (10.1)$$

Обратите внимание, что здесь от распределения  $\mu$  ничего не требуется. В частности, верно следующее

**Утверждение 10.8.** *Если  $A \in \mathbf{P}$ , то для любого  $\mu$  выполнено  $(A, \mu) \in \text{distP}$ .*

*Доказательство.* Действительно, если  $A \in \mathbf{P}$ , то  $A$  распознаётся некоторым алгоритмом  $M$  со временем работы не больше  $cn^k$ . Взяв  $\varepsilon = \frac{1}{k}$ , получаем требуемое условие.  $\square$

С другой стороны, как мы видели,  $\text{dist3COL} \in \mathbf{distP}$ , так что первая компонента задачи из  $\mathbf{distP}$  не обязана лежать в  $\mathbf{P}$ .

Класс  $\mathbf{distP}$  замкнут относительно теоретико-множественных операций при одном и том же распределении:

**Утверждение 10.9.** *Если  $(A, \mu) \in \mathbf{distP}$  и  $(B, \mu) \in \mathbf{distP}$ , то  $(\bar{A}, \mu)$ ,  $(A \cap B, \mu)$  и  $(A \cup B, \mu)$  также лежат в  $\mathbf{distP}$ .*

*Доказательство.* Проведём для примера рассуждение на  $A \cap B$ . Если машина  $M$  распознаёт  $A$  за время  $\text{time}_M(x)$ , а машина  $N$  распознаёт  $B$  за время  $\text{time}_N(x)$ , то язык  $A \cap B$  можно распознать за время  $\text{time}_M(x) + \text{time}_N(x) + O(1)$ . Пусть выполнены соотношения  $\mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon}{n} < C$  и  $\mathbb{E}_{\mu_n} \frac{(\text{time}_N(x))^\delta}{n} < D$ . Без ограничения общности полагаем, что  $\varepsilon \leq \delta < 1$ . В таком случае

$$\begin{aligned} \mathbb{E}_{\mu_n} \frac{(\text{time}_M(x) + \text{time}_N(x) + O(1))^\varepsilon}{n} &< \mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon + (\text{time}_N(x))^\varepsilon + O(1)}{n} \leq \\ &\leq \mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon + (\text{time}_N(x))^\delta + O(1)}{n} = \mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon}{n} + \mathbb{E}_{\mu_n} \frac{(\text{time}_N(x))^\delta}{n} + O(1) < C + D + \end{aligned}$$

где первое неравенство следует из неравенства  $(a + b)^\varepsilon \leq a^\varepsilon + b^\varepsilon$  при  $\varepsilon \in (0, 1)$ , второе следует из  $\varepsilon \leq \delta$ , равенство следует из линейности математического ожидания, а последнее неравенство выполнено по предположению. Таким образом, условие  $(A \cap B, \mu) \in \mathbf{distP}$  доказано, что и требовалось.  $\square$

Нам будет полезно такое эквивалентное определение:

**Утверждение 10.10.** *Класс  $\mathbf{distP}$  не изменится, если в определении 10.7 заменить второе условие на такое: для некоторой константы  $d \geq 1$  и некоторых констант  $\varepsilon > 0$  и  $C$  при любом  $n$  выполнено*

$$\mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon}{n^d} < C. \quad (10.2)$$

*Доказательство.* Из неравенства (10.1) следует (10.2) для тех же самых  $\varepsilon$  и  $C$ : достаточно заметить  $\frac{1}{n^d} \leq \frac{1}{n}$ . В обратную сторону подойдут константы  $\frac{\varepsilon}{d}$  и  $C^{\frac{1}{d}}$ . Действительно,

$$\mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^{\frac{\varepsilon}{d}}}{n} = \mathbb{E}_{\mu_n} \left( \frac{(\text{time}_M(x))^\varepsilon}{n^d} \right)^{\frac{1}{d}} < \left( \mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon}{n^d} \right)^{\frac{1}{d}} < C^{\frac{1}{d}},$$

где первое неравенство следует из неравенства Иенсена, а второе неравенство — из (10.2).  $\square$

### 10.3 Класс $\mathbf{distNP}$

Наша цель — построить аналог теории  $\mathbf{NP}$ -полноты для сложности в среднем. Для этого нужны следующие шаги:

1. Определить аналог класса  $\mathbf{NP}$ ;
2. Определить полиномиальную сводимость, которая была бы транзитивна, а сведение к задаче из  $\mathbf{distP}$  доказывало бы принадлежность к  $\mathbf{distP}$ ;
3. Найти полные задачи для так определённой сводимости.



Вспомним, откуда вообще возникла наша задача: нам интересно, насколько сложно решать задачи из **NP** в «типичном» случае. Поэтому не нужно менять класс языков. А вот ограничить распределения нужно, т.к. если распределение будет произвольным, то никакой отдельной теории не получится. Это мотивирует следующее

**Определение 10.11.** Класс **distNP** состоит из задач с распределением на входах  $(A, \mu)$ , таких что  $A \in \mathbf{NP}$ , а  $\mu$  полиномиально вычислимо.

### 10.3.1 Сводимость в **distNP**

Нам нужно определить сводимость  $\leq_{\text{dist}}$  задачи с распределением на входах  $(A, \mu)$  к  $(B, \nu)$ , удовлетворяющей следующим свойствам:

- Транзитивность: если  $(A, \mu) \leq_{\text{dist}} (B, \nu)$ , а  $(B, \nu) \leq_{\text{dist}} (C, \kappa)$ , то  $(A, \mu) \leq_{\text{dist}} (C, \kappa)$ .
- Возможность использовать решение той задачи, к которой осуществляется сводимость: если  $(A, \mu) \leq (B, \nu)$ , а  $(B, \nu) \in \mathbf{distP}$ , то  $(A, \mu) \in \mathbf{distP}$ .

Второе условие показывает, что обычную сводимость по Карпу на первых компонентах использовать нельзя: типичная задача для  $A$  может отобразиться в редкую задачу для  $B$ , на которой алгоритм работает очень долго. Значит, нужно наложить условия на распределения, так чтобы такого не получилось. Кроме того, мы решили рассматривать отдельные распределения в каждой длине, а распределения на длинах не определяли. Поэтому нужно, чтобы и при сводимости разные длины не перемешивались. Все эти соображения мотивируют следующее определение:

**Определение 10.12.** Задача с распределением на входах  $(A, \mu)$  полиномиально сводится к задаче  $(B, \nu)$ , если существует такая полиномиально вычислимая функция  $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$  и полином  $q$ , что выполнены следующие свойства:

1. **Регулярность.** Длина  $f(x)$  зависит только от длины  $x$ . (Из полиномиальности  $f$  следует, что эта длина ограничена некоторым полиномом  $s(|x|)$ ).
2. **Сводимость по Карпу.** Слово  $x$  лежит в  $A$  тогда и только тогда, когда  $f(x) \in B$ .
3. **Доминирование.** Для всех  $n$  и всех  $y$  выполнено  $\sum_{x \in f^{-1}(y)} p_\mu(x) \leq q(n) p_\nu(y)$ .

Свойство доминирования как раз гарантирует, что в слишком редкие  $y$  не перейдут слова  $x$  слишком большого веса. Покажем, что наложенные требования действительно выполнены.

**Утверждение 10.13.** Полиномиальная сводимость на задачах с распределением на входах транзитивна.

*Доказательство.* Пусть функция  $f$  сводит  $(A, \mu)$  к  $(B, \nu)$ , а функция  $g$  —  $(B, \nu)$  к  $(C, \kappa)$ . Покажем, что композиция  $g \circ f$  сводит  $(A, \mu)$  к  $(C, \kappa)$ . Действительно, регулярность очевидна, сводимость по Карпу проверяется стандартным образом, а доминирование следует из такой цепочки:

$$\sum_{x \in f^{-1}(g^{-1}(z))} p_\mu(x) \leq q_1(n) \sum_{y \in g^{-1}(z)} p_\nu(y) \leq q_1(n) q_2(n) p_\kappa(z).$$

□

**Утверждение 10.14.** Если задача  $(A, \mu)$  сводится к  $(B, \nu)$ , которая лежит в  $\mathbf{distP}$ , то и  $(A, \mu)$  лежит в  $\mathbf{distP}$ .

*Доказательство.* Пусть функция  $f$  сводит  $(A, \mu)$  к  $(B, \nu)$ , а машина  $N$  распознаёт язык  $B$ . Выберем константы  $\varepsilon$  и  $C$ , так что для всех  $m$ :

$$\mathbb{E}_{\nu_m} \frac{(\text{time}_N(y))^\varepsilon}{m} < C. \quad (10.3)$$

Тогда построим машину  $M$ , которая запускает  $N$  на входе  $f(x)$ . В таком случае время работы  $M$  складывается из времени вычисления  $f(x)$  и времени работы  $N(f(x))$ . Будем считать, что для  $x$  длины  $n$  длина  $f(x)$  составит  $s(n)$ , а вычисление  $f(x)$  потребует не больше  $t(n)$  шагов. В таком случае

$$\text{time}_M(x) \leq \text{time}_N(f(x)) + t(n). \quad (10.4)$$

Будем считать, что  $\varepsilon$  с самого начала выбрано так, что

$$t(n)^\varepsilon < n. \quad (10.5)$$

Согласно утверждению 10.10 достаточно доказать ограниченность ожидания  $\mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon}{q(n)s(n)}$ . Она получается такой цепочкой (не)равенств:

$$\begin{aligned} \mathbb{E}_{\mu_n} \frac{(\text{time}_M(x))^\varepsilon}{q(n)s(n)} &\leq \mathbb{E}_{\mu_n} \frac{(\text{time}_N(f(x)) + t(n))^\varepsilon}{q(n)s(n)} < \mathbb{E}_{\mu_n} \frac{(\text{time}_N(f(x)))^\varepsilon + t(n)^\varepsilon}{q(n)s(n)} < \\ &< \mathbb{E}_{\mu_n} \frac{(\text{time}_N(f(x)))^\varepsilon}{q(n)s(n)} + O(1) = \sum_{x \in \{0,1\}^n} p_\mu(x) \frac{(\text{time}_N(f(x)))^\varepsilon}{q(n)s(n)} + O(1) = \\ &= \sum_{y \in \{0,1\}^{s(n)}} \sum_{x \in f^{-1}(y)} p_\mu(x) \frac{(\text{time}_N(y))^\varepsilon}{q(n)s(n)} + O(1) \leq \sum_{y \in \{0,1\}^{s(n)}} q(n)p_\nu(y) \frac{(\text{time}_N(y))^\varepsilon}{q(n)s(n)} + O(1) = \\ &= \mathbb{E}_{\nu_{s(n)}} \frac{(\text{time}_N(y))^\varepsilon}{s(n)} + O(1) < C + O(1). \end{aligned}$$

Здесь первое неравенство следует из (10.4), второе — из неравенства  $(a+b)^\varepsilon < a^\varepsilon + b^\varepsilon$ , третье — из (10.5), первое равенство есть определение матожидания, второе равенство есть перегруппировка слагаемых, четвёртое неравенство выполнено по доминированию, последнее равенство есть снова определение матожидания, а последнее неравенство следует из (10.3). Таким образом, нужная оценка доказана, и  $(A, \mu) \in \mathbf{distNP}$ .  $\square$

### 10.3.2 Полные задачи в $\mathbf{distNP}$

Теперь мы докажем, что в  $\mathbf{distNP}$  есть полная задача. Как и раньше, полной будет задача  $\mathbf{TMSAT}$ , нужно только добавить правильное распределение и построить правильную сводимость (для обычной сводимости может быть не выполнено свойство доминирования).

Вначале напомним формальное определение  $\mathbf{distNP}$ -полноты:

**Определение 10.15.** Задача с распределением на входах  $(B, \nu)$  называется  $\mathbf{distNP}$ -полной, если она лежит в  $\mathbf{distNP}$  и любая другая задача  $(A, \mu) \in \mathbf{distNP}$  к ней полиномиально сводится.

Теперь определим задачу, полноту которой будем доказывать:

**Определение 10.16.** Задачей  $\text{distTMSAT}$  будем называть пару  $(\text{TMSAT}, \tau)$ , где  $\text{TMSAT} = \{(M, x, 1^t) \mid \text{для недетерминированной машины } M \text{ и некоторого сертификата } y \text{ выполнено } M(x, y) = 1, \text{ при этом вычисление длится не больше } t \text{ шагов}\}$ . Распределение  $\tau$  строится так: вначале равновероятно выбирается описание  $M$  длины не больше  $\log n$ , затем  $t$  выбирается равновероятно от 1 до  $n - |M|$  и, наконец,  $x$  выбирается равномерно среди всех слов длины  $n - t - |M|$ .

**Замечание 10.17.** Формально запись  $(M, x, 1^k)$  будет немного длиннее  $n$ , так как нужно предусмотреть место на разделителе. Однако легко сделать так, чтобы и место на разделителе уложилось в  $n$ . Договоримся о конкретном способе: машину  $M$  будем записывать с помощью *беспрефиксного кода*, например удваивая каждый бит, а в конце написав 01. Затем напишем  $1^t 0$ . А затем уже  $x$ . Чтобы было удобнее доказывать полиномиальную вычислимость, сделаем ещё так, чтобы описание  $M$  занимало всегда одну и ту же длину. Для этого можно перед этим описанием также поставить 01, а остаток занять нулями. Также мы будем считать, что любое описание такой длины задаёт какую-то машину (если не задаёт — будем считать, что задаёт нигде не определённую).

**Утверждение 10.18.** Задача  $(\text{TMSAT}, \tau)$  лежит в  $\text{distNP}$ .

*Доказательство.* Утверждение  $\text{TMSAT} \in \text{NP}$  мы уже доказывали. Докажем полиномиальную вычислимость  $\tau$ . Напомним, что нам нужно вычислить вероятность возникновения тройки не больше  $(M, x, 1^t)$  в лексикографическом порядке. С нашим кодированием сравнивать тройки очень легко:  $(N, z, 1^s) \leq (M, x, 1^t)$ , если либо  $N <_{\text{lex}} M$ , либо  $N = M$ ,  $s < t$ , либо  $N = M$ ,  $s = t$ ,  $z \leq_{\text{lex}} x$ . Вероятность каждого из этих событий легко подсчитать, что и доказывает полиномиальную вычислимость.  $\square$

Однако сводить любую задачу к  $(\text{TMSAT}, \tau)$  нужно с осторожностью. Если использовать стандартную процедуру, то условие доминирования может быть не соблюдено. Ведь исходное распределение может быть каким угодно, и отдельные «плохие» входы могут иметь гораздо бóльшую вероятность, чем при распределении  $\tau$ . Идея доказательства состоит в том, чтобы «размазать» эти пики.

## 10.4 Задачи с квазиравномерными распределениями

Мы видели, что задача  $\text{dist3COL}$  лежит в  $\text{distP}$  и потому не может быть  $\text{distNP}$ -полной, если только  $\text{distNP}$  не вложено в  $\text{distP}$ . Оказывается, похожий результат имеет место для любой  $\text{NP}$ -задачи с достаточно равномерным распределением.

**Определение 10.19.** Вероятностный ансамбль  $\mu$  называется *квазиравномерным*, если для некоторого  $\varepsilon > 0$  при всех  $n$  и всех  $x$  длины  $n$  выполнено  $p_\mu(x) \leq 2^{-|x|^\varepsilon}$ . Говорят также, что в этом случае *мин-энтропия* распределения  $\mu_n$  не меньше  $n^\varepsilon$ , обозначается  $H_\infty(\mu_n) \geq n^\varepsilon$ .

### 10.5 Пять миров Импальяццо

### 10.6 Исторические замечания и рекомендации по литературе

### 10.7 Задачи и упражнения

# Глава 11

## Сложность задач поиска

*Мы – это мы; пусть время и судьба  
Нас подточили, но закал все тот же,  
И тот же в сердце мужественный пыл —  
Дерзать, искать, найти и не сдаваться!*

Альфред Теннисон, *Улисс* (пер. Г.М.Кружкова)

В математике длительное время велись дискуссии о приемлемости неконструктивных доказательств существования: если доказано, что объект существует, но сам объект не предъявлен, то можно ли это доказательство признать «настоящим»? В итоге большинство учёных согласны, что можно, но в практических задачах такой подход часто недопустим: нужно не просто убедиться, что решение существует, а найти его. В этой главе мы изучим, как измерять сложность задач поиска.

### 11.1 Что такое задача поиска

Напомним, что мы называем задачей поиска. Пусть задан некоторый предикат  $V(x, y)$ . Тогда задачей поиска называется задача нахождения по входу  $x$  такого  $y$ , что  $V(x, y) = 1$ , либо указания, что таких  $y$  нет. Класс задач поиска, для которых предикат  $V$  вычислим за полиномиальное от длины  $x$  время, назовём **FNP**. Как мы уже выяснили в разделе 3.6.2, для **NP**-полных задач задачи поиска и распознавания эквивалентны, из чего можно сделать следующий вывод:

**Теорема 11.1.**  $P = NP$  тогда и только тогда, когда  $FP = FNP$ .

*Доказательство.* Вложение  $FNP \subset FP$  следует из  $P = NP$  в силу теоремы 3.30. Обратное вложение выполнено всегда: можно рассмотреть  $V(x, y)$ , равный единице тогда и только тогда, когда  $y = f(x)$ .<sup>1</sup> Импликация в другую сторону верна просто потому, что задача распознавания есть составная часть задачи поиска.  $\square$

Для (предположительно) не **NP**-полных задач задача поиска может быть как сложнее задачи распознавания, так и такой же сложности в смысле сводимости по Куку. Так, задача проверки, составное ли число, решается за полиномиальное время АКС-алгоритмом, однако соответствующая задача поиска —

<sup>1</sup>Чтобы определения сошлись, нужно рассматривать **FP** как класс многозначных функций, таких что достаточно найти только одно значение, либо указать, что значений нет вообще.

разложение на множители — предположительно за полином не решается. С другой стороны, для задачи об изоморфизме графов задача поиска сводится к задаче распознавания (см. упр. 11.1).

Для сравнения задач по сложности внутри **FNP** мы будем использовать такую сводимость, похожую на сводимость по Левину:

**Определение 11.2.** Задача поиска  $V$  сводится к задаче поиска  $W$ , если существуют две полиномиально вычислимые функции  $f$  и  $g$  (вторая может возвращать символ ошибки  $\perp$ ), такие что:

- Если  $\exists y V(x, y) = 1$ , то  $\exists z W(f(x), z) = 1$ ;
- Если  $W(f(x), y) = 1$  и  $g(x, y) \neq \perp$ , то  $V(x, g(x, y)) = 1$ ;
- Если  $W(f(x), y) = 1$  и  $g(x, y) = \perp$ , то  $\forall y V(x, y) = 0$ .

## 11.2 Класс **TFNP** и его подклассы

*Кто весел — тот смеётся,  
Кто хочет — тот добьётся,  
Кто ищет — тот всегда найдёт!*

Василий Лебедев-Кумач, *Весёлый ветер*

Особенно ярко отличие задач поиска от задач распознавания проявляется в тех случаях, когда задача распознавания тривиальна: нужный объект всегда существует. Соответствующие задачи поиска называются *тотальными*, а класс таких задач называется **TFNP**.

**Определение 11.3.** Классом **TFNP** называется класс задач поиска, таких что для любого  $x$  существует  $y$ , такой что  $V(x, y) = 1$ .

Очевидно, что  $\mathbf{FP} \subset \mathbf{TFNP} \subset \mathbf{FNP}$ . Оба вложения предположительно являются строгими: если  $\mathbf{FP} = \mathbf{TFNP}$ , то  $\mathbf{P} = \mathbf{NP} \cap \mathbf{coNP}$ , а если  $\mathbf{TFNP} = \mathbf{FNP}$ , то  $\mathbf{NP} = \mathbf{coNP}$  (см. задачи 11.3 и 11.4). Тем не менее, в классе **TFNP** затруднительно найти полные задачи, которые указали бы его место между **FP** и **FNP**. Дело в том, что этот класс семантический, а не синтаксический: по описанию предиката  $V$  непонятно, будет ли задача поиска тотальной.<sup>2</sup> Поэтому, как и в случае с вероятностными классами, надеяться на полные задачи не стоит.

Но если в классе нет полных задач, то как структурировать его по сложности? В случае с **BPP** были другие параметры: величина ошибки и число случайных битов, а что мерить в задаче поиска? Ответ дал Христос Пападимитриу в статье [104]: нужно посмотреть, каковы причины того, что решение всегда есть. Известно не так много способов доказательства существования нужного объекта, и каждому из них можно сопоставить свой класс задач поиска. Сначала перечислим общие идеи, а потом дадим строгие определения.

- Конструктивный способ: некоторый алгоритм, позволяющий найти нужный объект. Этому способу соответствует класс **FP**.

<sup>2</sup>Если не ограничивать время работы, то множество таких предикатов и вовсе неразрешимо. Более того, и оно, и его дополнение не перечислимы.

- Локальная оптимизация: любая функция на конечном множестве достигает локального максимума (разумеется, и глобального тоже, но его так просто не найти). В частности, если есть процедура, позволяющая увеличить значение функции, то последовательным применением этой процедуры можно прийти в локальный максимум. На этом соображении основан класс **PLS** (polynomial local search).
- Принцип Дирихле: любая функция, у которой область значений содержит меньше элементов, чем область определения, имеет коллизию. На этом соображении основан класс **PPP** (polynomial pigeonhole principle).
- Лемма о рукопожатиях (лемма [A.4](#)): в любом неориентированном графе количество вершин нечётной степени чётно. Поэтому если дана одна вершина нечётной степени, то должна найтись и другая. Возникает класс **PPA** (polynomial parity argument).
- Вариация предыдущего соображения для ориентированных графов: сумма всех входящих степеней вершин равняется сумме всех исходящих. Поэтому если дана одна несбалансированная вершина, то должна найтись и другая. Соответствующий класс называется **PPAD** (polynomial parity argument, directed version).
- Ещё одна вариация: в любом ориентированном графе без циклов найдётся сток. Здесь появляется класс **PPADS** (polynomial parity argument, directed version, sink).
- И ещё одна вариация: если в ориентированном графе найдётся вершина, баланс которой не делится на  $p$ , то найдётся другая вершина с тем же свойством. На этом основан класс **PPA**- $p$ . (Названия сложились исторически, и не всегда отражают ни наличие соображения чётности, ни опору на ориентированный или неориентированный случай).

### 11.2.1 Класс PLS

**Определение 11.4.** Классом **PLS** называется класс задач поиска, для которых заданы такие полиномиальные алгоритмы:

- Проверка на допустимость решения:  $W(x, y) = 1$ , если  $y$  есть допустимое решение для задачи  $x$ ;
- Поиск начального допустимого решения:  $f(x) = y$ , т.ч.  $W(x, y) = 1$ ;
- Функция издержек  $c(x, y)$  (с числовыми значениями);
- Функция перечисления соседей:  $N(x, y)$  есть (полиномиальный) список допустимых решений, соседних с  $y$  (при этом от отношения соседства не требуется ни симметричности, ни даже рефлексивности);
- Процедура улучшения:  $g(x, y) = z$ , т.ч.  $z \in N(x, y)$  и  $c(x, z) < c(x, y)$ , либо  $g(x, y) = \top$ , если такого  $z$  нет.

Требуется по входу  $x$  найти допустимое  $y$ , оптимальное в своей окрестности. Иными словами,  $V(x, y) = 1$ , если  $W(x, y) = 1$  и  $g(x, y) = \top$ .

Легко увидеть, что  $\mathbf{FP} \subset \mathbf{PLS}$  (см. задачу 11.5). Ясно также, что  $\mathbf{PLS} \subset \mathbf{TFNP}$ : если запустить спуск  $y_0 = f(x)$ ,  $y_{i+1} = g(x, y_i)$ , то он не может быть бесконечным в силу того, что  $c(x, y_{i+1}) < c(x, y_i)$ , а потому  $g(x, y_j) = \top$  для какого-то  $j$ . Зато такой спуск может быть экспоненциально длинным, и его прямое применение может не дать ответ за полиномиальное время. Более того, поиск результата такого спуска для фиксированной начальной точки может быть  $\mathbf{NP}$ -трудным (см. задачу 11.6). Поэтому, если  $\mathbf{P} \neq \mathbf{NP}$ , то не может существовать универсальной процедуры, на которую можно заменить постепенный спуск, подобно тому как метод эллипсоидов Хачияна позволяет заменить симплекс-метод Данцига в задачах линейного программирования. Но если требуется найти хотя бы какой-нибудь локальный оптимум, а не конкретный, то в принципе возможен какой-то полиномиальный алгоритм. Гипотеза заключается в том, что такого алгоритма нет, т.е.  $\mathbf{FP} \neq \mathbf{PLS}$ .

В классе  $\mathbf{PLS}$  есть естественная полная проблема: машина получает на вход описание всех алгоритмов, а также  $x$ . Требуется найти  $y$ , такой что  $g(x, y) = \top$ . Любая другая к ней сводится очевидным образом: просто все описания алгоритмов приписываются к  $x$ , а ответ будет тем же самым. Поэтому часто  $\mathbf{PLS}$  и другие классы задач поиска определяют как классы всех задач, которые сводятся к некоторой фиксированной. Мы так делать не будем, чтобы не затуманивать суть. Кроме того, если алгоритмы становятся частью входа, то можно их заменить на схемы полиномиального размера. С точки зрения полноты такая замена ни на что не повлияет, но задачу со схемами можно считать более общей.

### 11.2.2 Класс PPA

Класс  $\mathbf{PPA}$  состоит из задач, которые можно сформулировать так: по одной вершине нечётной степени в графе найти другую вершину нечётной степени. Чтобы задача не решалась простым перебором, граф должен быть сверхполиномиального размера. Но при этом у каждой вершины должна быть полиномиальная степень и, более того, список соседей каждой вершины можно вычислить полиномиальным алгоритмом. Чтобы не возиться с проверкой на симметричность, формальное определение делается таким:

**Определение 11.5.** Рассмотрим полиномиальный алгоритм  $N$ , который получает на вход  $x \in \{0, 1\}^n$  и возвращает либо  $\perp$ , либо полиномиальный список других элементов  $\{0, 1\}^n$ .<sup>3</sup> Такой алгоритм задаёт граф, вершинами которого являются элементы  $x$ , такие что  $N(x) \neq \perp$ , а рёбрами — такие пары  $(x, y)$ , что  $y \in N(x)$  и  $x \in N(y)$ . Классом  $\mathbf{PPA}$  называется класс задач поиска, в которых в графе, заданном вышеописанным способом, по одной вершине нечётной степени требуется найти другую вершину нечётной степени. Формально предикат можно определить так:  $V(x, y) = 0$ , если  $x$  — вершина графа нечётной степени, а  $y$  — не вершина графа,  $y = x$  или  $y$  есть вершина чётной степени, а во всех остальных случаях  $V(x, y) = 1$ .

Как и класс  $\mathbf{PLS}$ , класс  $\mathbf{PPA}$  находится между  $\mathbf{FP}$  и  $\mathbf{TFNP}$  (см. упр. 11.7). Вопрос о соотношении между  $\mathbf{PPA}$  и  $\mathbf{PLS}$  открыт: известны оракулы, при которых  $\mathbf{PPA} \not\subset \mathbf{PLS}$  [99] и при которых  $\mathbf{PLS} \not\subset \mathbf{PPA}$  [26].

<sup>3</sup>Формально список может быть пустым, и это не то же самое, что  $\perp$ , но на дальнейшее определение это различие не влияет, см. упр. 11.7.



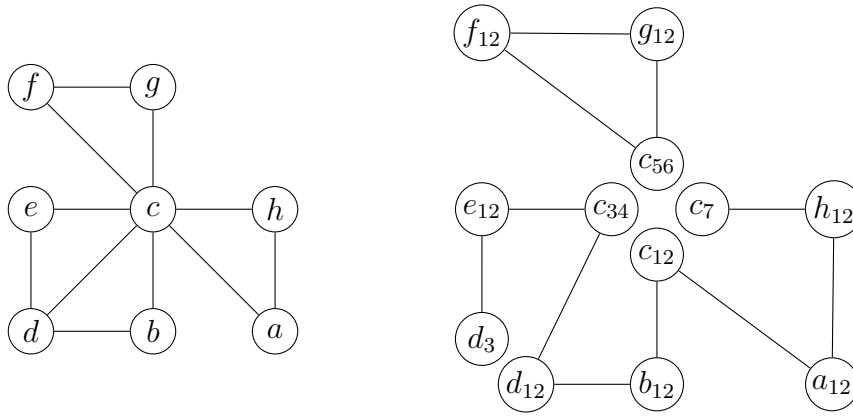


Рис. 11.1: Преобразование графа с произвольными степенями вершин в граф со степенями не выше 2. Вершинам нечётной степени  $c$  и  $d$  соответствуют висячие вершины  $c_7$  и  $d_3$ .

Оказывается, класс **PPA** не изменится, если ограничить степень всех вершин двойкой. Ясно, что в таком случае граф превратится в совокупность циклов и цепочек, а задача будет состоять в том, чтобы по одному концу цепочки найти другой конец цепочки (не обязательно той же самой).

**Теорема 11.6** ([104]). *Ограничим длину списка, который возвращает алгоритм  $N$  из определения 11.5, двойкой. Назовём **PPA'** класс задач поиска, в которых в графе, заданном алгоритмом  $N$ , по одной висячей вершине нужно найти другую висячую вершину. Тогда **PPA'** = **PPA**.*

*Доказательство.* Ясно, что **PPA'**  $\subset$  **PPA**: мы сузили класс алгоритмов  $N$ . В обратную сторону нужно доказать следующее: любую задачу из **PPA** можно задать не только исходным графом, но и каким-то другим, с ограниченной степенью. Для этого исходный граф преобразуется следующим образом: вершина  $x$  степени  $k$  превращается в  $\lceil \frac{k}{2} \rceil$  вершин:  $x_{12}, x_{34}, \dots$ . Последняя вершина будет либо  $x_{k-1,k}$ , если  $k$  чётно, либо  $x_k$ , если  $k$  нечётно. Если в исходном графе было ребро  $(x, y)$ , причём  $y$  было  $i$ -м соседом  $x$  (в лексикографическом порядке), а  $x$  было  $j$ -м соседом  $y$ , то проводится ребро между новой вершиной  $x$  с  $i$  в индексе (т.е.  $x_{i-1,i}$ , если  $i$  чётно,  $x_{i,i+1}$ , если  $i$  нечётно и не максимальное, или  $x_i$ , если  $i$  нечётно и максимальное) и новой вершиной  $y$  с  $j$  в индексе (аналогично  $y_{j-1,j}$ ,  $y_{j,j+1}$  или  $y_j$ ), см. пример на рис. 11.1. Ясно, что новые вершины с двумя индексами будут иметь степень 2, а новые вершины с одним индексом будут иметь степень 1. Каждой нечётной вершине исходного графа соответствует ровно одна новая вершина с одним индексом, а каждой чётной их не соответствует вовсе. Поэтому задача поиска по нечётной вершине исходного графа другой нечётной вершины эквивалентна задаче поиска по висячей вершине нового графа другой висячей вершины.  $\square$

Как и в классе **PLS**, в классе **PPA** есть естественная полная задача: машина получает на вход алгоритм  $N$ , или, если угодно, схему полиномиального размера, вычисляющую функцию  $N$ , и вершину  $x$  нечётной степени в графе, построенном по  $N$ , а должна вернуть другую вершину нечётной степени. Часто класс **PPA** определяют как класс всех задач, которые сводятся к данной.

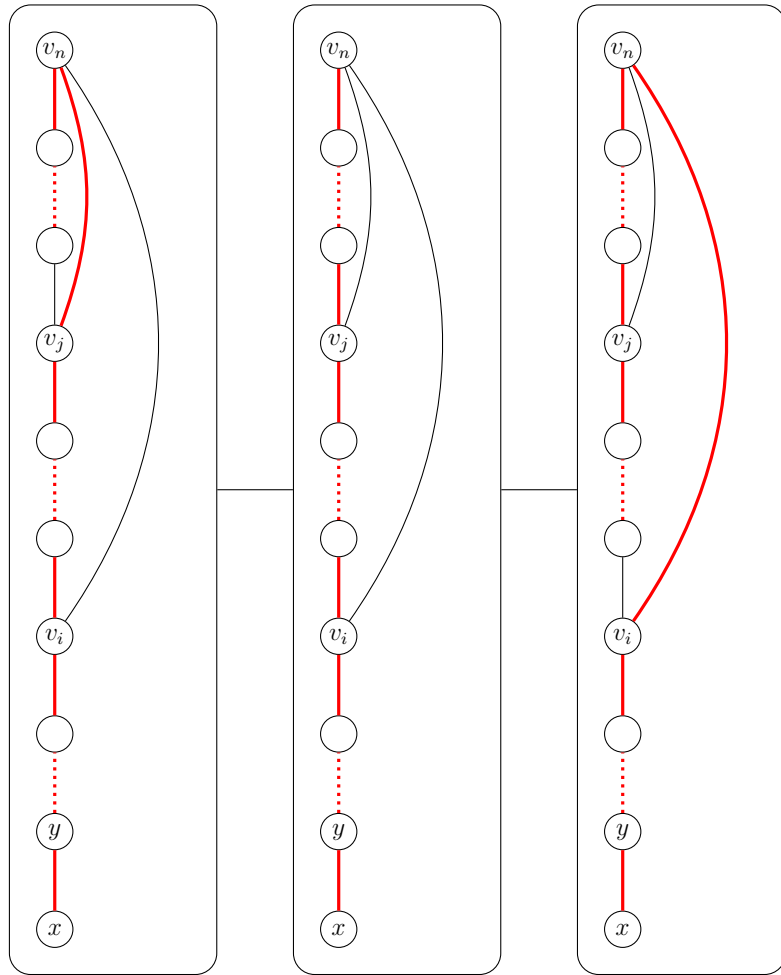


Рис. 11.2: Преобразование гамильтоновых путей, начинающихся с ребра  $(x, y)$ .

Интересным примером задачи из **PPA** является задача **SECONDHAMCYCLE**: дан кубический граф (т.е. граф, степени всех вершин которого равны трём), и некоторый гамильтонов цикл в нём, содержащий выделенное ребро. Требуется найти другой гамильтонов цикл, содержащий то же самое ребро. В силу теоремы Смита [136] число таких гамильтоновых циклов чётно, мы докажем принадлежность **PPA** при помощи алгоритма Томасона [134].

**Теорема 11.7.** *Задача поиска **SECONDHAMCYCLE** лежит в **PPA**.*

*Доказательство.* Мы построим граф со степенями вершин не выше 2, висячие вершины которого будут соответствовать гамильтоновым циклам, содержащим  $(x, y)$ . Вершинами нового графа будут гамильтоновы *пути*, начинающиеся с  $(x, y)$ . Циклам будут соответствовать те из них, что заканчиваются в одном из двух других соседей  $x$  (назовём их  $z$  и  $t$ ). Теперь определим, как в новом графе проводятся рёбра. Пусть есть путь  $(v_1 = x, v_2 = y, \dots, v_n)$ . Степень  $v_n$  равна трём, поэтому кроме  $v_{n-1}$  есть ещё два соседа  $v_i$  и  $v_j$ ,  $i < j$ . Если  $i > 1$  (т.е.  $v_i \neq x$ ), то в качестве соседей исходного пути будем рассматривать пути  $(v_1, \dots, v_i, v_n, v_{n-1}, \dots, v_{i+1})$  и  $(v_1, \dots, v_j, v_n, v_{n-1}, \dots, v_{j+1})$  (см. рис. 11.2). Если же  $i = 1$ , то соседом будет только второй путь. Таким образом, у каждой вершины действительно не больше двух соседей, а вершины с одним соседом соответствуют гамильтоновым циклам. Ясно также, что преобразование вычисляется за полиномиальное время, а поэтому задача лежит в **PPA**.

□

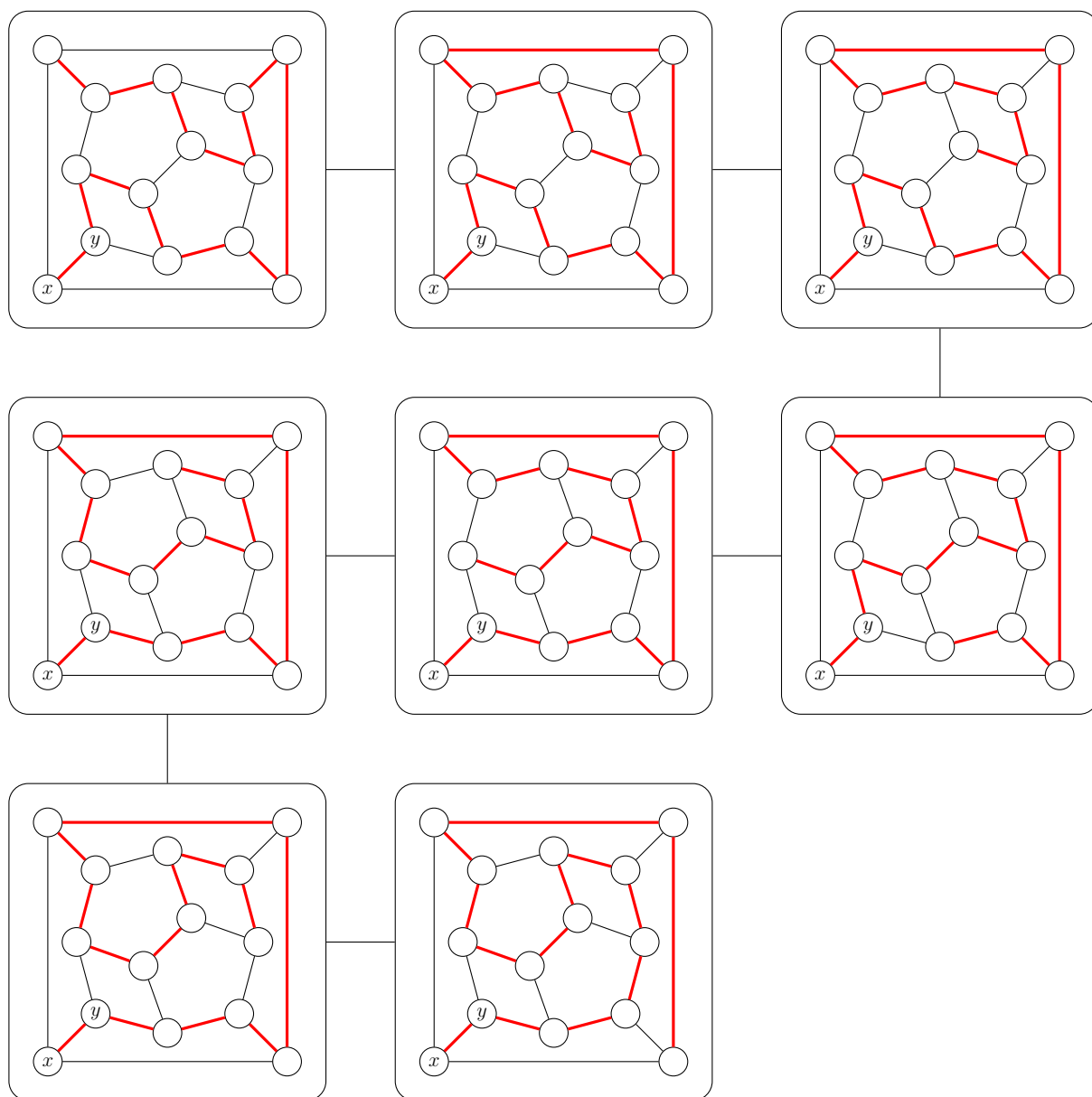


Рис. 11.3: Цепочка преобразований гамильтоновых путей, начинающихся с ребра  $(x, y)$  в графе из 14 вершин.

Показано [27, 87], что алгоритм Томасона может приводить к экспоненциально длинным блужданиям. Иных способов полиномиального поиска другого гамильтонова цикла пока не известно. Тем не менее, **РРА**-полнота задачи **SECONDHAMCYCLE** также не доказана, её точная классификация остаётся открытой проблемой. Обзор **РРА**-полных задач можно найти в работе [39].

### 11.2.3 Классы **PPAD** и **PPADS**

*Пойдѣшь ты дорогою длинною,  
Дорогой длинною-длинною,  
Минуя страну журавлиную,  
Минуя страну лебединую.  
Зимою и летом ты будешь в пути,  
Пока не найдѣшь  
То, что надо найти.*

Овсей Дриз, *Дороги*

Класс **PPAD** основан на следующем принципе: если в ориентированном графе есть несбалансированная вершина, то есть и другая несбалансированная вершина. Многие источники утверждают, что в этом случае также легко перейти от общего графа к графу степени не выше двух (в данном случае и входящая, и исходящая степени должны быть не выше 1). На проверку оказывается, что рассуждение из теоремы 11.6 не срабатывает (см. упр. 11.13), а другого рассуждения придумать также не удаётся. Поэтому мы определим **PPAD** через граф малой степени: именно для такой формулировки получены все результаты о полноте. Поскольку граф теперь ориентированный, проверка на симметричность больше не нужна. Зато нужно уметь считать входящую степень.

**Определение 11.8.** Рассмотрим два полиномиальных алгоритма  $N$  и  $P$ , преобразующие элементы  $\{0, 1\}^n$  в элементы  $\{0, 1\}^n$ . Они определяют граф, в котором есть ребро  $(x, y)$ , если  $y = N(x)$  и  $x = P(y)$ . Ясно, что в таком графе входящая и исходящая степень каждой вершины не больше 1, поэтому он представляется в виде объединения циклов, цепочек и изолированных вершин. Классом **PPAD** называется класс задач поиска, в которых в орграфе, заданном таким образом, по одному источнику нужно найти либо сток, либо другой источник. Формально предикат можно определить так:  $V(x, y) = 0$ , если  $x$  — источник, а  $y$  — либо изолированная вершина, либо промежуточная, либо равен  $x$ , а во всех остальных случаях  $V(x, y) = 1$ .

Если в качестве искомого объекта нужен обязательно сток, то возникает класс **PPADS**.

**Определение 11.9.** Классом **PPADS** называется класс задач поиска, в которых в орграфе, заданном алгоритмами  $N$  и  $P$ , по источнику нужно найти некоторый сток.

Поскольку **PPAD** и **PPADS** отличаются только тем, что в **PPAD** больше искомым объектам, получаем вложение  $\mathbf{PPAD} \subset \mathbf{PPADS}$ . Обе функции в сводимости будут попросту тождественными. Также легко понять, почему  $\mathbf{PPAD} \subset \mathbf{PPA}$ : достаточно забыть про ориентацию рёбер в графе. Получаем цепочки вложений  $\mathbf{FP} \subset \mathbf{PPAD} \subset \mathbf{PPADS} \subset \mathbf{TFNP}$  и  $\mathbf{FP} \subset \mathbf{PPAD} \subset \mathbf{PPA} \subset \mathbf{TFNP}$ .

В отличие от **PPAD**, для **PPADS** работает теорема о преобразовании графа произвольной степени в граф степени не выше 2.

**Теорема 11.10.** Пусть алгоритмы  $N$  и  $P$  возвращают не один элемент, а список полиномиальной длины, либо  $\perp$ . Составим граф, вершинами которого будут все вершины  $x$ , для которых  $P(x) \neq \perp$  и  $N(x) \neq \perp$ , а рёбрами — такие пары  $(x, y)$ , что  $y \in N(x)$  и  $x \in P(y)$ . Тогда задача поиска в таком графе по вершине с положительным балансом вершины с отрицательным балансом лежит в PPADS.

Доказательство остаётся в качестве упражнения 11.12.

Класс PPAD оказался богат на полные задачи, прежде всего связанные с конструктивными версиями теорем о неподвижных точках. Мы отложим их изучение, в том числе доказательство принадлежности к PPAD, до раздела 11.3.

### 11.2.4 Класс PPP

В русской традиции принцип Дирихле обычно рассказывают на примере кроликов и клеток: если  $k$  кроликов поместить в  $k - 1$  клетку, то хотя бы в одну клетку попадёт больше одного кролика. На английском языке говорят о голубях и гнёздах. Поэтому то же самое соображение называется pigeonhole principle. На его применении основан класс PPP.

**Определение 11.11.** Пусть задана полиномиально вычислимая функция  $f: \{0, 1\}^n \rightarrow \{0, 1\}^n$ . Тогда классом PPP называется множества задач поиска либо по некоторому  $z$  либо его прообразу, либо *коллизии*, т.е. такой пары  $(x, y)$ , что  $f(x) = f(y)$ .

Класс PPP связан с уже изученными классами так:

**Теорема 11.12.**  $\text{PPADS} \subset \text{PPP} \subset \text{TFNP}$ .

*Доказательство.* Вложение в TFNP, как всегда, очевидно. Задача о поиске стока решается по принципу Дирихле вот как: определим  $f(x)$  как  $N(x)$ , если  $P(N(x)) = x$ , и как  $x$  в противном случае. Без ограничения общности можно считать. Тогда изолированные вершины остаются на месте, у источников прообраза нет, у стоков прообраза ровно два, один из которых он сам, а у промежуточных вершин прообраз один, отличный от самой вершины. Если на вход дан источник, то алгоритм должен найти коллизию, которая и задаст сток.  $\square$

К сожалению, в классе PPP неизвестно естественных полных задач, не получающихся напрямую из принципа Дирихле. Одной из открытых проблем является классификация задачи EQUAL SUBSETS. Эта задача задана набором из  $n$  неотрицательных целых чисел, сумма которых меньше, чем  $2^n - 1$ . Требуется найти два подмножества с одинаковой суммой. Для этой задачи неизвестно ни полиномиального алгоритма, ни доказательства PPP-полноты.

## 11.3 Полнота в классе PPAD

На текущий момент класс PPAD оказался самым богатым на полные задачи. Именно сюда были классифицированы задачи, связанные с неподвижными точками.

### 11.3.1 Лемма Шпернера

### 11.3.2 Теорема Брауэра

### 11.3.3 Равновесие Нэша

### 11.3.4 Рыночные равновесия

## 11.4 Исторические замечания и рекомендации по литературе

В работе [19] в предположении  $\mathbf{EE} \neq \mathbf{NEE}$  показано, что существуют задачи из  $\mathbf{NP}$ , для которых поиск сложнее распознавания.

Класс  $\mathbf{TFNP}$  впервые определён в явном виде в статье Меджиддо и Пападимитриу [95]. Класс  $\mathbf{PLS}$  был определён ещё раньше, в статье [78]. Классы  $\mathbf{PPP}$ ,  $\mathbf{PPA}$  и  $\mathbf{PPAD}$  появились в основополагающей статье Пападимитриу [104],<sup>4</sup> там же была доказана полнота многих задач.

Теорема о чётном числе гамильтоновых циклов в кубическом графе впервые была опубликована в 1946 году в работе Уильяма Татта [136], где атрибутирована Седрику Смиту. Представленное здесь доказательство взято из работы Эндрю Томасона [134] — именно из него следует принадлежность к  $\mathbf{PPA}$ . В 1988 году в работе [32] впервые был поставлен вопрос о вычислительной сложности задачи о нахождении другого гамильтонова цикла. На этот вопрос ответили Адам Кравчук, построивший пример [87], на котором алгоритм Томасона экспоненциален, и Кэти Кэмерон [27], доказавшая эту оценку. Ещё один алгоритм, также экспоненциальный в худшем случае, предложен Томми Дженсенем [77].

$\mathbf{PPAD}$ -полнота поиска равновесия Нэша для 4 игроков была установлена Даскалакисом, Голдбергом и Пападимитриу в 2005 году (полная журнальная статья [36]), затем число игроков было уменьшено до трёх независимо в [30] и [37] и, наконец, до двух в [31].

## 11.5 Задачи и упражнения

**11.1. Поиск изоморфизма.** Докажите, что задача поиска изоморфизма между двумя графами сводится по Куку к задаче распознавания, есть ли изоморфизм.

**11.2. Наибольшая клика.** Лежит ли задача поиска наибольшей по числу вершин клики в  $\mathbf{TFNP}$ ?

**11.3. Альтернативное определение  $\mathbf{TFNP}$  ([95]).**

- а) Обозначим через  $\mathbf{F}(\mathbf{NP} \cap \mathbf{coNP})$  класс задач поиска, для которых дано два полиномиально вычислимых предиката  $V$  и  $W$ , таких что для любого  $x$  гарантируется существование либо такого  $y$ , что  $V(x, y) = 1$ , либо такого  $z$ , что  $W(x, z) = 0$ . Задача состоит в том, чтобы по входу  $x$  найти либо указанное  $y$ , либо указанное  $z$ . Докажите, что  $\mathbf{F}(\mathbf{NP} \cap \mathbf{coNP}) = \mathbf{TFNP}$ .

<sup>4</sup>Некоторые комментаторы полагают, что названием  $\mathbf{PPAD}$  Пападимитриу увековечил буквы своей фамилии.

б) Докажите, что если  $\mathbf{TFNP} \subset \mathbf{FP}$ , то  $\mathbf{NP} \cap \mathbf{coNP} = \mathbf{P}$ .

**11.4. Полнота в  $\mathbf{TFNP}$  (/95/).** Докажите, что  $\mathbf{NP} = \mathbf{coNP}$  тогда и только тогда, когда некоторая  $\mathbf{FNP}$ -полная задача  $W$  (относительно сводимости из определения 11.2) лежит в  $\mathbf{TFNP}$ .

**11.5.** Докажите, что  $\mathbf{FP} \subset \mathbf{PLS}$ .

**11.6. Поиск стандартного минимума (/78/).** Определим задачу поиска стандартного минимума для задач из  $\mathbf{PLS}$ : по допустимому решению  $y$  найти локальный оптимум  $y^*$ , к которому сойдётся стандартный алгоритм (т.е.  $y_0 = y$ ,  $y_{i+1} = g(x, y_i)$ ). Докажите, что такая задача является  $\mathbf{NP}$ -трудной.

**11.7. Анализ определения  $\mathbf{PPA}$ .**

- а) Докажите, что предикат  $V$  из определения 11.5, вычисляется за полиномиальное время.
- б) Докажите, что  $\mathbf{FP} \subset \mathbf{PPA} \subset \mathbf{TFNP}$ .
- в) Докажите, что класс  $\mathbf{PPA}$  не изменится, если отождествить  $\perp$  и пустой список соседей.

**11.8. Третий гамильтонов цикл.**

- а) Постройте пример кубического графа, общее количество гамильтоновых циклов в котором нечётно.
- б) Докажите, что в любом гамильтоновом кубическом графе хотя бы три гамильтоновых цикла.
- в) Докажите, что задача поиска двух других гамильтоновых циклов по кубическому графу и одному гамильтонову циклу в нём лежит в  $\mathbf{PPA}$ .

**11.9. Совершенные паросочетания в эйлеровом графе (/43/).** Докажите, что задача поиска по одному совершенному паросочетанию в эйлеровом графе другого совершенного паросочетания лежит в  $\mathbf{PPA}$ .

**11.10.  $\mathbf{PPA}$  через ориентированные графы.** Рассмотрим такую задачу поиска: дан ориентированный граф, требуется по одной вершине с нечётным балансом найти другую вершину с нечётным балансом. Докажите, что эта задача  $\mathbf{PPA}$ -полна.

**11.11. Класс  $\mathbf{PPA-p}$ .** Рассмотрим две задачи:

- В неориентированном двудольном графе по одной вершине, чья степень не сравнима с 0 по модулю  $p$ , найти другую такую вершину.
- В ориентированном графе по одной вершине, чей баланс не сравним с 0 по модулю  $p$ , найти другую такую вершину.

Докажите, что обе эти задачи лежат в  $\mathbf{TFNP}$  и сводятся друг к другу. (Класс задач, которые к ним сводятся, называется  $\mathbf{PPA-p}$ ).

**11.12. Альтернативное определение  $\mathbf{PPADS}$ .** Докажите теорему 11.10.

**11.13. Другая несбалансированная вершина.** Рассмотрим такой способ определения орграфов по двум схемам  $P$  и  $N$ : ребро  $(x, y)$  проводится, если

$y \in N(x)$  и  $x \in P(y)$ . Рассмотрим задачу поиска в таком графе по одной несбалансированной вершине другой несбалансированной вершины.

- а) Покажите, что попытка доказать принадлежность этой задачи к **PPAD** аналогично доказательству теоремы ?? терпит крах.
- б) Докажите, что эта задача лежит в **PPADS**.
- в) Докажите, что эта задача лежит в **PPA**.
- г) Докажите, что эта задача лежит в **PPA**- $p$  для любого  $p$ . (См. задачу 11.11)

**11.14. Задача о равных множествах.** Докажите формально, что EQUAL SUBSETS  $\in$  **PPP**. (Т.е. постройте функцию  $f$ ).



# Глава 12

## Интерактивные доказательства

Обычное математическое доказательство выглядит как некоторый текст, например, в статье или в письменной контрольной работе. Человек, который проверяет доказательство, читает этот текст и либо удостоверяется, что оно верно, либо находит ошибку. Такая же ситуация имеет место для формальных систем проверки доказательств, таких как Coq. Отличие состоит разве что в формализации записи доказательства и большей надёжности проверки. Однако доклад на семинаре или ответ на устном экзамене устроены по-другому: проверка доказательства имеет вид диалога, в процессе которого можно задавать уточняющие вопросы или, наоборот, пропускать некоторые части исходя из уже сказанного. Таким образом, доказательство становится *интерактивным*, зависящим от двух сторон. Выборочная проверка доказательства может значительно ускорить процесс, но также приводит к росту вероятности ошибки. В этой главе мы изучим интерактивные доказательства с точки зрения вычислительной сложности.

### 12.1 Интерактивные системы доказательств

Неформально интерактивное доказательство представляется как диалог между двумя сторонами: прувером (доказывающим)  $P$  и верификатором (проверяющим)  $V$ .<sup>1</sup> При этом обычно предполагается, что вычислительные ресурсы верификатора ограничены, а прувера — безграничны: он может получать значения даже невычислимых функций. Обе стороны получают на вход одно и то же слово, а также потенциально некоторые частные входы. Также стороны могут иметь доступ к случайным битам, могут быть и общие случайные биты. После запуска протокола происходит обмен сообщениями, по итогам которого верификатор говорит «да» или «нет». При этом важно, чтобы общие случайные биты выдавались порциями, так чтобы прувер заранее не знал, какими они будут в следующем раунде. Следующее определение — самое общее из возможных, в дальнейшем мы будем его так или иначе ограничивать. Пока что мы не определяем, как интерактивная система что-то вычисляет, распознаёт или доказывает, а определяем только саму систему.

**Определение 12.1.** Системой интерактивных доказательств называется пара из произвольной функции  $P: \{0, 1\}^* \rightarrow \{0, 1\}^*$  и полиномиального алгоритма  $V: \{0, 1\}^* \rightarrow \{0, 1\}^* \cup \{Y, N\}$ . *Протоколом работы* интерактивной системы на

---

<sup>1</sup>Участникам протокола часто дают имена, например, Мерлин и Артур [11], Алиса и Боб, Пегги и Виктор, Пэт и Ванна [94] и т.д.

общем входе  $x$ , частном входе прувера  $y$ , частном входе верификатора  $z$ , последовательности наборов общих случайных битов  $s = (s_1, s_2, s_3, \dots)$ , частных случайных битах прувера  $q$  и частных случайных битах верификатора  $r$  называется следующая последовательность сообщений:<sup>2</sup>

- $m_1 = V(x, z, s_1, r)$ ;
- $m_2 = P(x, y, s_1, q, m_1)$ ;
- $m_3 = V(x, z, s_1, s_2, r, m_1, m_2)$ ;
- $m_4 = P(x, y, s_1, s_2, q, m_1, m_2, m_3)$ ;
- ...
- $m_{2k-1} = V(x, z, s_1, s_2, \dots, s_k, r, m_1, m_2, \dots, m_{2k-2})$ ;
- $m_{2k} = P(x, y, s_1, s_2, \dots, s_k, q, m_1, m_2, \dots, m_{2k-1})$ ;
- ...

Цепочка продолжается, пока на каком-то этапе не будет  $m_{2k-1} \in \{Y, N\}$ . В этом случае говорят, что верификатор принял доказательство, если ответ  $Y$ , и не принял, если ответ  $N$ .<sup>3</sup> Ответ верификатора обозначают через  $V^{P(y)}(x, z)$ , подразумевается, что это случайная величина, полученная при равномерном и независимом распределении  $r, q, s$ . Если каких-то входов в изучаемой модели нет, то их опускают и в обозначениях.

На сертификаты для языков из **NP** можно смотреть как на неинтерактивное доказательство: прувер присылает сертификат, а верификатор проверяет, что он подходит. Оказывается, если добавить интерактивность, но не добавлять случайности, то ничего сверх **NP** получить не получится.

**Определение 12.2.** Назовём классом **dIP** множество языков  $A$ , для которых существует система интерактивных доказательств  $\langle P, V \rangle$  без дополнительных входов и случайных битов, такая что:

- Если  $x \in A$ , то  $V^P(x) = 1$ ;
- Если  $x \notin A$ , то для любого  $P^*$  (также без дополнительного входа и случайных битов) выполнено  $V^{P^*}(x) = 0$

**Теорема 12.3.** **dIP = NP.**

*Доказательство.* Вложение **NP**  $\subset$  **dIP** доказывается каноническим протоколом: прувер посылает сертификат, а верификатор его проверяет и даёт соответствующий ответ.

Обратно: при отсутствии случайности прувер может заранее вычислить все будущие ответы верификатора на любые свои сообщения и прислать весь будущий диалог одним пакетом. Формально пусть  $A \in \mathbf{dIP}$ . Рассмотрим такой **NP**-верификатор:  $W(x, \mathbf{m}) = W(x, m_1, \dots, m_{2k}) = 1$ , если  $V(x) = m_1$ ,  $V(x, m_1, m_2) = m_3, \dots, V(x, m_1, m_2, \dots, m_{2k-2}) = m_{2k-1}$  и  $V(x, m_1, m_2, \dots, m_{2k}) = Y$ . Если для

<sup>2</sup>Здесь диалог начинается с сообщения верификатора. Если требуется, чтобы начал прувер, то можно считать, что первое сообщение верификатора осталось пустым.

<sup>3</sup>В дальнейшем мы будем обозначать ответ верификатора через 1 и 0, пока что  $Y$  и  $N$  выбраны, чтобы не путать с обычными сообщениями.

некоторого  $P$  выполнено  $V^P(x) = 1$ , то и  $W(x, \mathbf{m} = 1$  для диалога с  $P$ . Если же  $W(x, \mathbf{m}^*) = 1$  для некоторого диалога  $\mathbf{m}^*$ , то для  $P^*$ , присылающего сообщения из этого диалога, будет выполнено  $V^{P^*}(x) = 1$ . Таким образом,  $x \in A \Leftrightarrow \exists \mathbf{m} W(x, \mathbf{m}) = 1$ . Отсюда  $A \in \mathbf{NP}$ , что и требовалось.  $\square$

## 12.2 Класс $\mathbf{IP}$

Теперь перейдём к такой модели: дополнительных входов по-прежнему нет, а случайные биты есть только у верификатора. В таком случае для каждого входа возникают две случайные величины: количество раундов до окончания протокола и ответ верификатора. Как и в обычных вероятностных вычислениях, возникают ошибки первого рода, когда правильный ответ отрицательный, а система возвращает положительный, и ошибки второго рода, когда всё наоборот. Хотелось, чтобы ошибки были достаточно маленькими, чтобы в итоге можно было установить правильный ответ.

**Определение 12.4.** Классом  $\mathbf{IP}[k(n)]$  называется множество таких языков  $A$ , для которых существует такая интерактивная система доказательств  $\langle P, V \rangle$  с частными случайными битами верификатора, что:

- При любом  $x$  диалог продолжается не более  $k(|x|)$  раундов;
- Если  $x \in A$ , то  $\Pr_r [V^P(x, r) = 1] \geq \frac{2}{3}$ ;
- Если  $x \notin A$ , то для любого  $P^*$  выполнено  $\Pr_r [V^{P^*}(x, r) = 1] \leq \frac{1}{3}$ .

Классом  $\mathbf{IP}$  называется  $\mathbf{IP}[\text{poly}] = \bigcup_{c=1}^{\infty} \mathbf{IP}[cn^c]$ .

Как обычно, пороги  $\frac{2}{3}$  и  $\frac{1}{3}$  весьма условны. В разделе 12.2.3 мы обсудим, в каких пределах можно эти пороги варьировать. Прямо из определения можно получить следующие вложения:

**Теорема 12.5.**  $\mathbf{NP} \subset \mathbf{IP} \subset \mathbf{PSPACE}$ .

*Идея доказательства.* Мы уже видели (теорема 12.3), что  $\mathbf{NP} = \mathbf{dIP}$ . Очевидно, что  $\mathbf{dIP} \subset \mathbf{IP}$ . Вложение  $\mathbf{IP} \subset \mathbf{PSPACE}$  доказывается аналогично игровой характеристике  $\mathbf{PSPACE}$  (см. задачу 12.2).  $\square$

Первым примером интерактивного доказательства для языка, про который неизвестна принадлежность к  $\mathbf{NP}$ , стал протокол для задачи о неизоморфизме графов. Но вначале изложим основную его идею метафорически.

### 12.2.1 Метафора

Боб — дальтоник, не отличающий синий цвет от красного («ему всё фиолетово»). Однажды он купил через интернет три пары красных носков, но ему доставили две пары красных и одну пару синих. По крайней мере, так было написано на упаковках. Теперь его сестра Алиса, обладающая полноцветным зрением, всё время поддразнивает, что он надел носки разного цвета. Он не знает, говорит она правду или дразнит просто так. Чтобы вывести её на чистую воду, он делает следующее: сначала показывает, какие носки он надел. Затем уходит в соседнюю комнату, кидает там монетку и при выпадении орла меняет носки местами, а при выпадении решки оставляет их на месте. Вернувшись к Алисе, он просит угадать, поменял он носки или нет. Если носки действительно

разные, то Алиса легко даст верный ответ. Если же носки на самом деле одинаковые, то Алиса будет вынуждена отвечать наудачу. Что бы она ни ответила, она угадает с вероятностью 50%. Если же Боб повторит процедуру 10 раз, то Алиса угадает все 10 раз с вероятностью меньше 0.1%, что даст достаточный уровень уверенности Бобу.

### 12.2.2 Задача о неизоморфизме графов

**Определение 12.6.** Языком GNI называется множество  $\{(G_0, G_1) \mid \text{графы } G_0 \text{ и } G_1 \text{ неизоморфны}\}$ .

**Теорема 12.7.**  $\text{GNI} \in \text{IP}$ .

Неформально протокол можно описать следующей метафорой. Виктор — дальтоник и не может отличить красный носок от зелёного. Его сестра Пегги обладает полноцветным зрением и дразнит Виктора, что он надел разные носки. Виктор хочет убедиться, действительно ли носки разные, или Пегги дразнится просто так. Виктор просит Пегги указать, какой носок красный, а какой зелёный. Виктор запоминает указание, уходит в другую комнату и либо меняет носки местами, либо оставляет. После этого он возвращается и снова просит указать, где какой носок, и так несколько раз. Если носки и правда разного цвета,<sup>4</sup> то она с лёгкостью каждый раз угадает. Если же носки на самом деле одинаковые, то в каждой попытке вероятность удачи будет равна лишь  $\frac{1}{2}$ . Перейдём к формальному рассмотрению на графах.

*Доказательство.* Рассмотрим следующий протокол: верификатор выбирает случайный бит  $b \in \{0, 1\}$  и случайную перестановку  $\sigma \in S_n$ , вычисляет  $H = \sigma(G_b)$  и посылает его пруверу. Прувер в ответ посылает бит  $c \in \{0, 1\}$ . Если этот бит совпал с  $b$ , то верификатор принимает доказательство, иначе отвергает его. Мы докажем, что если на самом деле  $G_0 \not\simeq G_1$ , то вероятность того, что верификатор примет доказательство, равна единице, а если на самом деле  $G_0 \simeq G_1$ , то такая вероятность не больше  $\frac{1}{2}$ . Чтобы добиться уменьшения вероятности ниже  $\frac{1}{3}$ , достаточно повторить протокол ещё раз с новыми случайными битами.

Действительно, пусть  $G_0 \not\simeq G_1$ . Тогда множества графов, изоморфных  $G_0$  и  $G_1$ , не пересекаются, т.к. изоморфность есть отношение эквивалентности. Получив  $H$ , прувер сможет установить, из какого множества он получен, и вернуть соответствующий ответ.

Теперь пусть  $G_0 \simeq G_1$ . В таком случае множество графов, изоморфных  $G_0$ , совпадает с множеством графов, изоморфных  $G_1$ . Но этого мало для ограничения вероятности успеха прувера. Важно, что случайная перестановка  $G_0$  и случайная перестановка  $G_1$  приводят к одному и тому же распределению на этом множестве. Это доказывается так: у всех графов, изоморфных  $G_b$ , одно и то же число автоморфизмов, и столько же есть изоморфизмов между любыми двумя из них. Таким образом, случайная перестановка что  $G_0$ , что  $G_1$  приводят к равномерному распределению на множестве всех графов, им изоморфных.  $\square$

### 12.2.3 Вариации с точностью и полнотой

Как и в классе BPP, пороги  $\frac{1}{3}$  и  $\frac{2}{3}$  выбраны совершенно произвольно. Как и в классе BPP, при помощи многократного повторения их можно сделать экс-

<sup>4</sup>Будем считать, что отличить носки иначе нельзя, например, левый и правый не отличаются.

пониженно близкими к 0 и 1 соответственно. Более того, количество раундов при таком повторении не увеличится: можно несколько запусков протокола выполнять параллельно, а не последовательно. Встаёт вопрос, можно ли их приблизить ещё сильнее. Если ошибку в вероятностных вычислениях сделать нулевой, то получатся классы **RP** и **coRP**. Для класса **IP** ситуация иная: если нижний предел превратить в 0, то получится класс **NP**, а вот верхний предел можно превратить в 1 с сохранением класса.

**Теорема 12.8.** *Заменим в определении **IP** « $\leq \frac{1}{3}$ » на « $= 0$ ». Тогда получившийся класс совпадёт с **NP**.*

*Доказательство.* Если для слов не из языка вероятность, что верификатор примет протокол, нулевая, то любой протокол общения, который принимается верификатором, вместе с использованными случайными битами будет сертификатом принадлежности к языку. Новый верификатор проверяет, что все сообщения верификатора в предъявленном протоколе получены в соответствии с программой старого верификатора, а в конце он возвращает 1.  $\square$

**Теорема 12.9.** *Заменим в определении **IP** « $\geq \frac{2}{3}$ » на « $= 1$ ». Тогда получившийся класс совпадёт с **IP**.*

*Идея доказательства.* У этого факта есть два доказательства, которые основаны на двух главных теоремах настоящей главы: теореме Шамира **IP** = **PSPACE** и теореме Голдвассер–Сипсера о моделировании интерактивных протоколов при помощи общих случайных битов. В первом случае конструкция интерактивного доказательства для языка из **PSPACE** фактически даёт доказательство с единичной полнотой. Во втором случае применяется самостоятельное рассуждение наподобие теоремы Лаутемана. После доказательства этих теорем мы вернёмся к вопросу о единичной полноте.  $\square$

## 12.3 Игры Артура–Мерлина

Дата: среда, 13 дек 89 19:42:14

От кого: merlin@cave

Кому: arthur

Сэр,

в отдельном сообщении посылаю Вам квадратную матрицу  $M$  порядка 103680300, состоящую из чисел  $\{-1, 0, 1, 2, 3\}$ . При помощи статьи L.G.Valiant, TCS 8 (1979), pp.189–201 Вы легко проверите, что перманент  $M$  ровно в  $2^{43200000}$  раз больше числа гамильтоновых циклов в <<графе рассадки>>.

Сообщите, когда сможете уделить мне полиномиальное время. После беседы Вы будете убеждены с точностью  $1-2^{-1000}$ , что этот перманент равен нулю.

Пожалуйста, подготовьте игральные кости и повторите вышмат, особенно схему Горнера.

Ваш, Мерлин

Ласло Бабаи, [12]

Перейдём к изучению интерактивных доказательств с общими случайными битами. Для такой ситуации Ласло Бабаи предложил [11] метафору, ставшую общепринятой: вычислительно ограниченный Артур ведёт диалог с магом

Мерлином. Мерлин, во-первых, может находить значение любой функции, а во-вторых, мгновенно узнаёт результаты бросаний монетки Артура. Единственное ограничение Мерлина состоит в том, что он не может узнать результатов будущих бросаний.

Формально можно определять классы  $\mathbf{AM}[k]$  через ограничение определения 12.1. Однако проще поступить другим образом:

**Определение 12.10.** Классом  $\mathbf{AM}[k]$  называется подкласс  $\mathbf{IP}[k]$ , где верификатор в качестве своего сообщения может посылать только очередную порцию своих случайных битов.

В задаче 12.5 предлагается установить эквивалентность двух подходов. Если у класса  $\mathbf{IP}$  не указано число раундов, значит, их полиномиальное число. С  $\mathbf{AM}$  ситуация иная: по умолчанию число раундов равно двум. Это возникло не просто так, а из совокупности следующих теорем:  $\mathbf{AM}[const] = \mathbf{AM}[2]$ , но  $\mathbf{AM}[\text{poly}(n)] = \mathbf{IP}[\text{poly}(n)]$ . В следующих разделах мы докажем эти факты.

### 12.3.1 Классы $\mathbf{MA}$ и $\mathbf{AM}$

Начнём с рассмотрения протоколов с общими битами, длящимися два раунда. В этом случае важно, кто отправляет первое сообщение: Мерлин или Артур. Интерактивности в этом случае почти нет: определения классов можно сформулировать через случайность и недетерминизм.

**Определение 12.11.** Классом  $\mathbf{MA}$  называется множество языков  $B$ , для которых существует полиномиальный алгоритм от трёх аргументов  $x$ ,  $s$  и  $r$ , такой что:

- Если  $x \in B$ , то  $\exists s \Pr_r [V(x, s, r) = 1] \geq \frac{2}{3}$ ;
- Если  $x \notin B$ , то  $\forall s \Pr_r [V(x, s, r) = 1] \leq \frac{1}{3}$ .

Таким образом, класс  $\mathbf{MA}$  похож на класс  $\mathbf{NP}$ , но только сертификат проверяется вероятностным алгоритмом.<sup>5</sup> По определению получается, что сертификат один и тот же для всех случайных исходов. В классе  $\mathbf{AM}$ , напротив, сертификат может зависеть от выпавших монеток.

**Определение 12.12.** Классом  $\mathbf{AM}$  называется множество языков  $B$ , для которых существует полиномиальный алгоритм от трёх аргументов  $x$ ,  $s$  и  $r$ , такой что:

- Если  $x \in B$ , то  $\Pr_r [\exists s V(x, s, r) = 1] \geq \frac{2}{3}$ ;
- Если  $x \notin B$ , то  $\Pr_r [\exists s V(x, s, r) = 1] \leq \frac{1}{3}$ .

Здесь, наоборот, вероятность добавляется поверх недетерминизма. Поэтому этот класс ещё называют  $\mathbf{BP} \cdot \mathbf{NP}$ . Ясно, что возможности Мерлина возрастают, т.к. он может выбрать сертификат в зависимости от случайных битов. Казалось бы, из этого очевидно следует, что  $\mathbf{MA} \subset \mathbf{AM}$ . Проблема в том, что у нечестного Мерлина также появляется больше возможностей. Поэтому вложение становится нетривиальным.

<sup>5</sup>Тут есть одна тонкость: вероятностный алгоритм всегда должен давать вероятность успеха либо  $\geq \frac{2}{3}$ , либо  $\leq \frac{1}{3}$ . Здесь в случае  $x \in B$  достаточно одного сертификата, при котором вероятность будет  $\geq \frac{2}{3}$ . См. также упр. 12.6.



**Теорема 12.13.**  $\mathbf{MA} \subset \mathbf{AM}$ .

*Доказательство.* Пусть  $B \in \mathbf{MA}$ . Применим стандартную амплификацию, запустив алгоритм  $V$  для многих разных независимых  $r$  (но одного и того же сертификата  $s$ ). За счёт этого для  $x \notin B$  вероятность успеха может быть снижена до  $\frac{1}{3 \cdot 2^{|s|}}$ , а для  $x \in B$  — повышена до  $1 - \frac{1}{3 \cdot 2^{|s|}}$ . Теперь при  $x \in B$  годится один и тот же сертификат для всех  $r$ . А при  $x \notin B$  общая доля случайных битов, к которым подойдёт хотя бы один сертификат, не превосходит числа сертификатов ( $2^{|s|}$ ), умноженного на долю подходящих к фиксированному сертификату ( $\leq \frac{1}{3 \cdot 2^{|s|}}$ ), что в результате будет не больше  $\frac{1}{3}$ , что и требовалось.  $\square$

Класс  $\mathbf{MA}$  естественно считать самым широким классом языков, для которых существуют неинтерактивные (допускающие публикацию) доказательства принадлежности. Действительно, от Мерлина требуется только сертификат  $s$ : для фиксированного сертификата проверка может осуществляться полиномиальным вероятностным алгоритмом без участия Мерлина.

**12.3.2 Уменьшение числа раундов**

По аналогии с  $\mathbf{MA}$  и  $\mathbf{AM}$  можно определять и другие классы:  $\mathbf{AMA}$ ,  $\mathbf{MAM}$ ,  $\mathbf{AMAM}$  и так далее. Число букв отвечает за число раундов, а сами буквы — за то, какая сторона первой посылает сообщение. На первый взгляд кажется, что возникает иерархия, подобная полиномиальной: на первом уровне лежат  $\mathbf{NP}$  и  $\mathbf{VPR}$ , на втором  $\mathbf{MA}$  и  $\mathbf{AM}$ , и т.д. Однако эта иерархия выше второго уровня не поднимается: вложение  $\mathbf{MA} \subset \mathbf{AM}$  позволяет доказать, что все высшие классы совпадают с  $\mathbf{AM}$ .

**Теорема 12.14** (О коллапсе  $\mathbf{AM}$ -иерархии, [11]). *Для любой константы  $k$  выполнено  $\mathbf{AM}[k] = \mathbf{AM}$ .*

*Доказательство частного случая.* Для примера докажем, почему  $\mathbf{AMA} = \mathbf{AM}$ . Идея заключается в том, что из  $\mathbf{MA} \subset \mathbf{AM}$  следует  $\mathbf{AMA} \subset \mathbf{AAM} = \mathbf{AM}$ . Действительно, наличие первого раунда со случайными битами почти не повлияет на рассуждение. Нужно только уточнить, что амплификация делается при помощи параллельных запусков, чтобы число раундов оставалось прежним. Более подробно: Артур выбирает случайные  $r_1, \dots, r_k$  и посылает их Мерлину, который в ответ присылает сертификаты  $s_1, \dots, s_k$ , после чего Артур несколько раз проверяет каждую пару  $(r_i, s_i)$ , каждый раз со свежими случайными битами  $q_{i1}, \dots, q_{ik}$ . Если общая доля успешных запусков больше  $\frac{1}{2}$ , то Артур принимает доказательство, иначе отвергает. Стандартным образом показывается, что ошибка может быть сделана экспоненциально малой, нам понадобится  $\frac{1}{12 \cdot 2^{|s|}}$ . Теперь покажем, что если Мерлин заранее узнает все  $q_{ij}$ , то это не изменит корректности алгоритма. Действительно, если  $x \in B$ , то Мерлин может послать одно и то же  $s_i$  с той же вероятностью успеха. Если же  $x \notin B$ , то нужно отдельно оценить вероятность для  $r$ . А именно, по неравенству Маркова хотя бы для трёх четвертей  $r$  условная вероятность успеха при данном  $r$  и любом  $s$  будет не больше  $\frac{1}{9} \cdot 2^{|s|}$ . Это значит, что доля  $q$ , которые подойдут хоть к какому-то  $s$ , будет не больше  $\frac{1}{9}$ . Поэтому вероятность успеха Мерлина, если он заранее узнает  $r$  и  $q$ , не превосходит  $\frac{1}{4} + \frac{3}{4} \cdot \frac{1}{9} = \frac{1}{3}$ , что и требовалось.  $\square$

Доказательство остальных случаев в теореме остаётся в качестве задачи 12.7. Той же техникой можно доказать и более общую теорему:

**Теорема 12.15** (Об ускорении, [14]). Для любого полиномиально ограниченного  $k(n)$  выполнено  $\mathbf{AM}[2k(n)] = \mathbf{AM}[k(n)]$ .

Её доказательство также остаётся в качестве задачи 12.10.

### 12.3.3 Идеальная полнота

Как обычно, за счёт многократного повторения ошибка может быть сделана экспоненциально малой. Как и в классе  $\mathbf{IP}$ , идеальная точность может быть достигнута только для языков из  $\mathbf{NP}$ , а вот полнота может быть сделана идеальной всегда. Более точно, верна следующая теорема:

**Теорема 12.16** ([51]). Обозначим через  $\mathbf{AM}^0[k]$  аналог  $\mathbf{AM}[k]$ , где  $\geq \frac{2}{3}$  заменено на  $= 1$ . Тогда  $\mathbf{AM}[k] \subset \mathbf{AM}^0[k + 1]$

В силу теоремы об ускорении добавление одного раунда почти всегда не изменяет класса. Исключением будут только однораундовые протоколы, для которых теорема превратится в утверждение  $\mathbf{BPP} \subset \mathbf{MA}^0$ .

*Доказательство.* Идея доказательства та же, что и в теореме Гача–Сипсера–Лаутемана 8.18: сначала за счёт амплификации добьёмся, что почти все случайные биты подходит, потом за счёт небольшого числа сдвигов сделаем так, чтобы подошли совсем все. Сдвиги пришлёт Мерлин, за счёт чего и возникнет дополнительный раунд.

Вначале продемонстрируем эту идею для  $\mathbf{MA}$ :  $\mathbf{MA} \subset \mathbf{MA}^0$ . Аналогично рассуждению из теоремы Гача–Сипсера получаем, что при  $x \in A$  для некоторого сертификата  $s$  и некоторых сдвигов  $q_1, \dots, q_k$  выполнено  $\exists q_1 \dots \exists q_k \forall r (V(x, s, r \oplus q_1) \vee \dots \vee V(x, s, r \oplus q_k))$ , а при  $x \notin A$  при любых сертификате и сдвигах это будет верно с вероятностью меньше  $\frac{1}{3}$ . В результате получаем следующий протокол: Мерлин присылает сертификат  $s$  и сдвиги  $q_1, \dots, q_k$ , затем Артур выбирает случайное  $r$  и проверяет  $V(x, s, r \oplus q_1) \vee \dots \vee V(x, s, r \oplus q_k)$ .

Теперь перейдём к случаю  $\mathbf{AM}$ . В этом случае мы докажем  $\mathbf{AM} \subset \mathbf{MAM}^0$ , а затем аналогично теореме 12.14  $\mathbf{MAM}^0 \subset \mathbf{AM}^0$ . Сочетание амплификации со случайными сдвигами даст такой результат: при  $x \in A$  для некоторых сдвигов выполнено  $\exists q_1 \dots \exists q_k \forall r (\exists s_1 V(x, s_1, r \oplus q_1) \vee \dots \vee \exists s_k V(x, s_k, r \oplus q_k))$ , а при  $x \notin A$  при любых сдвигах это выполнено с малой вероятностью. В результате получаем такой протокол: Мерлин присылает сдвиги  $q_1, \dots, q_k$ , затем Артур выбирает  $r$ , затем Мерлин присылает  $i$  и  $s_i$ , а Артур проверяет  $V(x, s_i, r \oplus q_i)$ .

Общий случай основан на тех же идеях. Сдвиги выбираются отдельно для всех раундов, Мерлин в самом первом сообщении присылает эти сдвиги, Артур модифицирует с их помощью случайные биты. Детали остаются в качестве упражнения 12.11.  $\square$

### 12.3.4 Связь с другими сложностными классами

Классы  $\mathbf{MA}$  и  $\mathbf{AM}$  сочетают в себе случайность и недетерминизм. Неудивительно, что они вкладываются в полиномиальную иерархию. Во-первых, выполнено такое вложение.

**Теорема 12.17.**  $\mathbf{AM} \subset \Pi_2^P$ .

*Доказательство.* Воспользуемся теоремой 12.16 об идеальной полноте. Рассмотрим верификатор из этой теоремы: если  $x \in A$ , то для любого  $r$  существует



$s$ , такое что  $V(x, s, r) = 1$ . Если же  $x \notin A$ , то для большинства  $r$  такого  $s$  не существует. Получаем  $\Pi_2$ -выражение  $\forall r \exists s V(x, s, r)$ , эквивалентное  $x \in A$ , т.е.  $A \in \Pi_2^P$ .  $\square$

Для класса **МА** выполняется аналог теоремы Гача–Сипсера–Лаутемана.

**Теорема 12.18.**  $\mathbf{MA} \subset \Pi_2^P \cap \Sigma_2^P$ .

*Доказательство.* Поскольку уже доказано  $\mathbf{MA} \subset \mathbf{AM} \subset \Pi_2^P$ , осталось доказать  $\mathbf{MA} \subset \Sigma_2^P$ . Здесь также возьмём протокол с идеальной полнотой. Тогда вложение очевидно:  $x \in A$  эквивалентно  $\exists s \forall r V(x, s, r)$ .  $\square$

Наконец, выполнен такой аналог теоремы Эйдельмана:

**Теорема 12.19.**  $\mathbf{AM} \subset \mathbf{NP}/_{\text{poly}}$ . (Под  $\mathbf{NP}/_{\text{poly}}$  понимается множество языков  $A$ , распознаваемых недетерминированными схемами из функциональных элементов полиномиального размера. У таких схем две группы входов:  $x$  и  $s$ . Если  $x \in A$ , то для некоторого  $s$  верно  $C(x, s) = 1$ , если же  $x \notin A$ , то для всех  $s$  верно  $C(x, s) = 0$ .)

*Доказательство.* Вновь воспользуемся протоколом с идеальной полнотой. Зафиксируем длину  $x$ . После достаточной амплификации можно считать, что вероятность ошибки в случае  $x \notin A$  не превосходит  $2^{-(|x|+|s|)}$ . В таком случае должно найтись  $r$ , такое что при любом  $x \notin A$  и любом  $s$  выполнено  $V(x, s, r) = 0$ . Это  $r$  можно «запаять» в схему, получится схема от  $x$  и  $s$ . При  $x \notin A$  никакое  $s$  не подойдёт, а при  $x \in A$  в силу идеальной полноты такое  $s$  найдётся.  $\square$

### 12.3.5 Неизоморфизм графов с общей случайностью

Модели интерактивных доказательств с общими и с частными случайными битами появились в одном и том же 1985 году. Уже через два года оказалось, что в случае полиномиального числа раундов они задают один и тот же класс языков. А именно, верна такая теорема.

**Теорема 12.20** ([68]).  $\mathbf{IP}[k] \in \mathbf{AM}[k + 2]$ .

Вначале покажем, как построить протокол с общими случайными битами для задачи о неизоморфизме графов. Ясно, что изученный ранее протокол не годится: пружеру как раз надо было угадать один из случайных битов.

**Теорема 12.21.**  $\mathbf{GNI} \in \mathbf{AM}$ .

*Доказательство.* Это доказательство в целом следует предложенному в работе [BNZ]. Идея состоит в том, чтобы превратить вопрос об изоморфизме графов в вопрос о количестве элементов в некотором множестве. Действительно, рассмотрим множество графов, изоморфных  $G_0$ . Если графы  $G_0$  и  $G_1$  не изоморфны, то добавление графов, изоморфных  $G_1$ , расширит это множество. Если же они изоморфны, то не расширит. То есть множество графов, изоморфных хотя бы одному из  $G_0, G_1$ , больше, если  $G_0 \not\cong G_1$ . Трудность состоит в том, что точный размер этого множества зависит от наличия автоморфизмов у графов  $G_a$ , поэтому однозначной границы провести нельзя. Выход состоит в том, чтобы добавить к графу произвольный автоморфизм этого графа.

**Утверждение 12.22.** Рассмотрим множество  $S = \{(H, \sigma) \mid H \approx G_0 \text{ или } H \approx G_1, \sigma \text{ — автоморфизм } H\}$ . Тогда если  $G_0 \approx G_1$ , то  $|S| = n!$ , а если  $G_0 \not\approx G_1$ , то  $|S| = 2n!$

*Доказательство.* Ясно, что  $S = S_0 \cup S_1$ , где  $S_a = \{(H, \sigma) \mid H \approx G_a, \sigma \text{ — автоморфизм } H\}$ . При этом если  $G_0 \approx G_1$ , то  $S_0 = S_1$ , а если  $G_0 \not\approx G_1$ , то  $S_0 \cap S_1 = \emptyset$ . Поэтому достаточно доказать, что  $|S_a| = n!$

Как известно, всего есть  $n!$  перестановок на множестве из  $n$  элементов. Перестановки образуют группу, действующую на множестве всех графов на  $n$  вершинах. Все графы, изоморфные  $G_a$ , образуют орбиту этого графа, а все автоморфизмы  $G_a$  — стабилизатор. По теореме из алгебры размер орбиты, умноженный на размер стабилизатора, равен размеру группы. А поскольку у  $H \approx G_a$  столько же автоморфизмов, сколько у  $G_a$ , то и размер  $S_a$  равен такому произведению. Значит,  $|S_a| = n!$ , что и требовалось.  $\square$

Заметим, что принадлежность к  $S$  можно удостоверить сертификатом, в котором записан изоморфизм  $H$  и  $G_a$ . Поэтому задачей Мерлина становится доказательство того, что  $|S| \geq 2n!$ . Нужно составить протокол, такой что в случае  $|S| \geq 2n!$  вероятность его успеха была существенно больше, чем в случае  $|S| \leq n!$ . Как обычно, за счёт многократного повторения можно приблизить вероятность ошибки к нулю.

Первая идея — использовать вероятностный метод: Артур выбирает несколько случайных пар (граф, перестановка) и просит Мерлина удостоверить их принадлежность к  $S$ . Если множество  $S$  большое, то вероятность успеха будет вдвое больше, чем если оно маленькое. Проблема состоит в том, что даже большое  $S$  занимает экспоненциально маленькую долю пространства ( $2n!$  из  $2^{\frac{n(n-1)}{2}} \cdot n!$ ), и потому Артур почти наверняка в него не попадёт. Поэтому нужно как-то сжать объёмлющее пространство, при этом не сильно изменив  $S$ . Это делается при помощи семейства специально выбранных хеш-функций.

**Определение 12.23.** Семейство  $\mathcal{H}_{n,k}$  функций из  $\{0,1\}^n$  в  $\{0,1\}^k$  называется семейством попарно независимых хеш-функций, если выполнены три свойства:

- **Малый размер.** Семейство  $\mathcal{H}_{n,k}$  содержит  $2^{\text{poly}(n,k)}$  функций;
- **Эффективная вычислимость.** По номеру  $h$  и входу  $x \in \{0,1\}^n$  можно за полиномиальное время вычислить  $h(x)$ ;
- **Попарная независимость.** Для любых  $x \neq x' \in \{0,1\}^n$  и любых  $y, y' \in \{0,1\}^k$  выполнено

$$\Pr_{h \in \mathcal{H}_{n,k}} [h(x) = y \wedge h(x') = y'] = \frac{1}{2^k}.$$

Идея этого определения такова: если в качестве  $h$  взять случайную функцию среди всех функций из  $\{0,1\}^n$  в  $\{0,1\}^k$ , то попарная независимость была бы заведомо выполнена. Действительно, значения  $h(x)$  и  $h(x')$  не зависят друг от друга и принимают равновероятно все возможные значения, которых  $2^k$ . Однако всех функций слишком много: двойная экспонента. Среди них нужно выбрать небольшое семейство, для которого тем не менее была бы выполнена попарная независимость.

**Теорема 12.24.** Существует семейство попарно независимых хеш-функций из  $\{0,1\}^n$  в  $\{0,1\}^n$  размера  $2^{2n}$ .

*Доказательство.* Рассмотрим поле  $\mathbb{F}_{2^n}$  из  $2^n$  элементов. Функция  $h$  будет определяться двумя элементами этого поля  $a$  и  $b$  и будет попросту линейной функцией  $h_{a,b}(x) = ax + b$ . Поскольку  $a$  и  $b$  произвольны, размер семейства составит как раз  $2^{2n}$ . Эффективная вычислимость также легко получается. Докажем попарную независимость. Пусть заданы  $x \neq x'$ ,  $y$  и  $y'$ . Докажем, что ровно для одной пары  $a$  и  $b$  выполнено одновременно  $h_{a,b}(x) = y$  и  $h_{a,b}(x') = y'$ . Действительно, система  $\begin{cases} ax + b = y \\ ax' + b = y' \end{cases}$  равносильна  $\begin{cases} ax + b = y \\ a(x' - x) = y' - y \end{cases}$ . Второе уравнение однозначно задаёт значение  $a = \frac{y' - y}{x' - x}$  (здесь используется  $x' \neq x$ ). А потому и значение  $b$  определяется однозначно как  $b = y - ax = y - \frac{y' - y}{x' - x}x$ . Поскольку нам подошла ровно одна функция из  $2^{2n}$ , вероятность будет равна  $\frac{1}{2^{2n}}$ , что и требовалось.  $\square$

Надо написать приложение про эффективный поиск неприводимого многочлена.

**Следствие 12.25.** Для всех  $k \leq n$  существует семейство попарно независимых хеш-функций из  $\{0, 1\}^n$  в  $\{0, 1\}^k$  размера  $2^{2n}$ .

*Доказательство.* Достаточно рассмотреть предыдущую конструкцию, в которой оставлены только первые  $k$  битов. Тогда

$$\begin{aligned} \Pr_{h \in \mathcal{H}_{n,k}} [h(x) = y \wedge h(x') = y'] \\ &= \sum_{z, z' \in \{0,1\}^{n-k}} \Pr_{h \in \mathcal{H}_{n,n}} [h(x) = yz \wedge h(x') = y'z'] \\ &= \sum_{z, z' \in \{0,1\}^{n-k}} \frac{1}{2^{2n}} = 2^{2(n-k)} \cdot \frac{1}{2^{2n}} = \frac{1}{2^{2k}}, \end{aligned}$$

что и требовалось для попарной независимости. Малый размер и эффективная вычислимость очевидны.  $\square$

Вернёмся к доказательству основной теоремы. Напомним, задано некоторое множество  $S \subset \{0, 1\}^m$ , которое может иметь размер либо  $K = n!$ , либо  $2K$ . Принадлежность к  $S$  удостоверяется при помощи сертификата, проверяемого за полиномиальное время.<sup>6</sup> Идея состоит в том, что некоторая хеш-функция применяется для уменьшения размера охватываемого пространства, так чтобы  $S$  уменьшилось не сильно, а потом проверяется, что случайный элемент лежит в (новом)  $S$ . Возможны два подхода к выбору хеш-функции: либо Артур выбирает её случайно, либо Мерлин специально подбирает хорошую. Оказывается, в данном случае выбор одного из двух вариантов практически не влияет на результат: почти все хеш-функции будут «хорошими». Мы проведём рассуждение для случайной функции: тогда сразу получится двухраундовый, а не трёхраундовый протокол.

Пусть  $K = n!$ . Выберем  $k$  таким образом, что  $2^{k-3} < K \leq 2^{k-2}$ . (Выбор именно такого смещения станет ясен позднее). Построим протокол так: Артур выбирает случайную хеш-функцию  $h \in \mathcal{H}_{n,k}$  и посылает её описание (в нашей конструкции это просто  $a$  и  $b$ ) Мерлину. Кроме того, Артур выбирает и отправляет случайное  $y \in \{0, 1\}^k$ . В ответ Мерлин присылает некоторое  $x \in \{0, 1\}^m$  вместе с сертификатом  $s$ , удостоверяющим его принадлежность к  $S$ . Артур, во-первых, проверяет верность сертификата и, во-вторых, проверяет равенство

<sup>6</sup>Хочется сказать, что  $S \in \mathbf{NP}$ , но в данном случае это не совсем грамотно, т.к. размер  $S$  уже фиксирован и конечен.

$h(x) = y$ . Нужно проверить, что в случае  $|S| = K$  и в случае  $|S| = 2K$  вероятности успеха Мерлина существенно отличаются, тогда за счёт многократного повторения и сравнения числа успехов со средним можно добиться нужной точности и полноты.

Пусть  $p = \frac{|S|}{2^k}$ . Таким образом,  $\frac{1}{8} < p \leq \frac{1}{2}$ : это нам пригодится позже. Докажем, что  $\Pr_{h,y}[\exists x \in S h(x) = y] \in [\frac{3p}{4}, p]$ . Этого достаточно для создания нужного разрыва, т.к. указанная вероятность в точности равна вероятности успеха Мерлина, при этом при  $|S| = K$  она не больше  $\frac{K}{2^k}$ , а при  $|S| = 2K$  — не меньше  $\frac{3}{4} \cdot \frac{2K}{2^k} = \frac{3}{2} \cdot \frac{K}{2^k}$ .

Обозначим через  $E_x$  событие  $h(x) = y$  (для случайных  $h$  и  $y$ ). В таком случае  $\Pr_{h,y}[\exists x \in S h(x) = y] = \Pr_{h,y}[\bigcup_{x \in S} E_x]$ . Вероятность объединения не больше суммы вероятностей, т.е.  $\sum_{x \in S} \Pr_{h,y}[E_x]$ . Из попарной независимости легко следует, что  $\Pr_h[h(x) = y] = \frac{1}{2^k}$  (нужно зафиксировать  $x' \neq x$  и просуммировать условия попарной независимости для всех  $y'$ ). Значит, и  $\Pr_{h,y}[E_x] = \Pr_{h,y}[h(x) = y] = \frac{1}{2^k}$  (усредняем константу по всем  $y$ ). Значит,  $\sum_{x \in S} \Pr_{h,y}[E_x] = |S| \cdot \frac{1}{2^k} = p$ , и мы получили требуемую верхнюю оценку.

Нижняя оценка доказывается по формуле включений-исключений:

$$\Pr_{h,y} \left[ \bigcup_{x \in S} E_x \right] \geq \sum_{x \in S} \Pr_{h,y}[E_x] - \sum_{x, x' \in S, x \neq x'} \Pr_{h,y}[E_x \cap E_{x'}]. \quad (12.1)$$

Как уже доказано, уменьшаемое равно  $p$ . Каждое из слагаемых вычитаемой суммы по свойству попарной независимости равно  $\frac{1}{2^{2k}}$  (это вновь верно для любого фиксированного  $y$ , а потому и для случайного). Всего слагаемых будет  $C_{|S|}^2 = \frac{|S|(|S|-1)}{2}$ . Это меньше  $\frac{|S|^2}{2}$ , поэтому правая часть (12.1) больше  $p - \frac{|S|^2}{2} \cdot \frac{1}{2^{2k}} = p - \frac{p^2}{2}$ . Здесь мы наконец воспользуемся тем, что  $p \leq \frac{1}{2}$ : при таком  $p$  выполнено  $p - \frac{p^2}{2} \geq \frac{3}{4}p$ , и нужная оценка получена.

Итак, вероятность успеха прувера либо не больше  $p^* = \frac{K}{2^k}$ , либо не меньше  $\frac{3}{2} \cdot \frac{K}{2^k} = \frac{3}{2}p^*$ . Чтобы получить окончательный протокол, нужно сделать несколько параллельных запусков вышеописанного и сравнить число успехов с  $\frac{5}{4}p^*$ . Стандартная оценка по неравенству Чернова показывает, что вероятность ошибки будет не больше  $\frac{1}{3}$ .  $\square$

### 12.3.6 Моделирование частных случайных битов при помощи общих

В этом разделе мы разберём общий случай моделирования протокола с частными случайными битами при помощи общих, т.е. докажем теорему 12.20. Доказательство проводится по индукции: в качестве базы теорема доказывается для произвольного двухраундового протокола, в качестве перехода число раундов увеличивается на 2.

**Теорема 12.26.**  $\mathbf{IP}[2] \subset \mathbf{AM}$ .

*Доказательство.* Пусть  $A \in \mathbf{IP}[2]$ . Это значит, что некоторый верификатор  $V$ , получив вход  $x$  и случайные биты  $r$ , вычисляет сообщение  $z$ . Если  $x \in A$ , то у прувера с высокой вероятностью есть ответ  $P(z)$ , такой что  $V(x, r, P(z)) = 1$  (такой ответ будем называть «хорошим»). Если же  $x \notin A$ , то такого ответа с высокой вероятностью нет. Нам будет удобно считать, что протокол имеет единичную полноту. Формально создаётся порочный круг: единичная полнота

следует как раз из текущей теоремы. Но, во-первых, протокол легко модифицировать для неидеальной полноты, а во-вторых, теорему о единичной полноте можно доказать и другим способом.

Построим протокол для  $A$  с общими случайными битами. Когда речь идёт о старом протоколе, мы будем говорить о верификаторе и прувере, а когда о новом — об Артуре и Мерлине. Идея конструкции такова: Мерлин доказывает, что множество случайных битов верификатора, для которых есть хороший ответ прувера, большое. Трудность состоит в том, что ответ прувера зависит не непосредственно от случайных битов верификатора  $r$ , а от его сообщения  $z$ . При этом одни сообщения могут появляться чаще или реже других, поэтому размер множества *сообщений*, на которые у прувера есть ответ, не связан напрямую с размером множества *случайных битов*, для которых у него есть ответ.

Предположим вначале, что любое сообщение верификатора появляется для одного и того же числа случайных битов, и потому указанной проблемы нет. Здесь возникает другая проблема: Артур может не знать, что же это за число, и потому не сможет с ним ничего сравнить. Разберём случай, когда и этой проблемы нет: Артур может сам вычислить число различных сообщений верификатора  $M(x)$ . Тогда в случае  $x \in A$  для любого сообщения верификатора есть хороший ответ прувера, а в случае  $x \notin A$  он есть только для небольшой доли сообщений. Таким образом, множество  $S$  сообщений верификатора, на которые прувер может ответить, имеет размер либо  $M(x)$ , либо меньше  $\frac{1}{2}M(x)$ . Мерлин может удостовериться принадлежность к  $S$ , прислав нужное сообщение прувера. Теперь Мерлину достаточно запустить протокол нижней оценки величины множества, аналогично рассмотренному при доказательстве теоремы 12.21.

Теперь допустим, что все сообщения верификатора появляются с одинаковой частотой, но Артур не может вычислить, сколько их. Тогда в этом вычислении ему поможет Мерлин. При этом нужно проверить, что Мерлин это число не занизил, то есть требуется протокол не нижней, а верхней оценки. Рецепт состоит в том, чтобы ограничивать не число различных сообщений, а число случайных строк, приводящих к одному сообщению. Итак, протокол будет двуступенчатым: сначала Мерлин пришлёт число сообщений верификатора  $M$  и докажет, что множество случайных строк верификатора, приводящих к некоторому конкретному сообщению, не меньше  $2^{|r|}/M$ . Затем он докажет, что множество различных сообщений, на которые есть ответ у прувера, не меньше  $M$ . При нашем предположении  $M$  обязательно является степенью двойки, поэтому если Мерлин занизит  $M$ , то хотя бы вдвое, и Артур скорее всего найдёт ошибку. Если же  $M$  правильное, а  $x \notin A$ , то Артур тоже скорее всего найдёт ошибку, как и раньше.

Наконец, избавимся от предположения о равномерности. В таком случае Мерлин передаст информацию о распределении разных сообщений по частоте. Конечно, перечислить все возможные частоты вместе с количеством слов такой частоты Мерлин не сможет: это слишком большой объём информации. Вместо этого он кластеризует сообщения по частоте от  $2^{i-1}$  до  $2^i - 1$  и пришлёт размер  $c_i$  каждого кластера. Артур проверяет, что  $\sum c_i 2^i > 2^{|r|}$ : кластеры заявленных размеров покрывают всё пространство случайных строк. Далее он проверяет, что каждый кластер действительно имеет нужный размер. Это делается так: для каждого кластера Мерлин доказывает при помощи протокола нижней оценки, что в нём хотя бы  $c_i$  элементов. Отличие состоит в том, что Мерлин уже не может предоставить один сертификат принадлежности к кластеру. Вместо этого он запускает тот же протокол и доказывает, что случайных строк, по-

рождающих данное сообщение, хотя бы  $2^{i-1}$ . Чтобы число раундов осталось константным, нужно запускать протокол для всех кластеров параллельно. Потом нужно его снизить до 2 стандартным образом.

Чтобы доказать корректность протокола, нужно сначала снизить ошибку до  $\frac{1}{16}$ . Идея состоит в следующем. Во-первых, Мерлин может занизить в 2 раза размеры кластеров. Во-вторых, порог на число случайных строк, выполнение которого доказывает Мерлин, может быть вдвое меньше настоящего. В-третьих, и этот порог разрешается недобрать наполовину. Наконец, можно фальшивым образом использовать кластеры меньшего размера. Но если ошибка снижена до  $\frac{1}{16}$ , то Мерлин в любом случае проиграет с большой вероятностью.

Более подробно, пусть  $c_i$  — заявленные Мерлином размеры кластеров, а  $c_i^*$  — настоящие.  $\square$

### 12.3.7 Может ли задача GI быть NP-полной?

Мы уже не раз упоминали, что для задачи об изоморфизме графов не известно ни полиномиального алгоритма, ни доказательства NP-полноты. Весомым аргументом в пользу не NP-полноты этой задачи стал квазиполиномиальный алгоритм Бабаи [13] для этой задачи. Другой аргумент был известен гораздо раньше [23]: если задача NP-полна, то полиномиальная иерархия схлопывается.

**Теорема 12.27 ([23]).** *Если задача GI является NP-полной, то  $\Sigma_2^P = \Pi_2^P$ .*

*Доказательство.* Как обычно, достаточно доказать  $\Sigma_2^P \subset \Pi_2^P$ , затем за счёт перехода к дополнениям  $\Pi_2^P \subset \Sigma_2^P$  и потому эти классы равны. Если задача GI является NP-полной, то GNI является coNP-полной, а потому  $\text{TAUT} \leq_p \text{GNI}$ . Иными словами, есть некоторая полиномиально вычислимая функция  $g$ , переводящая формулы в пары графов, такая что  $\varphi$  является тавтологией тогда и только тогда, когда  $g(\varphi)$  есть пара неизоморфных графов. Покажем, что в таком случае  $\Sigma_2\text{SAT} \in \Pi_2$ . Пусть  $\psi \in \Sigma_2\text{SAT}$ . Иными словами,  $\exists x \forall y \psi(x, y)$ , т.е.  $\exists x \psi_x \in \text{TAUT}$  (Здесь  $\psi_x$  — формула  $\psi$ , у которой зафиксировано значение первой группы переменных). Значит,  $\exists x g(\psi_x) \in \text{GNI}$ . При помощи теоремы 12.16 модифицируем АМ-протокол для GNI, так чтобы полнота была идеальной, а затем при помощи многократного повторения сделаем ошибку меньше  $2^{-|z|}$ , где  $z$  — общий вход протокола. Значит, по определению, есть некоторый полиномиальный алгоритм  $V$ , такой что при  $z \in \text{GNI}$  для любого  $r$  существует  $w$ , такой что  $V(z, r, w) = 1$ , а при  $z \notin \text{GNI}$  только для доли меньше  $2^{-|z|}$  всех  $r$  существует  $w$ , такой что  $V(z, r, w) = 1$ . Последнее значит, что для некоторого универсального  $r$  при всех  $z \notin \text{GNI}$  и всех  $w$  выполнено  $V(z, r, w) = 0$ .

Теперь покажем, что  $\psi \in \Sigma_2\text{SAT}$  эквивалентно  $\forall r \exists x \exists w V(g(\psi_x), r, w) = 1$ . Действительно, если  $\psi \in \Sigma_2\text{SAT}$ , то  $\exists x g(\psi_x) \in \text{GNI}$ , откуда  $\exists x \forall r \exists w V(g(\psi_x), r, w) = 1$ . Стандартной перестановкой кванторов выводим, что  $\forall r \exists x \exists w V(g(\psi_x), r, w) = 1$ , что и требовалось. Теперь предположим, что  $\psi \notin \Sigma_2\text{SAT}$ . Тогда  $\nexists x g(\psi_x) \in \text{GNI}$ , т.е.  $\forall x g(\psi_x) \notin \text{GNI}$ . Но поскольку  $\exists r \forall z \notin \text{GNI} \forall w V(z, r, w) = 1$ , то  $\exists r \forall x \forall w V(g(\psi_x), r, w) = 0$ , а это в точности отрицание  $\forall r \exists x \exists w V(g(\psi_x), r, w) = 1$ , что и требовалось. Поскольку  $\psi \in \Sigma_2\text{SAT}$  эквивалентно  $\forall r \exists x \exists w V(g(\psi_x), r, w) = 1$ , имеем  $\Sigma_2\text{SAT} \in \Pi_2^P$ , а поскольку  $\Sigma_2\text{SAT}$  полна в  $\Sigma_2^P$ , то  $\Sigma_2^P \subset \Pi_2^P$ , откуда  $\Sigma_2^P = \Pi_2^P$ , что и требовалось.  $\square$

Заметим, что вместо GNI можно было использовать любой другой язык из АМ. Отсюда можно вывести более общую теорему:



**Теорема 12.28** ([23]). Если некоторая  $\mathbf{coNP}$ -полная лежит в  $\mathbf{AM}$ , то  $\Sigma_2^P = \Pi_2^P$ .

*Доказательство.* Доказательство дословно повторяет предыдущее с заменой  $\mathbf{GN}$  на данную  $\mathbf{coNP}$ -полную задачу.  $\square$

## 12.4 $\mathbf{IP} = \mathbf{PSPACE}$

Как мы уже видели, сама по себе интерактивность не выводит за пределы класса  $\mathbf{NP}$ . Сама по себе случайность тоже не даёт многого: большинство исследователей верят, что  $\mathbf{P} = \mathbf{BPP}$ , и уж точно  $\mathbf{BPP}$  лежит на втором уровне полиномиальной иерархии. Более того, мы видели, что  $\mathbf{IP}[const] = \mathbf{AM} \subset \Pi_2^P$ , так что случайность и интерактивность вместе тоже дают мало, если число раундов ограничено константой. Долгое время интерактивных протоколов не удавалось построить даже для  $\mathbf{coNP}$ -полных задач, и потому многие предполагали, что и полиномиальное число раундов не сможет ничего добавить, так что  $\mathbf{IP}$  окажется тоже где-то невысоко в полиномиальной иерархии. Это предположение оказалось в корне неверно: на самом деле  $\mathbf{IP}$  равняется  $\mathbf{PSPACE}$ . Это утверждение называют  $\mathbf{IP}$ -теоремой или теоремой Шамира. Ади Шамир действительно сделал последний шаг [122], но большой вклад сделали и другие исследователи. Забавно, что незадолго до того появившаяся интерактивность в виде рассылок по электронной почте сыграла решающую роль в получении доказательства. Подробности приведены в исторических замечаниях и обзоре [12]. Итак, в этом разделе мы доказываем следующую теорему:

**Теорема 12.29** ([122]).  $\mathbf{IP} = \mathbf{PSPACE}$ .

Мы уже выяснили (теорема 12.5), что  $\mathbf{IP} \subset \mathbf{PSPACE}$ , т.е. осталось доказать  $\mathbf{PSPACE} \subset \mathbf{IP}$ . Для этого достаточно построить систему интерактивных доказательств для какой-нибудь  $\mathbf{PSPACE}$ -полной задачи, например,  $\mathbf{TQBF}$ . Но начнём мы с общей идеи и более простых задач.

### 12.4.1 Идея арифметизации

Идея доказательства заключается в том, чтобы перейти от логики к алгебре. А именно, логическая формула заменяется на многочлен в некотором поле  $\mathbb{F}$ , принимающий на элементах 0 и 1 из поля те же значения, что и на формула. Если  $\mathbb{F}$  есть поле из двух элементов, то получаются известные многочлены Жегалкина. В общем случае нужно учитывать, что сложение и вычитание это не одно и то же:  $\neg x$  заменяется на  $1 - x$ ,  $x \vee y$  на  $x + y - xy$ , и только  $x \wedge y$  остаётся  $xy$ . В результате дизъюнкт трёх литералов превращается в мультилинейный<sup>7</sup> многочлен третьей степени, например:

$$x \vee \neg y \vee z \mapsto (x + (1 - y) - x(1 - y)) + z - z(x + (1 - y) - x(1 - y)) = 1 - y + xy - yz - xyz.$$

А вся формула, приведённая к виду 3-КНФ, превращается в многочлен степени  $3m$ . Впрочем, в этом случае лучше все скобки не раскрывать, а оставить произведение  $m$  многочленов третьей степени. Тогда длина записи останется полиномиальной.

<sup>7</sup>Для мультилинейности требуется, чтобы все переменные в дизъюнкте были разными, но этого легко добиться.

После того, как формула превращена в многочлен, нужно переформулировать в новых терминах её свойства. Например, тавтологичность формулы означает, что многочлен всегда будет равен единице, если в качестве значений переменных подставить нули и единицы в произвольной комбинации. Пока что польза от арифметизации не видна: посчитать  $2^n$  значений многочлена так же сложно, как проверить формулу для  $2^n$  наборов. Однако многочлен имеет малую степень, что накладывает определённые ограничения. Общая идея дальнейшего протокола состоит в том, что верификатор запрашивает у прувера определённые характеристики многочлена, в процессе проверяет соотношения, а в итоге получает нужную информацию. В следующих двух подразделах мы конкретизируем эту идею.

### 12.4.2 Интерактивные доказательства для тавтологий

Как известно, задача TAUT о тавтологичности булевой формулы является **coNP**-полной. Изначально интерактивные протоколы были построены только для задач, чья **coNP**-полнота неизвестна, таких как GNI. Более того, как мы видели в теореме 12.28, доказательства с константным числом раундов для TAUT скорее всего нет. Идея арифметизации и линейное число раундов позволили перескочить этот порог и довольно быстро дойти до всей полиномиальной иерархии. Итак, мы доказываем такую теорему:

**Теорема 12.30.**  $\text{TAUT} \in \text{IP}$ .

*Доказательство.* Пусть задана формула  $\varphi$  в виде 3-КНФ, по которой построен многочлен  $P_\varphi$  над некоторым полем  $\mathbb{F}_p$ . Тогда  $\varphi$  является тавтологией тогда и только тогда, когда  $P_\varphi(b_1, \dots, b_n) = 1$  для всех  $b_i \in \{0, 1\}$ . Поскольку по построению значения многочлена равны либо нулю, либо единице, при  $p > 2^n$  это эквивалентно условию на сумму:

$$\sum_{(b_1, \dots, b_n) \in \{0, 1\}^n} P_\varphi(b_1, b_2, \dots, b_n) = 2^n. \quad (12.2)$$

Нам будет удобно представить это выражение как результат последовательного суммирования:

$$\sum_{b_1 \in \{0, 1\}} \sum_{b_2 \in \{0, 1\}} \cdots \sum_{b_n \in \{0, 1\}} P_\varphi(b_1, b_2, \dots, b_n) = 2^n. \quad (12.3)$$

Теперь раскроем первую сумму:

$$\sum_{b_2 \in \{0, 1\}} \cdots \sum_{b_n \in \{0, 1\}} P_\varphi(0, b_2, \dots, b_n) + \sum_{b_2 \in \{0, 1\}} \cdots \sum_{b_n \in \{0, 1\}} P_\varphi(1, b_2, \dots, b_n) = 2^n.$$

На получившееся равенство можно посмотреть так:  $Q_1(0) + Q_1(1) = 2^n$ , где

$$Q_1(x) = \sum_{b_2 \in \{0, 1\}} \cdots \sum_{b_n \in \{0, 1\}} P_\varphi(x, b_2, \dots, b_n).$$

Поскольку  $P_\varphi$  является многочленом маленькой степени, то и  $Q_1$  является многочленом маленькой степени — не больше  $t$  за счёт мультилинейности всех скобок. Коэффициенты такого многочлена уже можно передать в полиномиальном сообщении, что и сделает прувер. Верификатор, получив многочлен  $\tilde{Q}_1$ , первым делом проверяет  $\tilde{Q}_1(0) + \tilde{Q}_1(1) = 2^n$  и отвергает всё доказательство, если



проверка не прошла. Но он не может проверить самостоятельно, что  $\tilde{Q}_1 = Q_1$ . Здесь мы воспользуемся свойствами поля и малой степенью многочлена: если  $\tilde{Q}_1 \neq Q_1$ , то число таких  $a$ , что  $\tilde{Q}_1(a) = Q_1(a)$ , не превосходит  $m$ . Верификатор выбирает случайное  $a_1 \in \mathbb{F}_p$  и предлагает пруверу доказать  $Q_1(a_1) = \tilde{Q}_1(a_1)$ , т.е.

$$\sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\varphi(a_1, b_2, \dots, b_n) = \tilde{Q}_1(a_1).$$

Это выражение имеет тот же вид, что и (12.3), только цепочка сумм на 1 короче, а в правой части  $\tilde{Q}_1(a_1)$  вместо  $2^n$ . Доказательство также будет идти аналогично: верификатор ожидает от прувера коэффициенты многочлена

$$Q_2(x) = \sum_{b_3 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\varphi(a_1, x, b_3, \dots, b_n),$$

прувер высылает некоторый  $\tilde{Q}_2$ , верификатор проверяет  $\tilde{Q}_2(0) + \tilde{Q}_2(1) = Q_1(a_1)$ , затем выбирает случайное  $a_2 \in \mathbb{F}_p$  и просит прувера доказать

$$\sum_{b_3 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\varphi(a_1, a_2, \dots, b_n) = \tilde{Q}_2(a_2).$$

Так продолжается по индукции, пока не будут определены все  $a_1, a_2, \dots, a_n$ . Верификатору останется проверить, что  $P_\varphi(a_1, a_2, \dots, a_n) = \tilde{Q}_n(a_n)$ . Это он может сделать уже самостоятельно, подсчитав значения всех скобок в многочлене  $P_\varphi$  и перемножив их.

Протокол полностью описан, осталось доказать его корректность. Если изначальное условие выполнено, то пруверу достаточно на каждом этапе присылать  $\tilde{Q}_i = Q_i$ , и все проверки пройдут с вероятностью 1. Если же изначальное условие не выполнено, то в первом раунде у прувера есть два варианта. Если он присылает  $\tilde{Q}_1 = Q_1$ , то проверка  $\tilde{Q}_1(0) + \tilde{Q}_1(1) = 2^n$  не проходит, и доказательство отвергается. Если же он присылает  $\tilde{Q}_1 \neq Q_1$ , для которого проверка пройдёт, то с вероятностью не меньше  $1 - \frac{m}{p}$  верификатор выберет такое  $a_1$ , что  $\tilde{Q}_1(a_1) \neq Q_1(a_1)$ , и прувер останется с неправильным утверждением на втором раунде. Продолжая по индукции, получим, что с вероятностью не меньше  $\left(1 - \frac{m}{p}\right)^n$  утверждение останется неверным и после  $n$  раундов, а тогда его ложность верификатор обнаружит при последней проверке. По неравенству Бернулли  $\left(1 - \frac{m}{p}\right)^n > 1 - \frac{mn}{p}$ , так что при  $p > 3mn$  вероятность того, что неверное доказательство будет принято, будет меньше трети, что и требуется в определении  $\mathbf{IP}$ .

Осталось осветить вопрос, откуда взять  $p$ . Нам требуется 3 вещи:  $p$  должно быть простым, чтобы  $\mathbb{F}_p$  было полем,  $p$  должно быть больше  $2^n$ , чтобы тавтологичность  $\varphi$  была эквивалентна условию (12.2), и  $p$  должно быть больше  $3mn$ , чтобы получить нужную ошибку. На самом деле нужна и четвёртая вещь:  $p$  должно быть достаточно маленьким, чтобы все операции в поле можно было провести за полиномиальное время. Третье условие фактически следует из второго: если  $m$  настолько большое, что  $3mn > 2^n$ , то можно просто вычислить всю таблицу истинности за полиномиальное время. Четвёртое условие тоже необременительное: как известно, между  $x$  и  $2x$  всегда найдётся простое число, так что можно считать, что  $p < 2^{n+1}$ . Остаётся вопрос, как его найти. Можно пойти двумя путями: либо число пришлёт прувер, а верификатор проверит простоту, либо верификатор будет пытаться найти его сам, тогда останется небольшая вероятность, что поиск не увенчается успехом. В любом случае число будет получено за полиномиальное время.  $\square$

### 12.4.3 Интерактивные доказательства для булевых формул с кванторами

Техники предыдущего раздела хватает, чтобы построить интерактивные протоколы для всей полиномиальной иерархии (см. упр. 12.16 и 12.17), т.е. для задач о булевых формулах с кванторами, где кванторы меняются константное число раз. Чтобы получить результат  $\mathbf{IP} = \mathbf{PSPACE}$ , нужно распространить подход на формулы с полиномиальным числом перемен кванторов. Для этого нам потребуется идея линеаризации: мы пока нигде не использовали, что на интересующих нас значениях выполнено  $x^k = x$  при  $k > 0$ .

Итак, рассмотрим формулу с кванторами  $\exists x_1 \forall x_2 \dots \forall x_k \varphi(x_1, x_2, \dots, x_k)$ . Для простоты будем считать, что каждый квантор действует ровно на одну переменную (добавим фиктивных переменных, если это не так). Как и прежде, превратим формулу  $\varphi$  в многочлен  $P_\varphi$ . Раньше мы использовали такую эквивалентность:  $\forall x \in \{0, 1\} \varphi(x)$  равносильно  $P_\varphi(0) + P_\varphi(1) = 2$ . Теперь это неудобно: непонятно, как коротко записать, что одно из двух значений равно двум, а с увеличением числа кванторов сложность будет нарастать. Вместо этого мы используем соглашение, типичное для многих языков программирования: ложь обозначается нулём, а истина — любым целым положительным числом. В этом случае  $\forall x \in \{0, 1\} \varphi(x)$  равносильно  $P_\varphi(0) \cdot P_\varphi(1) > 0$ , а  $\exists x \in \{0, 1\} \varphi(x)$  равносильно  $P_\varphi(0) + P_\varphi(1) > 0$ . Сделав так для всех кванторов, получаем, что  $\exists x_1 \forall x_2 \dots \forall x_k \varphi(x_1, x_2, \dots, x_k)$  равносильно

$$\sum_{b_1 \in \{0,1\}} \prod_{b_2 \in \{0,1\}} \sum_{b_3 \in \{0,1\}} \dots \prod_{b_n \in \{0,1\}} P_\varphi(b_1, b_2, b_3, \dots, b_n) > 0. \quad (12.4)$$

Первая идея состоит в том, чтобы повторить предыдущие рассуждения: сначала пруввер присылает точное значение  $K$  выражения в левой части, затем присылает многочлен  $Q_1$ , получающийся после снятия первого суммирования. Верификатор проверяет, что  $Q_1(0) + Q_1(1) = K$ , выбирает случайное  $a_1$  и ожидает доказательства, что  $Q_1(a_1)$  равняется оставшемуся выражению, в которое подставили  $a_1$ . Прувер присылает многочлен  $Q_2$ , верификатор проверяет  $Q_2(0) \cdot Q_2(1) = Q_1(a_1)$ , выбирает случайное  $a_2$ , и так далее. Такая схема действительно работает, вот только время работы может быть сверхполиномиальным. Даже на самом первом этапе, как только  $b_1$  заменена на переменную  $x$ , каждый значок  $\prod$  в выражении потенциально удваивает степень многочлена, а всего таких операций  $\frac{n}{2}$ . Так что и передача коэффициентов  $Q_1$ , и вычисление его значений могут занять экспоненциальное время. Это же верно и для последующих  $Q_i$ . Есть и другая связанная проблема: каждая операция  $\prod$  может возвести число в квадрат, отчего само значение левой части (12.4) может стать дважды экспоненциальным. Тогда уже нельзя взять размер поля  $p$ , который будет заведомо больше этого значения.

Для удерживания степени многочлена в полиномиальных рамках нам понадобится идея линеаризации. На самом деле значения многочлена нас интересуют только на 0 и 1, для которых выполнено соотношение  $x^2 = x$ . Так что если где-то возникнет нелинейность, можно её оперативно убрать.

**Определение 12.31.** Обозначим через  $x_{-i}$  набор  $(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$ .<sup>8</sup> Оператором линеаризации по переменной  $x_i$  называется функция  $\mathcal{L}_i$ , которая переводит многочлен  $Q(x_i, x_{-i})$  в многочлен  $x_i \cdot Q(1, x_{-i}) + (1 - x_i) \cdot Q(0, x_{-i})$ . (Все операции проводятся в некотором поле  $\mathbb{F}_p$ ).

<sup>8</sup>Такое обозначение стандартно в теории игр.

Ясно, что для любых  $Q$  и  $i$  многочлен  $\mathcal{L}_i Q$  линеен по  $x_i$  и совпадает с  $Q$  при  $x_i \in \{0, 1\}$ .

Для единообразия введём ещё два оператора:

**Определение 12.32.** *Оператором суммирования по переменной  $x_i$  называется функция  $\mathcal{S}_i$ , которая переводит многочлен  $Q$  в многочлен  $Q(0, x_{-i}) + Q(1, x_{-i})$ . Оператором перемножения по переменной  $x_i$  называется функция  $\mathcal{P}_i$ , которая переводит многочлен  $Q$  в многочлен  $Q(0, x_{-i}) \cdot Q(1, x_{-i})$ .*

В таком случае неравенство (12.4) можно представить в таком виде:

$$\mathcal{S}_1 \mathcal{P}_2 \mathcal{S}_3 \dots \mathcal{P}_n P_\varphi > 0. \quad (12.5)$$

Теперь «разбавим» левую часть операторами линеаризации. Заметим, что операторы суммирования сохраняют линейность, а вот операторы перемножения её могут нарушить. Поэтому после применения каждого оператора перемножения её будем её восстанавливать (только для тех переменных, которые ещё остались):

$$\mathcal{S}_1 \mathcal{L}_1 \mathcal{P}_2 \mathcal{S}_3 \mathcal{L}_1 \mathcal{L}_2 \mathcal{L}_3 \mathcal{P}_4 \dots \mathcal{S}_{n-1} \mathcal{L}_1 \mathcal{L}_2 \dots \mathcal{L}_{n-1} \mathcal{P}_n P_\varphi > 0. \quad (12.6)$$

На дальнейшее индуктивное рассуждение можно смотреть двояко. С одной стороны, верификатор умеет сам вычислять значение  $P_\varphi$  в любой точке. При помощи прувера он научится вычислять значения  $\mathcal{P}_n P_\varphi$ ,  $\mathcal{L}_{n-1} \mathcal{P}_n P_\varphi$ , и т.д., пока не дойдёт до всей левой части (12.6), которую сравнит с заявленным прувером значением. С другой стороны, сам протокол будет устроен иначе: изначально прувер заявляет истинность соотношения (12.6), дальше верификатор будет просить его удостоверить подобные соотношения для всё меньших и меньших цепочек операторов, пока всё не сведётся к вычислению  $P_\varphi$ , что верификатор сможет сделать сам.

Навешивание операторов  $\mathcal{S}_i$  и  $\mathcal{P}_i$  производится так же, как и в теореме 12.30. А именно, пусть верификатор уже научился проверять любое условие вида  $Q(a_i, a_{-i}) = K$  для любых  $a_i$  и  $a_{-i}$ . Тогда он может проверить условие  $\mathcal{S}_i Q(x_i, a_{-i}) = M$  для фиксированного  $a_{-i}$  следующим образом: прувер присылает коэффициенты многочлена  $\tilde{Q}(x_i)$ , который должен совпадать с  $Q(x_i, a_{-i})$ . Верификатор проверяет  $\tilde{Q}(0) + \tilde{Q}(1) = M$ , выбирает случайное  $a_i$  и по индукции проверяет  $Q(a_i, a_{-i}) = \tilde{Q}(a_i)$ . Для условия  $\mathcal{P}_i Q(x_i, a_{-i}) = L$  всё точно так же, но с заменой  $\tilde{Q}(0) + \tilde{Q}(1) = M$  на  $\tilde{Q}(0) \cdot \tilde{Q}(1) = M$ . Для условия  $\mathcal{L}_i Q(x_i, a_{-i}) = M$  процедура также мало отличается: верификатор проверяет условие  $(1 - x_i)Q(0) + x_i Q(1) = M$ , выбирает случайное  $a_i$  и по индукции проверяет  $Q(a_i, a_{-i}) = \tilde{Q}(a_i)$ . Отличие состоит лишь в том, что не уменьшается число переменных: значение  $x_i$  также участвует в проверяемом условии.

Теперь посмотрим на это же рассуждение с другой стороны. В каждый промежуточный момент имеется утверждение такого вида: цепочка операторов суммирования, перемножения и линеаризации, применённая к  $P_\varphi$ , после фиксации значений свободных переменных  $a_i$  возвращает значение  $M$ . Нужно перейти к новому утверждению, в котором цепочка будет короче. Если внешний оператор является оператором суммирования или перемножения, то рассуждение точно такое же, как в теореме 12.30: прувер присылает многочлен  $\tilde{Q}$  от переменной  $x_i$ , который должен получиться, если снять внешний оператор, а значения всех зафиксированных переменных оставить прежними. Верификатор проверяет, что  $Q(0) + Q(1) = M$  (или  $Q(0) \cdot Q(1) = M$ ), выбирает случайное  $a_i$ , фиксирует  $x_i = a_i$  и просит доказать, что новое выражение равняется  $\tilde{Q}(a_i)$ . Если же внешний оператор является оператором линеаризации, то прувер

присылает многочлен  $\tilde{Q}$  от переменной  $x_i$ , который должен получиться, если убрать оператор линеаризации и *фиксацию значения*  $x_i$ . Верификатор проверяет  $(1 - x_i)Q(0) + x_iQ(1) = M$  для старого значения  $x_i$ , выбирает новое случайное  $a_i$ , фиксирует  $x_i = a_i$  и просит доказать, что новое выражение равно  $\tilde{Q}(a_i)$ .

Корректность протокола обосновывается так же, как и в теореме 12.30. Разве что, потребуется более строгое условие на ошибку в каждом раунде. Во-первых, степень многочлена может вырастать до  $m^2$ , во-вторых, самих раундов будет примерно  $n^2/4$ . Однако экспоненциального  $p$  хватит, чтобы ошибка не накопилась.

Более важен вопрос о выборе  $p$ . Как уже говорилось, значение левой части  $K$  в (12.4) может быть дважды экспоненциальным, поэтому выбрать  $p$  больше этого значения не получится. Вместо этого нужно выбрать  $p$  так, чтобы нужное значение было не нулём по модулю  $p$ . Подходящее  $p$  пришлёт прuver, но нужно доказать, что такое есть. Это делается так:  $K$  будет не выше  $2^{2^{n/2}}$ . При этом по теореме о распределении простых чисел между  $2^n$  и  $2^{2n}$  при больших  $n$  будет больше  $\frac{2^{2n}}{3n}$  простых чисел. Произведение всех этих чисел будет больше

$$(2^n)^{\frac{2^{2n}}{3n}} = 2^{n \cdot \frac{2^{2n}}{3n}} > 2^{2^{n/2}} \geq K,$$

откуда следует, что  $K$  хотя бы на одно из них не делится. Именно его прuver и пришлёт в качестве  $p$ .

## 12.5 Исторические замечания и рекомендации по литературе

Идея чередования недетерминизма и случайности впервые появилась в статье Христоса Пападимитриу [102]. В этой статье класс **PSPACE** был охарактеризован как класс игр, в которых есть выигрышная стратегия против «природы», делающей случайные ходы.

Понятие интерактивного доказательства появилось в двух статьях 1985 года: с частными случайными битами в работе Шафи Голдвассер, Сильвио Микали и Чарльза Ракоффа [67] и с общими случайными битами в работе Ласло Бабаи [11], который также ввёл терминологию «игры Артура–Мерлина».

В последующие 5 лет область бурно развивалась: получен результат Голдвассер–Сипсера [68] о моделировании частных случайных битов при помощи общих, в работе Бошпаны–Хостада–Закоса [23] показано, что полиномиальная иерархия схлопывается при наличии коротких интерактивных доказательств для языков из **coNP**, доказана теорема об ускорении Бабаи–Морана [14]. Встал вопрос, даёт ли полиномиальное число раундов больше, чем константное? Было получено два разных (хотя формально и совместных друг с другом) ответа для релятивизованных вычислений. С одной стороны, интерактивность может много: в работе [9] Айелло, Голдвассер и Хостада построили оракул  $B$ , для которого  $\mathbf{IP}^B \not\subseteq \mathbf{PH}^B$ . С другой стороны, она может мало: в работе [50] Фортноу и Сипсер построили оракул  $C$ , для которого  $\mathbf{coNP}^C \not\subseteq \mathbf{IP}^C$ .

Доказательства с идеальной точностью или идеальной полнотой были рассмотрены в работе [51].

События резко ускорились в конце 1989 года, когда в результате интенсивного обмена сообщениями по электронной почте было доказано, что  $\mathbf{IP} = \mathbf{PSPACE}$ . Началось всё с письма Ноама Нисана 27 ноября 1989 года, сообщившего о придуманном им интерактивном протоколе с несколькими прuverами

для задачи о вычислении перманента. Важно, что использованный им метод не релятивизировался, что позволило обмануть оракул Фортноу–Сипсера. Сам Нисан уехал в путешествие по Южной Америке, но вскоре в Чикаго Фортноу совместно с Карстеном Лундом и Говардом Карлоффом придумали, как сократить число пруверов до одного, о чём и объявили 13 декабря. В итоге появилась статья 4 авторов [94]. Протокол для перманента означал, что  $\mathbf{PH} \subseteq \mathbf{IP}$ , стало понятно, что никаких концептуальных препятствий к равенству  $\mathbf{IP}$  и  $\mathbf{PSPACE}$  нет. Началась гонка за получением доказательства, в которой победил Ади Шамир. Соответствующее письмо он отправил 26 декабря, а впоследствии опубликовал статью [122]. Более подробно о переписке можно прочесть в обзоре Бабаи [12]. Доказательство с оператором линеаризации принадлежит Александру Шеню [124], в исходном доказательстве формула с кванторами хитрым образом модифицировалась, чтобы степень многочлена была не очень большой.

Класс  $\mathbf{IPR}$  (см. упр. 12.4) определён в работе [29]. Там же доказано, что  $\mathbf{IPR}^A = \mathbf{PSPACE}^A$  для любого оракула  $A$ , в то же время  $\mathbf{IP}^A \neq \mathbf{PSPACE}^A$  для случайного оракула  $A$ . Таким образом, была опровергнута «гипотеза о случайном оракуле»: соотношение между классами, выполненное при случайном оракуле, выполнено и без оракула.

В работе [45] показано, что для некоторого оракула  $\mathbf{NP}^{\mathbf{BPP}^A} \subsetneq \mathbf{MA}^A$ .

Протокол для  $\mathbf{GNI}$  впервые появился в статье Голдвассер, Микали и Вигдерсона [66].

## 12.6 Задачи и упражнения

**12.1. Квадратичный невычет.** Постройте систему интерактивных доказательств для языка  $\mathbf{QNR} = \{(x, m) \mid x \text{ является квадратичным невычетом по модулю } m\}$ .

**12.2. Оптимальный прувер.** Пусть фиксирован верификатор  $V$ . Докажите, что существует прувер, максимизирующий  $\Pr[V^P(x) = 1]$  для каждого  $x$  и вычислимый на полиномиальной памяти. Сделайте вывод, что  $\mathbf{IP} \subseteq \mathbf{PSPACE}$ .

**12.3. Прувер с рандомизацией.** Докажите, что класс  $\mathbf{IP}$  не изменится, если разрешить пруверу пользоваться случайными битами.

**12.4. Класс  $\mathbf{IPR}$ .** Назовём классом  $\mathbf{IPR}$  аналог класса  $\mathbf{IP}$ , где в определении  $\geq \frac{2}{3}$  и  $\leq \frac{1}{3}$  заменены на  $> \frac{1}{2}$  и  $< \frac{1}{2}$  соответственно. Докажите, что  $\mathbf{IPR} = \mathbf{IP}$ .

**12.5. Два определения  $\mathbf{AM}$ .** Докажите, что определение 12.10 эквивалентно аналогу определения 12.4, в котором верификатор вместо частных случайных битов использует общие.

**12.6.** Докажите, что  $\mathbf{NP}^{\mathbf{BPP}} \subseteq \mathbf{MA}$ .

**12.7.** Завершите доказательство теоремы 12.14.

**12.8. Двуступенчатая вероятность.** Определим классы  $\mathbf{AMA}'$  и  $\mathbf{AMA}''$  так:  $B \in \mathbf{AMA}'$  ( $\mathbf{AMA}''$ ), если существует полиномиальный алгоритм  $V(x, r, s, q)$ , такой что:

- Если  $x \in B$ , то  $\Pr_r [\exists s \Pr_q [V(x, r, s, q) = 1] \geq \frac{2}{3}] \geq \frac{2}{3}$

- (для **АМА'**) Если  $x \notin B$ , то  $\Pr_r [\exists s \Pr_q [V(x, r, s, q) = 1] \geq \frac{2}{3}] \leq \frac{1}{3}$
- (для **АМА''**) Если  $x \notin B$ , то  $\Pr_r [\forall s \Pr_q [V(x, r, s, q) = 1] \leq \frac{1}{3}] \geq \frac{2}{3}$
- а) Поясните, в чём отличие трёх определений (этих двух и стандартного **АМА**). А именно, почему один и тот же  $V$  может удовлетворять одному и не удовлетворять другому.
- б) Докажите, что  $\mathbf{PP} \subset \mathbf{АМА}'$ .
- в) Докажите, что  $\mathbf{АМА}'' = \mathbf{АМА}$ .

**12.9. Коллапсирование полиномиальной иерархии (/23/).** Пусть  $\mathbf{coNP} \subset \mathbf{AM}$ . Докажите, что тогда  $\Sigma_2^P = \Pi_2^P = \mathbf{AM}$ .

**12.10.** Докажите теорему 12.15.

**12.11.** Завершите доказательство теоремы 12.16.

**12.12. Протокол для малой разницы в размере.** Пусть задано множество  $S$ , принадлежность к которому можно быстро удостоверить, про которое известно, что либо  $|S| \geq K$ , либо  $|S| \leq 0.99K$ . Постройте **АМ**-протокол, в котором два случая отделяются друг от друга.

**12.13. Идеальная полнота для неизоморфизма графов.** Постройте явным образом **АМ**-протокол для задачи **GNI**, обладающий единичной полнотой.

**12.14. Артур начинает и выигрывает.** Докажите, что если  $\mathbf{NP} \subset \mathbf{P}/\text{poly}$ , то  $\mathbf{AM} = \mathbf{MA}$ .

**12.15. Мерлин всемогущий.** Докажите, что если  $\mathbf{PSPACE} \subset \mathbf{P}/\text{poly}$ , то  $\mathbf{IP} = \mathbf{MA}$ . (Значит, и  $\mathbf{PSPACE} = \mathbf{MA}$ ).

**12.16. Подсчёт числа выполняющих наборов.** Постройте систему интерактивных доказательств для языка  $\#\text{SAT}_D = \{(\varphi, k) \mid \text{у формулы } \varphi \text{ ровно } k \text{ выполняющих наборов}\}$ .

**12.17.  $\mathbf{PH} \subset \mathbf{IP}$ .** Не опираясь на конструкцию для булевых формул с кванторами, докажите, что полиномиальная иерархия вложена в  $\mathbf{IP}$ .



## Глава 13

# Доказательства с нулевым разглашением

В главе про интерактивные доказательства мы рассматривали ситуацию, когда прuver превосходит верификатора по вычислительной силе. Теперь рассмотрим ситуацию, когда прuver имеет ту же вычислительную силу, но обладает некоторым секретом: знает изоморфизм графов, правильную раскраску в 3 цвета, пароль с данным хеш-значением, короткое доказательство теоремы или сертификат к любой другой задаче из **NP**. Если прuver раскроет этот секрет, то последствия могут быть плачевными: например, подставной сервер украдёт пароль и с его помощью завладеет настоящим аккаунтом. С другой стороны, настоящий сервер обязан убедиться, что клиент знает пароль, прежде чем пускать его в аккаунт. Таким образом, нужно разработать процедуру доказательства знания секрета без разглашения самого секрета. Мы будем требовать большего: не будет разглашаться не только весь секрет, но и какая-либо частичная информация о нём, которая могла бы помочь, например, в подборе пароля. Можно сказать, объём возникшего в ходе протокола нового знания нулевой.

### 13.1 Идея нулевого разглашения

*Антидоказательство с нулевым разглашением: протокол общения, при котором выявляется всё, что знает прuver, кроме того, что хочет узнать верификатор. Успешно применяется на большинстве линий по обслуживанию клиентов.*

Alexander W. Dent, [40]

Мы будем работать в той же парадигме интерактивных протоколов, что и раньше. Верификатор будет вероятностным и вычислимым за полиномиальное время, прuver же может возвращать значения любой функции. Отличие состоит в том, что у прuverа теперь тоже будет предписанное, «правильное» поведение, которое также будет вероятностным. Поэтому вместо двух условий теперь будет три: во-первых, если обе стороны действуют согласно предписанию, то верификатор скорее всего примет протокол; во-вторых, если прuver пытается доказать неверное утверждение, то при любых его действиях верификатор скорее всего отвергнет протокол; в-третьих, любой верификатор за счёт участия в протоколе

не узнает ничего, кроме ответа на исходный вопрос. Последнее формализуется следующим образом: в процессе общения верификатор узнал значение некоторой случайной величины. Величину с «примерно таким же» распределением он мог бы породить и без общения с прувером. Иными словами, верификатор мог бы самостоятельно сгенерировать нечто похожее на общение с настоящим прувером. Эту общую идею нужно уточнять по нескольким направлениям:

- Что именно должен сгенерировать верификатор;
- Какими возможностями обладает верификатор при общении с прувером и при моделировании распределения;
- Насколько похожими должны быть сгенерированное распределение и настоящее.

На первый вопрос можно отвечать двумя способами: либо верификатор должен сгенерировать весь диалог, либо только ответ. Кажется, что первое существенно сложнее второго, но если посмотреть на весь класс задач, то подходы эквивалентны. Идея заключается в том, что для каждого бита диалога можно рассмотреть другой диалог, где этот бит и будет ответом.

Верификатор, как обычно, предполагается вероятностным, но полиномиально ограниченным и при общении с прувером, и при генерации распределения. При этом в стандартной модели он может как угодно отклоняться от предписанного протокола, например выбирать случайные биты не простым киданием монетки, а некоторым хитрым способом. Рассматривают также модели с честным верификатором, который полностью соблюдает протокол,<sup>1</sup> но может попытаться что-то вычислить на основе промежуточных сообщений.

Для близости настоящего и сгенерированного распределений используют три основных подхода: либо они должны быть полностью одинаковы, либо близки в статистическом смысле, либо вычислительно неотличимы. Соответствующие классы протоколов называют доказательствами с совершенно, статистически или вычислительно нулевым разглашением.

### 13.1.1 Передача знания или передача информации?

На бытовом уровне «передача знания» и «передача информации» — почти синонимы. Разве что, «знание» скорее относится к чему-то важному или полезному, а «информация» может быть любой. Мы будем понимать под «знанием» результат, который нельзя получить самостоятельно, а под «информацией» — информацию в классическом шенноновском смысле. Например, если прувер и верификатор знают некоторый граф, и прувер присылает гамильтонов цикл, то он сообщает новое знание о гамильтоновости графа: при  $P \neq NP$  верификатор сам не может его получить. С другой стороны, новой информации нет: вся информация про гамильтоновость есть в исходном графе. Если же прувер присылает результат 1000 бросаний монетки, то передаётся 1000 бит информации. Но никакой передачи знания тут нет: верификатор мог и сам подбросить монетку.

<sup>1</sup>Например, использует сертифицированное программное обеспечение.



## 13.2 Совершенно нулевое разглашение

При совершенно нулевом разглашении верификатор должен уметь генерировать случайную величину с точно таким же распределением, что и диалог с прuverом. Сначала мы опишем основную идею метафорически, затем приведём классический протокол для изоморфизма графов, только после этого дадим формальное определение и, наконец, докажем, что построенный протокол ему соответствует.

### 13.2.1 «Сезам, откройся!»

В этом подразделе пересказывается статья «Как рассказать детям о доказательствах с нулевым разглашением» Жан-Жака Кискатера и Луи Гийю [107]. Полный список соавторов (а это жёны и дети основных авторов, а также редактор английской версии) приведён в библиографии.

Али-Баба — торговец на багдадском рынке. Однажды вор выхватил у него кошелёк со всей дневной выручкой и бросился наутёк. Али-Баба погнался за ним и вскоре оказался в тёмной пещере. Али-Баба зажёл факел, стал продвигаться вглубь и оказался перед развилкой. Он не видел, куда убежал разбойник, но наудачу пошёл по левому ходу. Вскоре ход закончился тупиком. Али-Баба вернулся к развилке и пошёл по правому ходу, тоже приведшему в тупик. Так и не поймав вора, Али-Баба решил, что тот спрятался справа, но сбежал, пока Али-Баба проверял левый ход. На следующий день история повторилась, но на этот раз Али-Баба побежал по правому ходу. Увы, вор снова удрал. Так продолжалось на третий, четвёртый, ..., сороковой день: каждый раз Али-Баба выбирал направление произвольным образом, но разбойник убегал. На сорок первый день Али-Баба не пошёл на рынок, а затаился в пещере, в одном из тупиков. Через некоторое время он увидел вора с добычей. Дойдя до тупика, он прошептал: «Сезам, откройся!» — и тут огромная каменная глыба сдвинулась с места, открыв проход в другой тупик. Али-Баба сразу понял, как вору удавалось убегать 40 дней подряд. Немного поэкспериментировав, Али-Баба сумел изменить заклинание, и на следующий, 42-й день поймал вора. Эту историю он описал в манускрипте, не раскрыв нового заклинания, но оставив подсказки в тексте.

Много веков спустя археологи раскопали эту пещеру, нашли дверь, и самый проникательный из них, Мик Али,<sup>2</sup> сумел разгадать заклинание из манускрипта Али-Бабы. Опасаясь, что секрет украдут, он никому не рассказывает о нём до подготовки научной статьи. Тем не менее, краткое сообщение утекло в прессу и стало сенсацией. Многие телекомпании хотят подготовить сюжеты, и одной из них удалось уговорить Мика Али. Чтобы не разглашать секрет, он повторяет в сюжете историю с Али-Бабой и вором: под камеры убегает в пещеру, затем без камеры выбирает один из двух ходов, затем оператор с камерой идёт в пещеру, также выбирает один из двух ходов бросанием монетки, но не встречает там Мика Али.

Другая компания также хочет снять такой сюжет, но Мик Али наотрез отказывается. И тут они понимают, что сюжет можно снять без участия учёного. Они находят похожего актёра и делают всё то же самое. Правда, актёр не знает заклинания, и примерно в половине случаев оператор оказывается в том же туннеле, что и актёр. Но эти дубли просто вырезаются при монтаже, так что итоговый фильм получается точно таким же. Этот опыт показывает, что Мик

---

<sup>2</sup>Здесь авторы явно имели в виду Сильвио Микали.

Али действительно ничего не разгласил: всё, что он продемонстрировал, можно было проделать и без него. Отличие состоит лишь в том, что у него всё всегда получается с первого дубля.

### 13.2.2 Протокол для задачи об изоморфизме графов

Изложенную выше идею можно после небольшой модификации представить в виде протокола для задачи об изоморфизме графов  $GI = \{G_0, G_1 \mid G_0 \simeq G_1\}$ . А именно, прuver присылает случайный граф  $H$ , изоморфный  $G_0$  (а также  $G_1$ ). Верификатор бросает монетку и сообщает результат прuverу. Если выпал орёл, то прuver предоставляет изоморфизм между  $H$  и  $G_0$ , а если выпала решка, то изоморфизм между  $H$  и  $G_1$ . Верификатор проверяет, что изоморфизм правильный. Если графы действительно изоморфны, то прuver будет успешен всегда, а если не изоморфны, то с вероятностью не больше половины. При этом верификатор узнает лишь изоморфизм между  $H$  и одним из графов  $G_b$ , что ничего ему не скажет об изоморфизме между  $G_0$  и  $G_1$ .

Заметим, что для реализации протокола со стороны прuverа не нужно производить сверхполиномиальные вычисления, достаточно знать изоморфизм  $\varphi: G_1 \rightarrow G_0$ . В таком случае протокол осуществляется так:

0. Обе стороны узнают  $G_0$  и  $G_1$ , а прuver — ещё и  $\varphi$
1. Прuver выбирает случайное  $\sigma \in S_n$  и посылает  $H = \sigma(G_0)$  верификатору;
2. Верификатор выбирает случайный бит  $b \in \{0, 1\}$  и посылает его прuverу;
3. Прuver посылает  $\tau = \sigma \circ \varphi^b$ , т.е.  $\sigma$  при  $b = 0$  и  $\sigma \circ \varphi$  при  $b = 1$ ;
4. Верификатор проверяет, что  $\tau(G_b) = H$ .

Если прuver действует по алгоритму, то при  $b = 0$  выполнено  $\tau(G_b) = \sigma(G_0) = H$ , а при  $b = 1$  выполнено  $\tau(G_b) = \sigma(\varphi(G_1)) = \sigma(G_0) = H$ , т.е.  $\tau(G_b) = H$  всегда. Если же на самом деле  $G_0 \not\simeq G_1$ , то при любом  $H$  либо  $H \not\simeq G_0$ , либо  $H \not\simeq G_1$ , так что с вероятностью не меньше  $\frac{1}{2}$  прuver не сможет предоставить нужное  $\tau$ . Осталось показать, почему не разглашается знание. Для этого мы наконец дадим определение совершенно нулевого разглашения.

### 13.2.3 Формальное определение

**Определение 13.1.** Пусть задана система интерактивных доказательств  $(P, V)$ . Она обладает *совершенно нулевым разглашением* (perfect zero knowledge), если выполнено следующее условие: для любого полиномиального вероятностного интерактивного алгоритма  $V^*$  существует полиномиальный вероятностный неинтерактивный алгоритм  $M^*$ , возвращающий либо двоичное слово, либо специальный символ  $\perp$ , такой что:

1. Для любого входа  $x$  вероятность того, что  $M^*(x) = \perp$ , не превосходит  $\frac{1}{2}$ ;
2. Для любого входа  $x$  распределение  $(V^*)^P(x)$  и условное распределение  $M^*(x) \mid M^*(x) \neq \perp$  совпадают.

Заметим, что исходный верификатор  $V$  в определении не использован, так что на самом деле нулевое разглашение есть свойство прuverа.

**Определение 13.2.** Классом **PZK** называется класс языков, для которых существует интерактивная система доказательств в смысле определения 12.4, обладающая совершенно нулевым разглашением.

Про класс **PZK** можно сразу сказать следующее:

**Утверждение 13.3.**  $\text{BPP} \subset \text{PZK} \subset \text{PSPACE}$ .

*Доказательство.* Для языков из **BPP** существует тривиальный протокол: верификатор не получает сообщений от прувера, а сразу запускает вероятностный алгоритм и возвращает ответ. В этом случае даже не нужна возможность возврата  $\perp$ : можно просто взять  $M^* = V^*$ .<sup>3</sup> Ну а второе вложение следует из  $\text{PZK} \subset \text{IP}$  и  $\text{IP} = \text{PSPACE}$ .  $\square$

Данное выше определение можно критиковать следующим образом: в ходе диалога верификатор узнаёт не только свой ответ, но и все промежуточные сообщения прувера в сочетании с собственными случайными битами. Не может ли быть так, что знание будет передано в ходе промежуточного общения? Для конкретного верификатора такое может случиться, но требование существования симулятора для каких угодно верификаторов делает свойство эквивалентным. Более подробно, рассмотрим такое определение:

**Определение 13.4.** Пусть задана система интерактивных доказательств  $(P, V)$ . Случайной величиной  $\text{VIEW}_V^P(x)$  назовём кортеж  $(r, m_2, m_4, \dots, m_{2k})$ , где  $r$  — частные случайные биты верификатора,  $m_2, m_4, \dots, m_{2k}$  — сообщения прувера для общего входа  $x$ . Будем говорить, что система обладает совершенно нулевым разглашением, если для любого полиномиального вероятностного интерактивного алгоритма  $V^*$  существует полиномиальный вероятностный неинтерактивный алгоритм  $M^*$ , возвращающий либо двоичное слово, либо специальный символ  $\perp$ , такой что:

1. Для любого входа  $x$  вероятность того, что  $M^*(x) = \perp$ , не превосходит  $\frac{1}{2}$ ;
2. Для любого входа  $x$  распределение  $\text{VIEW}_{V^*}^P(x)$  и условное распределение  $M^*(x) \mid M^*(x) \neq \perp$  совпадают.

**Теорема 13.5.** Определения 13.1 и 13.4 эквивалентны.

*Доказательство.* По  $\text{VIEW}_{V^*}^P(x)$  можно однозначно восстановить ответ верификатора, поэтому из определения 13.4 следует определение 13.1. Обратно, применим определение 13.1 к верификатору, возвращающему  $\text{VIEW}_{V^*}^P(x)$ . Получим определение 13.4.  $\square$

### 13.2.4 Корректность протокола

В этом разделе мы формально докажем такую теорему:

**Теорема 13.6.**  $\text{GI} \in \text{PZK}$ .

---

<sup>3</sup>Наличие симулятора, никогда не возвращающего  $\perp$ , можно было бы рассмотреть отдельно, например, под названием «абсолютно нулевое разглашение». Однако примеров языков с таким свойством за пределами **BPP** неизвестно.

*Доказательство.* Докажем, что описанный выше протокол действительно соответствует определению. Во-первых, если  $G_0 \simeq G_1$ , то верификатор обязательно выдаст 1. Во-вторых, если  $G_0 \not\simeq G_1$ , то при любом  $H$  максимум одно из утверждений  $H \simeq G_0$  и  $H \simeq G_1$  может быть верным. Поскольку верификатор выбирает  $b$  независимо от  $H$ , то пруввер сможет предоставить изоморфизм с вероятностью не больше  $\frac{1}{2}$ . Осталось доказать совершенно нулевое разглашение. Трудность заключается в том, что рассуждение должно работать не только для «честного» верификатора  $V$ , выбирающего  $b$  равновероятно и независимо от  $H$ , но и для любого другого полиномиального  $V^*$ . Разумеется, симулятор  $M^*$  будет использовать  $V^*$  как подпрограмму.

На момент выбора  $b$  алгоритм  $V^*$  знает вход  $x = (G_0, G_1)$ , свои случайные биты  $r$  и присланный пруввером граф  $H$ . Он вычисляет  $b = V^*(x, r, H)$  и узнаёт изоморфизм  $\tau$  между  $G_b$  и  $H$ . Симулятор  $M^*$  будет делать следующее: получив  $x$  и  $r$ , он при помощи новых случайных битов генерирует независимые  $c \in \{0, 1\}$  и  $\sigma \in S_n$ , вычисляет  $H = \sigma(G_c)$ , а затем  $b = V^*(x, r, H)$ . Если  $b \neq c$ , он возвращает  $\perp$ , иначе  $\sigma$ . Требуется доказать, что  $M^*$  удовлетворяет требованиям определения 13.1.

Во-первых, заметим, что условные распределения  $H$  при  $c = 0$  и при  $c = 1$  совпадают: и то, и другое — равномерное распределение по всем графам, изоморфным  $G_0$  (и  $G_1$ ). По формуле Байеса это означает, что при любом фиксированном  $H$  условные вероятности того, что  $c = 0$ , и того, что  $c = 1$ , равны по  $\frac{1}{2}$ . Это значит, что  $\Pr[V^*(x, r, H) = c] = \frac{1}{2}$ , т.е. вероятность  $\perp$  равна  $\frac{1}{2}$ , что и требовалось.

Во-вторых, требуется доказать совпадение распределений. Мы будем рассматривать распределения на наборах  $(x, r, H, \tau)$ . Докажем, что и распределение в настоящем диалоге, и распределение в симуляторе устроено так:  $r$  распределено равномерно,  $H$  распределено равномерно и независимо от  $r$  среди всех графов, изоморфных  $G_i$ , а  $\sigma$  есть изоморфизм между  $G_{V^*(x, r, H)}$  и  $H$ , распределённый равномерно среди всех таких изоморфизмов независимо от  $r$  и  $H$ . Для настоящего диалога это следует вот из чего:  $H$  присылает пруввер, поэтому распределение действительно равномерно и независимо от  $r$ . Более того, при фиксированном  $H$  перестановка  $\sigma$  распределена равномерно среди всех изоморфизмов  $G_0$  и  $H$ , а перестановка  $\sigma \circ \varphi$  — равномерно среди всех изоморфизмов  $G_1$  и  $H$ . Поскольку при  $V^*(x, r, H) = 0$  пруввер присылает  $\sigma$ , а при  $V^*(x, r, H) = 1$  —  $\sigma \circ \varphi$ , требуемое условие на  $\tau$  соблюдено.

Наконец, проверим распределение для симулятора. Граф  $H$  выбирается независимо от  $r$  и распределён равномерно. Если  $M^*(x) \neq \perp$ , то  $\sigma$  является изоморфизмом между  $G_{V^*(x, r, H)}$  и  $H$ , что и требовалось. При этом распределение на таких изоморфизмах равномерное, поскольку граф  $H$  мог получиться при выборе любого из них.  $\square$

### 13.3 Статистически нулевое разглашение

В предыдущем разделе симулятор обязательно генерировал то же самое распределение, что и в настоящем диалоге. Однако использовать можно не распределение само по себе, а лишь конкретные реализации. Но тогда близкие распределения дадут близкие результаты. Поэтому наличие симулятора, генерирующего близкое распределение, достаточно для того, чтобы считать разглашение нулевым. Это мотивирует следующее распределение:

**Определение 13.7.** Пусть задана система интерактивных доказательств  $(P, V)$ . Она обладает *статистически нулевым разглашением* (statistical zero knowledge или almost-perfect zero knowledge), если выполнено следующее условие: для любого полиномиального вероятностного интерактивного алгоритма  $V^*$  существует полиномиальный вероятностный неинтерактивный алгоритм  $M^*$ , такой что для любого входа  $x$  распределения  $(V^*)^P(x)$  и  $M^*(x)$  статистически близки. Это означает, что для любого полинома  $p(\cdot)$ , для всех достаточно больших  $n$  при  $|x| > n$  для любого множества  $S$  вероятности событий  $(V^*)^P(x) \in S$  и  $M^*(x) \in S$  отличаются меньше чем на  $\frac{1}{p(n)}$ .

Заметим, что здесь  $M^*$  уже никогда не возвращает  $\perp$ . Такая возможность ничего не добавит.

**Определение 13.8.** Классом **SZK** называется класс языков, для которых существует интерактивная система доказательств, обладающая совершенно нулевым разглашением.

Место класса **SZK** задаёт следующее утверждение:

**Утверждение 13.9.**  $\text{PZK} \subset \text{SZK} \subset \text{PSPACE}$ .

*Доказательство.* Второе вложение следует из  $\text{SZK} \subset \text{IP}$  и  $\text{IP} = \text{PSPACE}$ . Чтобы доказать первое вложение, нужно превратить систему доказательств с совершенно нулевым разглашением в систему со статистически нулевым разглашением. Это делается при помощи многократного повторения: нужно запускать симулятор  $M^*$  (из определения **PZK**) до получения первого результата, отличного от  $\perp$ , и вернуть этот результат. Если же  $\perp$  появился  $n$  раз подряд, то нужно вывести какую-нибудь предустановленную константу. Вывод симулятора на каждом этапе имеет одно и то же распределение, которое и должно быть на выходе.  $\square$

## 13.4 Вычислительно нулевое разглашение

Мало получить новое знание, нужно ещё суметь его использовать или хотя бы понять, что оно действительно новое. Допустим, оно будет в дальнейшем использовано в каком-то полиномиальном вероятностном алгоритме. В таком случае даже статистическая близость к симуляции не нужна, достаточно вычислительной неотличимости. Действительно, в любой алгоритм, что-то делающий с результатом общения, можно подставить выход симулятора, и получится примерно то же самое. Иначе именно этот алгоритм мог бы отличить диалог от симуляции.

### 13.4.1 Неформальный протокол для задачи о 3-раскраске

## 13.5 Амплификация

## 13.6 Исторические замечания и рекомендации по литературе

Вопрос о нулевом разглашении был поставлен в первой же работе по интерактивным доказательствам Голдвассер, Микали и Ракоффа [67].

## 13.7 Задачи и упражнения

**13.1. Вариации в определении PZK.** Докажите, что класс **PZK** не изменится, если в определении 13.1 заменить  $\frac{1}{2}$  на  $\frac{1}{2^{\text{poly}(|x|)}}$ .

**13.2. Неизоморфизм графов.** Является ли стандартный протокол доказательства неизоморфизма графов протоколом с нулевым разглашением?

**13.3. Вариации в определении SZK.** Докажите, что класс **SZK** не изменится, если в определении 13.7 разрешить симулятору  $M^*$  возвращать  $\perp$  с вероятностью не больше  $\frac{1}{2}$  и требовать близости к условному распределению при  $M^* \neq \perp$ .

**13.4. Судoku с нулевым разглашением.** Обобщённым судoku называется такая задача: в квадрате  $n^2 \times n^2$  в некоторых клетках расставлены числа от 1 до  $n^2$ . Вопрос: можно ли заполнить оставшиеся клетки числами от 1 до  $n^2$ , так чтобы в каждой строке, каждом столбце, а также каждом из  $n^2$  «выровненных» квадратов  $n \times n$  каждое число встречалось бы по одному разу. В стандартном судoku  $n = 3$ . Известно, что эта задача **NP**-полна. Предложите протокол доказательства существования решения с вычислительно нулевым разглашением, не использующий сводимость к какой-либо другой задаче.

# Глава 14

## Вероятностно проверяемые доказательства

Обычное математическое доказательство выглядит как цепочка формул, каждая из которых есть либо аксиома, либо выводится из предыдущих. Ошибка даже в одном переходе может сделать вывод неверным, так что проверять нужно все шаги. Если выбрать случайно половину строчек и проверить только их, то можно быть уверенным в истинности вывода также лишь наполовину. Оказывается, можно построить систему доказательств на совершенно иных принципах, после чего становится достаточно проверить лишь небольшое число битов и с большой степенью надёжности отделить верное доказательство от неверного. Более того, само доказательство может быть сделано не слишком длинным, новая длина будет полиномом от старой. Удивительным образом оказывается, что такие доказательства связаны с задачами аппроксимации, а именно, позволяют доказать их **NP**-трудность. Более того, два подхода в некотором смысле эквивалентны.

### 14.1 Понятие вероятностно проверяемого доказательства

Можно сказать, что понятие корректности доказательства определяется через процедуру его проверки. Так мы поступим и в данном случае. Назовём **РСР**-верификатором вероятностный алгоритм  $V$  со входом  $x$ , имеющий произвольный доступ к строке  $\pi$ . Произвольный доступ означает, что алгоритм может вычислить номер  $i$  и за один шаг получить значение бита  $\pi_i$ . Мы будем рассматривать модель с адаптивными запросами, когда запрошенный номер бита может зависеть от результатов предыдущих запросов. В конце концов верификатор выдаёт бинарный ответ: 0 или 1, который мы обозначаем через  $V^\pi(x)$ . Далее определим, как верификатор распознаёт язык.

**Определение 14.1.** Пусть  $A$  — некоторый язык. **РСР**-верификатор  $V$  распознаёт язык  $A$  с параметром полноты  $c$  и точности  $s$ , если:

- Для каждого  $x \in A$  найдётся доказательство  $\pi$ , такое что  $\Pr [V^\pi(x) = 1] \geq 1 - c$ ;
- Если  $x \notin A$ , то для любого  $\pi$  выполнено  $\Pr [V^\pi(x) = 1] \leq s$ .

Помимо времени работы, нас будут интересовать два ресурса: количество запросов к доказательству и количество использованных случайных битов.

- 14.2 Приближённое решение NP-трудных задач
- 14.3 Эквивалентность двух подходов
- 14.4 Экспоненциальная РСР-теорема
- 14.5 Исторические замечания и рекомендации по литературе
- 14.6 Задачи и упражнения



## Глава 15

# Рациональные интерактивные доказательства

При изучении интерактивных доказательств мы изучали два варианта поведения прuverа: честное, когда прuver отвечает именно то, что от него требуется по протоколу, и «вредное», когда прuver делает всё, чтобы обмануть верификатора. При взгляде через призму теории игр и математической экономики возникает вопрос: а что, если прuver будет преследовать собственный интерес, а верификатор составит контракт с правильными стимулами? Иными словами, на ситуацию предлагается посмотреть с точки зрения теории разработки механизмов: верификатор играет роль принципала, а прuver — агента. При этом верификатор, назначающий награду, вычислительно ограничен, а прuver — нет. Более-менее ясно, что общие вычислительные возможности не расширятся: если верификатор будет платить только за верное доказательство, то неверного ему не предоставят. Оказывается, однако, что можно значительно сэкономить на числе раундов взаимодействия.

### 15.1 Однораундовые доказательства

### 15.2 Многораундовые доказательства

### 15.3 Исторические замечания и рекомендации по литературе

### 15.4 Задачи и упражнения



# Приложение А

## Математические сведения

В этом приложении собраны математические определения и теоремы, использующиеся в книге. Скорее всего читатель знаком со всеми или почти всеми этими понятиями и фактами, но для точного понимания и полноты изложения мы приводим все определения и формулировки, а также избранные доказательства.

### А.1 Множества и логика

Теория множеств является фундаментом всей математики.

### А.2 Комбинаторика

**Определение А.1.** *Перестановкой* на  $n$  элементах называется кортеж длины  $n$ , в котором каждое из чисел  $1, \dots, n$  встречается по одному разу. *Подстановкой* на  $n$  элементах разывается биекция из множества  $\{1, \dots, n\}$  в себя.

Перестановки и подстановки естественным образом соответствуют друг другу: подстановке  $\sigma$  соответствует перестановка  $(\sigma(1), \sigma(2), \dots, \sigma(n))$ , и наоборот.

### А.3 Теория графов

#### А.3.1 Вершины и рёбра

**Определение А.2.** *Ориентированным графом* или *оргафом* называется пара  $(V, E)$ , где  $V$  — некоторое конечное множество,  $E \subset (V \times V) \setminus \{(v, v) \mid v \in V\}$ . Элементы множества  $V$  называются *вершинами* или узлами, элементы множества  $E$ , т.е. упорядоченные пары несовпадающих вершин, — *рёбрами* или дугами. Если вместо упорядоченных пар рассмотреть неупорядоченные, то граф называют неориентированным, или просто графом.<sup>1</sup> Если не запрещаются пары из одинаковых вершин в качестве рёбер, то говорят о *псевдографе*, а сами такие рёбра называют *петлями*.

**Определение А.3.** *Степенью* вершины  $v$  в графе называется количество таких  $w$ , что  $\{v, w\} \in E$ . Вершина степени 1 называется *висячей*. В орграфе различают входящую степень  $(\#\{w \mid (w, v) \in E\})$ , исходящую степень  $(\#\{w \mid$

---

<sup>1</sup>Некоторые авторы используют слово «дуги» исключительно для орграфов, а «рёбра» оставляют для обычных.

$(v, w) \in E\}$ ) и полную степень (сумму входящей и исходящей). Вершина нулевой входящей степени называется *источником*, а нулевой исходящей степени — *стоком*. В псевдографе обычно предполагают, что петля увеличивает степень на 1, а не на 2.

**Лемма А.4** (Лемма о рукопожатиях). *В любом графе количество вершин нечётной степени чётно.*

*Доказательство.* Каждое ребро вносит единичный вклад в степень каждого из своих концов. Поэтому сумма степеней всех вершин в графе равна удвоенному числу рёбер и, следовательно, чётна. С другой стороны, если сумма целых чисел чётна, то нечётных среди них должно быть чётное количество.  $\square$

**Определение А.5.** Граф называется *полным*, если множество его рёбер содержит все возможные пары вершин. Полный граф на  $n$  вершинах обозначается  $K_n$ .

### А.3.2 Изоморфизмы

**Определение А.6.** Пусть  $G_0 = (V_0, E_0)$  и  $G_1 = (V_1, E_1)$  суть два графа (или орграфа, и т.д.) Они называются *изоморфными*, если существует биекция  $\varphi: V_0 \rightarrow V_1$ , такая что  $(x, y) \in E_0$  тогда и только тогда, когда  $(\varphi(x), \varphi(y)) \in E_1$ . Обозначение:  $G_0 \simeq G_1$ . Сама такая биекция называется *изоморфизмом*. В случае, когда  $G_0 = G_1$  изоморфизм называется *автоморфизмом*.

**Утверждение А.7.** *Наличие изоморфизма есть отношение эквивалентности на множестве графов.*

### А.3.3 Маршруты в графах

## А.4 Теория вероятностей

## А.5 Алгебра и теория чисел

# Приложение В

## Ответы, указания, решения

### Глава 1

**1.1.** В качестве  $g$  можно взять что угодно, например  $g(n) = n$ . А  $f$  должна колебаться между  $g$  и некоторой функцией  $o(g(n))$ , при этом сохраняя монотонность.

**1.3.**

	$o$	$O$	$\Theta$	$\Omega$	$\omega$
$o$	$o$	$o$	$o$	$?$	$?$
$O$	$o$	$O$	$O$	$?$	$?$
$\Theta$	$o$	$O$	$\Theta$	$\Omega$	$\omega$
$\Omega$	$?$	$?$	$\Omega$	$\Omega$	$\omega$
$\omega$	$?$	$?$	$\omega$	$\omega$	$\omega$

**1.4.** Например,  $2^n = o(2^{2n})$ , хотя  $n = \Theta(2n)$ . Поэтому  $O$ ,  $\Omega$ ,  $\Theta$  могут не сохраниться (и перейти во что угодно). При логарифмировании, напротив, могут разрушиться  $o$  и  $\omega$ . Можно лишь гарантировать, что  $o$  останется  $O$ , а  $\omega$  останется  $\Omega$ .

**1.5.** Общая таблица будет такая же, как для умножения.

**1.7.** Будем нумеровать клетки плоскости по спирали, начиная с центра. Тогда максимальный номер, до которого может добраться машина, имеет порядок  $T(n)^2$ , но номера любых соседних клеток отличаются не больше, чем на  $T(n)$ . Поэтому один шаг работы на клетчатой плоскости моделируется за время  $O(T(n))$ , а всё вычисление — за  $O(T(n)^2)$ .

**1.8.** Поскольку

**1.9.** Машина с двумя лентами, распознающая язык PAL за линейное время, строится очень просто. Достаточно скопировать входное слово на вторую ленту, перевести указатель на первой ленте в начало и затем сравнивать биты, идя по двум лентам в противоположных направлениях.

Докажем, что на одноленточной машине Тьюринга на некотором входе потребуется  $\Omega(n^2)$  шагов. Для удобства будем рассматривать входы длины  $4n$ . Рассмотрим лишь слова вида  $x0^{2n}y^R$ , где  $|x| = |y| = n$ . Слово такого вида является палиндромом тогда и только тогда, когда  $x = y$ . Неформально говоря, для проверки этого факта нужно перенести информацию об  $x$  ( $n$  битов) на расстояние хотя бы  $2n$ , что с учётом конечности памяти, заключённой во внутреннем

состоянии, потребует  $\Omega(n^2)$  операций. Теперь проведём формальное рассуждение.

Назовём протоколом работы машины в точке  $i$  список состояний и направлений, в которых машина пересекала границу между  $i$ -ой и  $(i+1)$ -ой ячейками.

*Утверждение.* Если  $x \neq z$ , то машина, распознающая язык **PAL**, имеет разные протоколы на  $x0^{2n}x^R$  и  $z0^{2n}z^R$  для любого  $i \in [n, 3n]$ .

*Доказательство.* Если машина работает правильно, то она принимает и  $x0^{2n}x^R$ , и  $z0^{2n}z^R$ . Докажем, что если протоколы в точке  $i$  одинаковы, то такой же протокол (и тот же результат) будет для  $x0^{2n}z^R$  и  $z0^{2n}x^R$ , так что машина ошибётся на этих входах. Действительно, работа машины на замкнутом отрезке полностью описывается начальным состоянием на этом отрезке и протоколом работы на его границах. Если протоколы в точке  $i$  одинаковы, то машина слева от этой точки будет одинаково работать на всех входах, начинающихся с  $x0^{i-n}$ , а справа — на всех входах, оканчивающихся на  $0^{3n-i}z^R$ . Значит, на входе  $x0^{2n}z^R$  левее точки  $i$  машина будет работать так же, как на входе  $x0^{2n}x^R$ , а правее — так же, как на входе  $z0^{2n}z^R$ . Поскольку и там, и там ответ положительный, то такой же ответ будет на  $x0^{2n}z^R$ , т.е. машина сработает неправильно. Значит, протоколы должны быть разными.

Вернёмся к доказательству основного утверждения. Поскольку у машины Тьюринга конечное число состояний (пусть  $q$ ), то любой протокол можно записать в алфавите из  $2q$  символов ( $q$  состояний и 2 направления). Поскольку для всех  $2^n$  слов  $x \in \{0, 1\}^n$  протоколы должны быть разными, то для любой точки  $i$  найдётся протокол длины  $\log_{2q} 2^n - 1 = \Omega(n)$ . (Если самый длинный протокол содержит  $l$  символов, то общее число протоколов не превосходит  $1 + 2q + (2q)^2 + \dots + (2q)^l$ , что меньше  $(2q)^{l+1}$ . Отсюда  $2^n < (2q)^{l+1}$ , откуда и получаем нужное условие на  $l$ ). Более того, найдётся  $x$ , для которого все  $2n+1$  протоколов имеют длину  $\log_{2q} \frac{2^n}{2n+1} - 1 = \Omega(n - \log n) = \Omega(n)$ . (Действительно, по принципу Дирихле найдётся  $i$ , такой что  $i$ -ый протокол будет самым коротким для  $\frac{2^n}{2n+1}$  слов, из предыдущего для одного из этих слов он будет иметь длину  $\log_{2q} \frac{2^n}{2n+1} - 1$ , а все остальные протоколы для этого слова будут не короче). Ну а поскольку каждому символу каждого протокола соответствует отдельный шаг работы машины, общее число шагов на этом  $x$  составит  $\Omega(n^2)$ , что и было заявлено.

## Глава 2

**2.1.** Можно искусственно ограничить этим полиномом время работы машины: если ответа не получено, то выдавать отрицательный.

### 2.2.

- а) Нужно включить в сертификат разбиение на слова из  $A$  и отдельные сертификаты для каждого из них.
- б) Нужно воспользоваться методом динамического программирования. Достаточно хранить ответ для каждого подслова, а их квадратичное число.

**2.3.** Достаточно доказать, что  $\mathbf{NP} \cap \mathbf{coNP}$  замкнут относительно дополнения.

## Глава 3

**2.3.** Пусть  $L$  является унарным **NP**-полным языком. Значит,  $\text{SAT} \leq_p L$ . Пусть  $A$  — соответствующая сводимость, работающая за время  $p(n)$ . Приведём алгоритм, который решает задачу **SAT** за полиномиальное время, тогда  $\mathbf{P} = \mathbf{NP}$ . Будем считать, что формула помимо переменных содержит логические константы.

```

1 Функция CheckSAT
   Вход: Формула  $\varphi$  (зависящая от переменных и констант)
   Выход: 1, если  $\varphi$  выполнима, 0 иначе
   Вспомогательные данные: Таблица  $T$ , где  $T(k) = 1$ , если  $1^k \in L$ ,
                                $T(k) = 0$ , если  $1^k \notin L$  и  $T(k) = ?$ , если
                               принадлежность к  $L$  неизвестна
   /* Таблица  $T$  имеет длину  $p(|\varphi|)$ , она одна и та же для всех
      рекурсивных вызовов функции, изначально заполнена
      вопросами */
2 если  $\varphi$  не содержит переменных то
3   |   вычислить  $\varphi$ ;
4   |   возвратить вычисленное значение;
5 иначе
6   |    $\text{str} := A(\varphi)$ ;
7   |   если  $\text{str}$  содержит нули то
8   |   |   возвратить 0;
9   |   иначе если  $\text{str} = 1^k$  и  $T(k) \neq ?$  то
10  |   |   возвратить  $T(k)$ ;
11  |   иначе
12  |   |    $p$  — первая переменная, существенно входящая в  $\varphi$ ;
13  |   |    $\psi_0 := \varphi|_{p=0}$ ;
14  |   |   если CheckSAT( $\psi_0$ ) = 1 то
15  |   |   |    $T(k) := 1$ ;
16  |   |   |   возвратить 1;
17  |   |   конец условия
18  |   |   если  $T(k) \neq ?$  то возвратить  $T(k)$ ;
19  |   |    $\psi_1 := \varphi|_{p=1}$ ;
20  |   |    $T(k) := \text{CheckSAT}(\psi_1)$ ;
21  |   |   возвратить  $T(k)$ ;
22  |   конец условия
23 конец условия

```

Приведённый алгоритм корректен. В самом деле, по свойству сводимости  $\varphi \in \text{SAT} \Leftrightarrow A(\varphi) \in L$ . Если  $A(\varphi)$  не имеет вид  $1^k$ , то заведомо  $A(\varphi) \notin L$  и потому  $\varphi \notin \text{SAT}$ . Если же  $A(\varphi) = 1^k$ , то если  $T(k)$  определено, то  $T(k) = 1 \Leftrightarrow 1^k \in L \Leftrightarrow \varphi \in \text{SAT}$ . В случае если  $T(k)$  не определено, мы пользуемся тем, что  $\varphi$  выполнима тогда и только тогда, когда выполнима хотя бы одна из формул  $\varphi|_{p=0}$  и  $\varphi|_{p=1}$ . Поэтому рекурсивный вызов функции на  $\varphi|_{p=0}$  и  $\varphi|_{p=1}$  даёт корректный ответ в виде дизъюнкции (если ответ на первый вызов положительный, то второй вызов можно не делать).

Осталось доказать, что время работы будет полиномиальным. Для этого заметим, что изначально в таблице  $T$  содержится полиномиальное число вопросительных знаков. Каждый запуск процедуры заменяет один вопросительный

знак на булево значение (в строке ?? или в строке ??). При этом если на каком-то нижнем уровне рекурсии был получен результат, который был запрошен на верхнем уровне, то благодаря проверке в строке ?? второй рекурсивный вызов на вернем уровне не будет совершён. Таким образом, дерево рекурсивных вызовов будет иметь не больше  $p(n)$  точек бинарного ветвления. Значит, в нём не больше  $p(n) + 1$  листьев и не больше  $n(p(n) + 1)$  вершин. Таким образом, совершается не больше  $n(p(n) + 1)$  рекурсивных вызовов, и поскольку каждый из вызовов полиномиален, весь алгоритм также полиномиален.

## Глава 4

**4.1.** Проверьте, что доказательство теоремы об иерархии по времени выдерживает переход к вычислениям с оракулом.

## Глава 5

**5.1.** Оба эти класса равны **PSPACE**.

## Глава 6

**6.1.** Воспользуйтесь определением полиномиальной иерархии через оракулы.

**6.2.**

## Глава 7

**7.1.** Среди всех булевых функций, требующих схем экспоненциального размера, рассмотрите лексикографически первую. Очевидно, соответствующий язык не лежит в  $\mathbf{P}/_{\text{poly}}$ . Докажите, что эту функцию тем не менее можно найти на экспоненциальной памяти.

## Глава 8

**8.1.** Сделайте ошибку экспоненциально малой, выберите универсальное  $r$  и запаяйте его в схему.

**8.2.** Первая часть следует из линейности математического ожидания. Далее значения переменных нужно выбирать так, чтобы условное ожидание числа выполненных скобок не уменьшалось.

**8.3.** Ответ \* означает, что точный ответ не удалось найти в разумные сроки.



8.4.

8.5. Предположите, что  $\text{SAT} \in \text{BPP}$ , рассмотрите алгоритм проверки выполнимости с достаточно маленькой ошибкой. Затем с помощью этого алгоритма попробуйте найти выполняющий набор, а в конце проверьте, верно ли получилось.

8.6. Используйте  $\text{RP} \subset \text{NP}$  и  $\text{ZPP} = \text{RP} \cap \text{coRP}$ .

8.7. Нужно сделать достаточно маленькую ошибку, а потом выбрать случайные биты при помощи оракула.

8.8.

## Глава 9

## Глава 10

## Глава 11

11.1. Путём «пометки» вершин последовательно установите, куда должна перейти какая вершина

11.2. Нет, потому что нельзя проверить, что клика наибольшая.

11.3.

а) В одну сторону нужно взять  $W \equiv 1$ . В другую — выразить задачу поиска  $y$  или  $z$  через 1 предикат.

б) Используйте предыдущий пункт.

11.4. В одну сторону это следует из  $\text{TFNP} = \text{F}(\text{NP} \cap \text{coNP})$ . В другую: нужно свести поисковый вариант SAT к  $W$  и проанализировать, как такая сводимость работает на невыполнимых формулах.

11.5. В качестве отношения соседства можно рассмотреть близость в метрике Хемминга, а в качестве издержек — хемминговское же расстояние до оптимума.

11.6. Рассмотрите задачу о выполнимости булевой формулы с  $n$  переменными. Любой набор значений воспринимайте как двоичную запись числа от 0 до  $2^n - 1$ . Считайте, что  $N(\varphi, y) = \{y - 1\}$  при  $y > 0$  и  $N(0) = \emptyset$ . Пусть  $c(\varphi, y) = -1$ , если  $y$  — выполняющий набор, и  $c(\varphi, y) = y$  иначе.

11.8.

а) Подойдёт  $K_4$ .

б) Во втором цикле есть какое-то ребро, которого нет в первом, и для него теорема Смита тоже верна.

в) Превратите предыдущее рассуждение в PPA-алгоритм поиска.

**11.9.** Рассмотрите граф из всех наборов рёбер, которые либо являются совершенными паросочетаниями, либо одну выделенную вершину не покрывают, а какую-то другую покрывают дважды.

**11.10.** Чётность полной степени совпадает с чётностью баланса, поэтому задача в **РРА**. Для сводимости к ней нужно ориентировать рёбра произвольным образом.

**11.11.** В одну сторону достаточно ориентировать все рёбра из левой доли в правую. В другую сторону, например, можно заменить каждую вершину на несколько графов  $K_p$ . Каждое ребро  $(x, y)$  в исходном графе заменяется на паросочетание между левой долей одного из графов, полученных из вершины  $x$ , и правой долей одного из графов, полученных из вершины  $y$ . Для баланса внутренние рёбра в этих графах убираются.

**11.12.** Конструкция аналогична теореме 11.6, надо только правильно ориентировать все рёбра.

**11.13.**

- а) Если вершина несбалансирована на  $\pm k$ , то она породит  $k$  отдельных концов источников или стоков. При  $k > 1$  по одному источнику может быть найден другой источник, получившийся из той же вершины, что не даст ответа в исходной задаче.
- б) А тут предыдущая конструкция сработает.
- в) Если изначально баланс был нечётным, то тоже сработает предыдущая конструкция. Если чётным, то нужно сначала применить ту же конструкцию, а потом рассмотреть максимальную степень двойки  $2^k$ , на которую делится баланс, и перейти к графу, образованному  $2^k$ -элементными подмножествами вершин полученного графа. Рёбра проводятся между множествами, где из каждой вершины первого множества идёт ребро в какую-то вершину второго множества. Тогда из исходного источника получится нечётное число новых источников.
- г) Нужно провести предыдущую конструкцию для степеней  $p$ .

**11.14.** Нужно отождествить множества и возможные суммы.

## Глава 12

**12.1.** Подойдёт такой протокол: верификатор выбирает случайные  $b \in \{0, 1\}$  и  $r \in \mathbb{Z}^m$  и посылает прuverу  $z = x^b r^2$ . Прувер в ответ посылает  $c \in \{0, 1\}$ . Верификатор принимает доказательство, если  $c = b$ .

**12.2.** Вначале предположим, что верификатор на каждом раунде использует новые случайные биты. В таком случае прuver будет максимизировать вероятность принятия рекурсивно. На шагах, где ходит верификатор, нужно брать математическое ожидание, а на шагах, где ходит прuver — максимум. Поскольку общее число шагов полиномиально, глубина рекурсии, а с ней и затраченная память, тоже полиномиальна. Если верификатор переиспользует старые биты, то рекурсию нужно устроить чуть более аккуратно.

**12.3.** Прувер может зафиксировать случайные биты так, чтобы вероятность принятия была наибольшей.

**12.4.**  $\mathbf{IP} \subset \mathbf{IPP} \subset \mathbf{PSPACE} \subset \mathbf{IP}$ .

**12.5.** Все вычисления верификатора прuver может провести самостоятельно, используя его случайные биты.

**12.7.** Нужно отдельно доказать, что  $\mathbf{MAM} = \mathbf{AM}$ , далее по индукции.

**12.8.**

**12.9.** Достаточно доказать, что в этом случае  $\Sigma_2^p \subset \mathbf{MAM}$ .

**12.10.** Идея состоит в том, чтобы каждые 4 раунда протокола  $\mathbf{AMAM}$  заменить на два раунда  $\mathbf{AM}$ . Проблема состоит в том, что при каждой такой замене число повторов увеличивается в полиномиальное число раз, и общее число повторов станет экспоненциальным. Поэтому нужно добавить ещё одну идею: после ответа Мерлина Артур фиксирует конкретный набор случайных битов, выбранный до этого.

**12.12.** Достаточно применить исходный протокол не к исходному  $S$ , а к подходящей его декартовой степени.

**12.13.** Во-первых, нужно чтобы хеш-функция не выбиралась случайно, а присылалась Мерлином. Во-вторых, нужно выбрать несколько хеш-функций, так чтобы  $\bigcup_i h_i(S)$  покрывало всё пространство для большого  $S$ , но не для маленького. Для этого предварительно может потребоваться увеличить разрыв.

**12.14.** Достаточно доказать, что  $\mathbf{AM} \subset \mathbf{MA}$ . Заметим, что множество  $S = \{(x, r) \mid \exists m V(x, r, m) = 1\}$  лежит в  $\mathbf{NP}$ . По предположению, оно лежит и в  $\mathbf{P/poly}$ . Более того, стандартной конструкцией решения задачи поиска можно построить схему, которая будет по паре  $(x, r) \in S$  выдавать подходящее  $m$ . Значит, Мерлин может изначально прислать такую схему, а Артур будет выбирать случайное  $r$ , получать  $m$  из схемы и проверять, что  $V(x, r, m)$  действительно равно 1. Получится протокол из класса  $\mathbf{MA}$ .

**12.15.** Если  $\mathbf{PSPACE} \subset \mathbf{P/poly}$ , то процедуру вычисления оптимального прuverа можно заменить схемой полиномиального размера. Именно эту схему пошлёт Мерлин в качестве своего сообщения. Затем Артур запустит интерактивный протокол, используя эту схему в качестве прuverа.

**12.16.** Достаточно заменить правую часть в (12.2) на  $k$  и повторить рассуждение.

**12.17.** Используйте предыдущую задачу для доказательства  $\mathbf{P}^{\#\mathbf{P}} \subset \mathbf{IP}$  и сошлитесь на теорему Тоды.

## Глава 13

**13.1.** Нужно запустить  $M^*(x)$  полиномиальное число раз и вернуть первый ответ, отличный от  $\perp$ .

**13.2.** Является: верификатор сам знает, каким должен быть ответ честного прuvera.

**13.3.** Нужно использовать конструкцию из упр. 13.1, а потом вместо  $\perp$  вывести что угодно.

**13.4.** Прuver случайным образом переставляет числа в правильном ответе, шифрует то, что получилось, а затем по просьбе верификатора либо раскрывает случайную строку, либо раскрывает случайный столбец, либо раскрывает случайный выровненный квадрат, либо раскрывает все исходно заданные клетки и сообщает перестановку.

## Глава 14

# Список литературы

- [1] S. Aaronson. “Is P Versus NP Formally Independent?” In: *Bulletin of the EATCS* 81 (2003), pp. 109–136. URL: <http://www.scottaaronson.com/papers/indep.pdf>.
- [2] S. Aaronson. *Why Philosophers Should Care about Computational Complexity*. 2011. URL: <http://www.scottaaronson.com/papers/philos.pdf>.
- [3] S. Aaronson. “ $P \stackrel{?}{=} NP$ ”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 24 (2017), p. 4. URL: <http://www.scottaaronson.com/papers/pnp.pdf>.
- [4] S. Aaronson, G. Kuperberg, C. Granade, and V. Russo. *Complexity Zoo*. URL: <https://complexityzoo.uwaterloo.ca>.
- [5] S. Aaronson and A. Wigderson. “Algebrization: A New Barrier in Complexity Theory”. In: *TOCT* 1.1 (2009). URL: <http://www.scottaaronson.com/papers/alg.pdf>.
- [6] L. M. Adleman. “Two Theorems on Random Polynomial Time”. In: *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. 1978, pp. 75–83. URL: <http://dx.doi.org/10.1109/SFCS.1978.37>.
- [7] L. M. Adleman and M. A. Huang. “Recognizing Primes in Random Polynomial Time”. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. 1987, pp. 462–469. URL: <http://doi.acm.org/10.1145/28395.28445>.
- [8] M. Agrawal, N. Kayal, and N. Saxena. “PRIMES is in P”. In: *Ann. of Math* 160.2 (2002), pp. 781–793.
- [9] W. Aiello, S. Goldwasser, and J. Håstad. “On the power of interaction”. In: *Combinatorica* 10.1 (1990), pp. 3–25. URL: <http://dx.doi.org/10.1007/BF02122692>.
- [10] S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.
- [11] L. Babai. “Trading Group Theory for Randomness”. In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*. 1985, pp. 421–429. URL: <http://doi.acm.org/10.1145/22145.22192>.
- [12] L. Babai. “E-mail and the Unexpected Power of Interaction”. In: (1990), pp. 30–44. URL: <http://dx.doi.org/10.1109/SCT.1990.113952>.
- [13] L. Babai. “Graph Isomorphism in Quasipolynomial Time”. In: *CoRR* abs/1512.03547 (2015). URL: <http://arxiv.org/abs/1512.03547>.

- [14] L. Babai and S. Moran. “Arthur-Merlin Games: A Randomized Proof System, and a Hierarchy of Complexity Classes”. In: *J. Comput. Syst. Sci.* 36.2 (1988), pp. 254–276. URL: [http://dx.doi.org/10.1016/0022-0000\(88\)90028-1](http://dx.doi.org/10.1016/0022-0000(88)90028-1).
- [15] T. P. Baker, J. Gill, and R. Solovay. “Relativizations of the  $P = ?$  NP Question”. In: *SIAM J. Comput.* 4.4 (1975), pp. 431–442. URL: <http://dx.doi.org/10.1137/0204037>.
- [16] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*. Vol. 22. EATCS Monographs on Theoretical Computer Science. Springer, 1990. URL: <http://dx.doi.org/10.1007/978-3-642-75357-2>.
- [17] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1995. URL: <http://dx.doi.org/10.1007/978-3-642-79235-9>.
- [18] R. Beigel, N. Reingold, and D. A. Spielman. “PP Is Closed under Intersection”. In: *J. Comput. Syst. Sci.* 50.2 (1995), pp. 191–202. URL: <http://dx.doi.org/10.1006/jcss.1995.1017>.
- [19] M. Bellare and S. Goldwasser. “The Complexity of Decision Versus Search”. In: *SIAM J. Comput.* 23.1 (1994), pp. 97–119. URL: <http://dx.doi.org/10.1137/S0097539792228289>.
- [20] S. Ben-David and S. Halevi. *On the Independence of P versus NP*. Technical report TR714. Technion, 1992. URL: <http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1992/CS/CS0714.revised.pdf>.
- [21] C. H. Bennett and J. Gill. “Relative to a Random Oracle A,  $P^A \neq NP^A \neq co-NP^A$  with Probability 1”. In: *SIAM J. Comput.* 10.1 (1981), pp. 96–113. URL: <http://dx.doi.org/10.1137/0210008>.
- [22] M. Blum. “A Machine-Independent Theory of the Complexity of Recursive Functions”. B: *J. ACM* 14.2 (amp. 1967), c. 322–336. URL: [http://port70.net/~nsz/articles/classic/blum\\_complexity\\_1976.pdf](http://port70.net/~nsz/articles/classic/blum_complexity_1976.pdf).
- [23] R. B. Boppana, J. Håstad, and S. Zachos. “Does co-NP Have Short Interactive Proofs?” In: *Inf. Process. Lett.* 25.2 (1987), pp. 127–132. URL: [https://www.researchgate.net/publication/223332087\\_Does\\_co-NP\\_have\\_short\\_interactive\\_proofs](https://www.researchgate.net/publication/223332087_Does_co-NP_have_short_interactive_proofs).
- [24] A. Borodin. “Computational Complexity and the Existence of Complexity Gaps”. In: *J. ACM* 19.1 (1972), pp. 158–174. URL: <http://doi.acm.org/10.1145/321679.321691>.
- [25] G. Brassard. “A note on the complexity of cryptography (Corresp.)” In: *IEEE Transactions on information Theory* 25.2 (1979), pp. 232–233.
- [26] J. Buhrsh-Oppenheimer and T. Morioka. “Relativized NP Search Problems and Propositional Proof Systems”. In: *19th Annual IEEE Conference on Computational Complexity (CCC 2004), 21-24 June 2004, Amherst, MA, USA*. 2004, pp. 54–67. URL: <http://dx.doi.org/10.1109/CCC.2004.1313795>.
- [27] K. Cameron. “Thomason’s algorithm for finding a second hamiltonian circuit through a given edge in a cubic graph is exponential on Krawczyk’s graphs”. In: *Discrete Mathematics* 235.1-3 (2001), pp. 69–77. URL: [http://dx.doi.org/10.1016/S0012-365X\(00\)00260-0](http://dx.doi.org/10.1016/S0012-365X(00)00260-0).

- [28] G. Cantor. „Über eine elementare Frage der Mannigfaltigkeitslehre“. In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 1 (1890/1891), S. 75–78.
- [29] R. Chang, B. Chor, O. Goldreich, J. Hartmanis, J. Håstad, D. Ranjan, and P. Rohatgi. “The Random Oracle Hypothesis Is False”. In: *J. Comput. Syst. Sci.* 49.1 (1994), pp. 24–39. URL: <https://www.csee.umbc.edu/~chang/papers/roh/roh.pdf>.
- [30] X. Chen and X. Deng. “3-NASH is PPAD-Complete”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 134 (2005). URL: <http://eccc.hpi-web.de/eccc-reports/2005/TR05-134/index.html>.
- [31] X. Chen and X. Deng. “Settling the Complexity of 2-Player Nash-Equilibrium”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 140 (2005). URL: <http://eccc.hpi-web.de/eccc-reports/2005/TR05-140/index.html>.
- [32] M. Chrobak and S. Poljak. “On common edges in optimal solutions to traveling salesman and other optimization problems”. In: *Discrete Applied Mathematics* 20.2 (1988), pp. 101–111. URL: [http://dx.doi.org/10.1016/0166-218X\(88\)90057-1](http://dx.doi.org/10.1016/0166-218X(88)90057-1).
- [33] A. Cobham. “The intrinsic computational complexity of functions”. In: *Proceedings of 1964 International Congress on Logic, Methodology and Philosophy of Science*. Amsterdam: North-Holland, 1965, pp. 24–30.
- [34] S. Cook. *The P versus NP Problem*. 2000. URL: <http://www.claymath.org/sites/default/files/pvsnp.pdf>.
- [35] S. A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 1971, pp. 151–158. С. А. Кук. «Сложность процедур вывода теорем». В: *Кибернетический сборник. Новая серия. Выпуск 12*. Под ред. О. Б. Лупанов. Москва: Мир, 1975, с. 5–15.
- [36] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. “The Complexity of Computing a Nash Equilibrium”. In: *SIAM J. Comput.* 39.1 (2009), pp. 195–259. URL: <http://dx.doi.org/10.1137/070699652>.
- [37] K. Daskalakis and C. H. Papadimitriou. “Three-Player Games Are Hard”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 139 (2005). URL: <https://people.eecs.berkeley.edu/~christos/papers/3players.pdf>.
- [38] R. A. DeMillo and R. J. Lipton. “The Consistency of “ $P = NP$ ” and Related Problems with Fragments of Number Theory”. In: *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*. 1980, pp. 45–57. URL: <https://www.researchgate.net/publication/221590666>.
- [39] X. Deng, J. R. Edmonds, Z. Feng, Z. Liu, Q. Qi, and Z. Xu. “Understanding PPA-Completeness”. In: *31st Conference on Computational Complexity, CCC 2016, May 29 to June 1, 2016, Tokyo, Japan*. 2016, 23:1–23:25. URL: <http://dx.doi.org/10.4230/LIPIcs.CCC.2016.23>.
- [40] A. W. Dent. “Cryptography in a Hitchhiker’s Universe”. In: *Journal of Craptology* 4 (2007). URL: [http://www.anagram.com/jcrap/Volume\\_4/Woofbark.pdf](http://www.anagram.com/jcrap/Volume_4/Woofbark.pdf).



- [41] D.-Z. Du and K.-I. Ko. *Theory of Computational Complexity, 2nd edition*. Wiley Series in Discrete Mathematics and Optimization. Hoboken, NJ: Wiley, 2014.
- [42] J. Edmonds. “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467. URL: [www.cs.berkeley.edu/~christos/classics/edmonds.ps](http://www.cs.berkeley.edu/~christos/classics/edmonds.ps).
- [43] J. Edmonds. “Euler Complexes (Oiks)”. In: *Electronic Notes in Discrete Mathematics* 36 (2010), pp. 1289–1293. URL: <https://pdfs.semanticscholar.org/6f60/fcf0594cc1b4de1b10ef31653a7a4f9187e5.pdf>.
- [44] S. Even, A. L. Selman, and Y. Yacobi. “The Complexity of Promise Problems with Applications to Public-Key Cryptography”. In: *Information and Control* 61.2 (1984), pp. 159–173. URL: [http://dx.doi.org/10.1016/S0019-9958\(84\)80056-X](http://dx.doi.org/10.1016/S0019-9958(84)80056-X).
- [45] S. Fenner, L. Fortnow, S. A. Kurtz, and L. Li. “An oracle builder’s toolkit”. In: *Information and Computation* 182.2 (2003), pp. 95–136. URL: [https://www.researchgate.net/publication/2897206\\_An\\_Oracle\\_Builder's\\_Toolkit](https://www.researchgate.net/publication/2897206_An_Oracle_Builder's_Toolkit).
- [46] G. S. Fishman. *Monte Carlo: Concepts, Algorithms and Applications*. New York: Springer, 1995.
- [47] L. Fortnow. *The Golden Ticket - P, NP, and the Search for the Impossible*. Princeton University Press, 2013. URL: <http://press.princeton.edu/titles/9937.html>.
- [48] L. Fortnow and S. Homer. “A Short History of Computational Complexity”. In: *Bulletin of the EATCS* 80 (2003), pp. 95–133.
- [49] L. Fortnow and N. Reingold. “PP is Closed Under Truth-Table Reductions”. In: *Inf. Comput.* 124.1 (1996), pp. 1–6. URL: <http://dx.doi.org/10.1006/inco.1996.0001>.
- [50] L. Fortnow and M. Sipser. “Are There Interactive Protocols for CO-NP Languages?” In: *Inf. Process. Lett.* 28.5 (1988), pp. 249–251. URL: [http://dx.doi.org/10.1016/0020-0190\(88\)90199-8](http://dx.doi.org/10.1016/0020-0190(88)90199-8).
- [51] M. Furer, O. Goldreich, Y. Mansour, M. Sipser, and S. Zachos. “On Completeness and Soundness in Interactive Proof Systems”. In: *Advances in Computing Research: a research annual, Vol. 5 (Randomness and Computation)*. 1989, pp. 429–442. URL: <http://www.wisdom.weizmann.ac.il/~oded/PSX/fgmsz.pdf>.
- [52] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. М. Гэри и Д. Джонсон. *Вычислительные машины и труднорешаемые задачи*. Москва: Мир, 1982.
- [53] W. I. Gasarch. “The P=?NP Poll”. In: *ACM SIGACT News* 33.2 (2002), pp. 34–47. URL: <https://www.cs.umd.edu/~gasarch/papers/poll1.pdf>.
- [54] W. I. Gasarch. “Guest Column: the second P =?NP poll”. In: *SIGACT News* 43.2 (2012), pp. 53–77. URL: <https://www.cs.umd.edu/~gasarch/papers/poll2012.pdf>.
- [55] J. Geelen, B. Gerards, and G. Whittle. “Solving Rota’s Conjecture”. In: *Notices of the American Mathematical Society* (2014), pp. 736–743. URL: <http://www.ams.org/notices/201407/rnoti-p736.pdf>.



- [56] J. Gill. “Probabilistic Turing Machines and Complexity of Computations”. PhD thesis. U. C. Berkeley, 1972.
- [57] J. Gill. “Computational Complexity of Probabilistic Turing Machines”. In: *SIAM J. Comput.* 6.4 (1977), pp. 675–695. URL: <http://dx.doi.org/10.1137/0206049>.
- [58] K. Gödel. *A letter to John von Neumann*. 1956. URL: <http://www.cs.cmu.edu/~15251/notes/godel-letter.pdf>.
- [59] P. W. Goldberg. “A Survey of PPAD-Completeness for Computing Nash Equilibria”. In: *CoRR* abs/1103.2709 (2011). URL: <http://arxiv.org/abs/1103.2709>.
- [60] P. W. Goldberg and C. H. Papadimitriou. *Towards a Unified Complexity Theory of Total Functions*. 2016. URL: <http://www.cs.ox.ac.uk/people/paul.goldberg/papers/paper-2.pdf>.
- [61] O. Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.
- [62] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
- [63] O. Goldreich. “On Promise Problems (a survey in memory of Shimon Even [1935-2004])”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 018 (2005). URL: <http://www.wisdom.weizmann.ac.il/~oded/prpr.html>.
- [64] O. Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008.
- [65] O. Goldreich. *P, NP, and NP-Completeness: The Basics of Complexity Theory*. Cambridge University Press, 2010.
- [66] O. Goldreich, S. Micali, and A. Wigderson. “Proofs that Yield Nothing But Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems”. In: *J. ACM* 38.3 (1991), pp. 691–729. URL: <http://doi.acm.org/10.1145/116825.116852>.
- [67] S. Goldwasser, S. Micali, and C. Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In: *SIAM J. Comput.* 18.1 (1989), pp. 186–208. URL: <http://dx.doi.org/10.1137/0218012>.
- [68] S. Goldwasser and M. Sipser. “Private Coins versus Public Coins in Interactive Proof Systems”. In: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*. 1986, pp. 59–68. URL: <http://doi.acm.org/10.1145/12130.12137>.
- [69] A. Hall. “On an Experimental Determination of  $\pi$ ”. In: *Messenger of Mathematics* 2 (1872), pp. 113–114. URL: <http://cerebro.xu.edu/math/Sources/Buffon/Hall%20on%20Buffon%20Needle.pdf>.
- [70] J. Hartmanis. “Gödel, von Neumann and the P=?NP Problem”. In: *Bulletin of the EATCS* 38 (1989), pp. 101–106. URL: <https://ecommons.cornell.edu/bitstream/handle/1813/6910/89-994.pdf>.
- [71] J. Hartmanis and R. E. Stearns. “On the computational complexity of algorithms”. In: *Transactions of the American Mathematical Society* 117 (1965), pp. 285–306.

- [72] L. A. Hemaspaandra and M. Ogihara. *The Complexity Theory Companion*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2002. URL: <http://dx.doi.org/10.1007/978-3-662-04880-1>.
- [73] M. J. H. Heule, O. Kullmann, and V. W. Marek. “Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 2016, pp. 228–245. URL: [https://doi.org/10.1007/978-3-319-40970-2\\_15](https://doi.org/10.1007/978-3-319-40970-2_15).
- [74] S. Homer and A. L. Selman. *Computability and Complexity Theory, Second Edition*. Texts in Computer Science. Springer, 2011. URL: <http://dx.doi.org/10.1007/978-1-4614-0682-2>.
- [75] J. Hopcroft and R. Tarjan. “Efficient Planarity Testing”. In: *J. ACM* 21.4 (Oct. 1974), pp. 549–568. URL: <http://doi.acm.org/10.1145/321850.321852>.
- [76] N. Immerman. “Nondeterministic Space is Closed Under Complementation”. In: *SIAM J. Comput.* 17.5 (1988), pp. 935–938. URL: <http://dx.doi.org/10.1137/0217058>.
- [77] T. R. Jensen. “Simple algorithm for finding a second Hamilton cycle”. In: *Siberian Electronic Mathematical Reports* 9 (2012), pp. 151–155. URL: <http://semr.math.nsc.ru/v9/p151-155.pdf>.
- [78] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. “How Easy is Local Search?” In: *J. Comput. Syst. Sci.* 37.1 (1988), pp. 79–100. URL: [http://dx.doi.org/10.1016/0022-0000\(88\)90046-3](http://dx.doi.org/10.1016/0022-0000(88)90046-3).
- [79] V. Kabanets and R. Impagliazzo. “Derandomizing Polynomial Identity Tests Means Proving Circuit Lower Bounds”. In: *Computational Complexity* 13.1-2 (2004), pp. 1–46. URL: <http://dx.doi.org/10.1007/s00037-004-0182-6>.
- [80] R. M. Karp. “Reducibility Among Combinatorial Problems”. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York*. 1972, pp. 85–103. URL: <http://www.cs.berkeley.edu/~luca/cs172/karp.pdf>. Р. М. Карп. «Сводимость комбинаторных проблем». В: *Кибернетический сборник. Новая серия. Выпуск 12*. Под ред. О. Б. Лупанов. Москва: Мир, 1975, с. 16–38.
- [81] R. M. Karp and R. J. Lipton. “Some Connections between Nonuniform and Uniform Complexity Classes”. In: *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*. 1980, pp. 302–309. URL: <http://doi.acm.org/10.1145/800141.804678>.
- [82] S. Kintali. *A Compendium of PPAD-complete problems*. URL: <http://www.cs.princeton.edu/~kintali/ppad.html>.
- [83] D. E. Knuth. “Big Omicron and Big Omega and Big Theta”. In: *SIGACT News* 8.2 (Apr. 1976), pp. 18–24. URL: <http://doi.acm.org/10.1145/1008328.1008329>.
- [84] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Progress in Theoretical Computer Science. Springer, 1993.

- [85] W. Kocay. *The Hopcroft–Tarjan planarity algorithm*. 1993. URL: [www.combinatorialmath.ca/G&G/articles/planarity.pdf](http://www.combinatorialmath.ca/G&G/articles/planarity.pdf).
- [86] B. Konev and A. Lisitsa. “Computer-aided proof of Erdős discrepancy properties”. In: *Artif. Intell.* 224 (2015), pp. 103–118. URL: <https://doi.org/10.1016/j.artint.2015.03.004>.
- [87] A. Krawczyk. “The Complexity of Finding a Second Hamiltonian Cycle in Cubic Graphs”. In: *J. Comput. Syst. Sci.* 58.3 (1999), pp. 641–647. URL: <http://dx.doi.org/10.1006/jcss.1998.1611>.
- [88] R. E. Ladner. “On the Structure of Polynomial Time Reducibility”. In: *J. ACM* 22.1 (1975), pp. 155–171. URL: <http://doi.acm.org/10.1145/321864.321877>.
- [89] C. Lautemann. “BPP and the Polynomial Hierarchy”. In: *Inf. Process. Lett.* 17.4 (1983), pp. 215–217. URL: [http://dx.doi.org/10.1016/0020-0190\(83\)90044-3](http://dx.doi.org/10.1016/0020-0190(83)90044-3).
- [90] K. de Leeuw, E. Moore, C. Shannon, and N. Shapiro. “Computability by Probabilistic Machines”. In: *Automata Studies*. Ed. by C. Shannon and J. McCarthy. Princeton University Press, 1956, pp. 183–212.
- [91] M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications, Third Edition*. Texts in Computer Science. Springer, 2008. URL: <http://dx.doi.org/10.1007/978-0-387-49820-1>.
- [92] R. J. Lipton. *The P=NP Question and Gödel’s Lost Letter*. Springer, 2010. URL: <http://www.springer.com/gp/book/9781441971548>.
- [93] R. J. Lipton and K. W. Regan. *People, Problems, and Proofs - Essays from Gödel’s Lost Letter: 2010*. Springer, 2013. URL: <http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-41421-3>.
- [94] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. “Algebraic Methods for Interactive Proof Systems”. In: *J. ACM* 39.4 (1992), pp. 859–868. URL: <http://doi.acm.org/10.1145/146585.146605>.
- [95] N. Megiddo and C. H. Papadimitriou. “On Total Functions, Existence Theorems and Computational Complexity”. In: *Theor. Comput. Sci.* 81.2 (1991), pp. 317–324. URL: [http://dx.doi.org/10.1016/0304-3975\(91\)90200-L](http://dx.doi.org/10.1016/0304-3975(91)90200-L).
- [96] N. Metropolis and S. M. Ulam. “The Monte Carlo Method”. In: *Journal of the American Statistical Association* 44.247 (Sept. 1949), pp. 335–341. URL: [http://homepages.rpi.edu/~angel/MULTISCALE/metropolis\\_Ulam\\_1949.pdf](http://homepages.rpi.edu/~angel/MULTISCALE/metropolis_Ulam_1949.pdf).
- [97] G. L. Miller. “Riemann’s Hypothesis and Tests for Primality”. In: *J. Comput. Syst. Sci.* 13.3 (1976), pp. 300–317. URL: [http://dx.doi.org/10.1016/S0022-0000\(76\)80043-8](http://dx.doi.org/10.1016/S0022-0000(76)80043-8).
- [98] C. Moore and S. Mertens. *The Nature of Computation*. Oxford University Press, 2011. URL: <http://nature-of-computation.org>.
- [99] T. Morioka. “Classification of Search Problems and Their Definability in Bounded Arithmetic”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 082 (2001). URL: <http://eccc.hpi-web.de/eccc-reports/2001/TR01-082/index.html>.

- [100] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [101] J. F. Nash. *Letters to the United States National Security Agency*. 1955. URL: [https://www.nsa.gov/news-features/declassified-documents/nash-letters/assets/files/nash\\_letters1.pdf](https://www.nsa.gov/news-features/declassified-documents/nash-letters/assets/files/nash_letters1.pdf).
- [102] C. H. Papadimitriou. “Games Against Nature”. In: *J. Comput. Syst. Sci.* 31.2 (1985), pp. 288–301. URL: <http://www.sciencedirect.com/science/article/pii/0022000085900455>.
- [103] C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
- [104] C. H. Papadimitriou. “On the Complexity of the Parity Argument and Other Inefficient Proofs of Existence”. In: *J. Comput. Syst. Sci.* 48.3 (1994), pp. 498–532. URL: <https://people.eecs.berkeley.edu/~christos/papers/On%5C%20the%20Complexity.pdf>.
- [105] C. H. Papadimitriou, O. Goldreich, A. Wigderson, A. A. Razborov, and M. Sipser. “The future of computational complexity theory: part I”. In: *ACM SIGACT News* 27.3 (1996), pp. 6–12. URL: [http://www.math.ias.edu/~avi/PUBLICATIONS/The\\_Future\\_of\\_Computational\\_Complexity\\_Theory\\_I.pdf](http://www.math.ias.edu/~avi/PUBLICATIONS/The_Future_of_Computational_Complexity_Theory_I.pdf).
- [106] H. C. Pocklington. “The direct solution of the quadratic and cubic binomial congruences with prime moduli”. In: *Proceedings of the Cambridge Philosophical Society* 19 (1917), pp. 57–59.
- [107] J. Quisquater et al. “How to Explain Zero-Knowledge Protocols to Your Children”. In: *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*. 1989, pp. 628–631. URL: [http://dx.doi.org/10.1007/0-387-34805-0\\_60](http://dx.doi.org/10.1007/0-387-34805-0_60).
- [108] M. O. Rabin. “Probabilistic algorithm for testing primality”. In: *Journal of number theory* 12.1 (1980), pp. 128–138.
- [109] A. A. Razborov and S. Rudich. “Natural Proofs”. In: *J. Comput. Syst. Sci.* 55.1 (1997), pp. 24–35. URL: <http://dx.doi.org/10.1006/jcss.1997.1494>.
- [110] O. Reingold. “Undirected connectivity in log-space”. In: *J. ACM* 55.4 (2008). URL: <http://doi.acm.org/10.1145/1391289.1391291>.
- [111] B. Reus. *Limits of Computation - From a Programming Perspective*. Undergraduate Topics in Computer Science. Springer, 2016. URL: <http://dx.doi.org/10.1007/978-3-319-27889-6>.
- [112] N. Robertson and P. D. Seymour. “Graph minors. I. Excluding a forest”. In: *J. Comb. Theory, Ser. B* 35.1 (1983), pp. 39–61. URL: [http://dx.doi.org/10.1016/0095-8956\(83\)90079-5](http://dx.doi.org/10.1016/0095-8956(83)90079-5).
- [113] N. Robertson and P. D. Seymour. “Graph Minors. XIII. The Disjoint Paths Problem”. In: *J. Comb. Theory, Ser. B* 63.1 (1995), pp. 65–110. URL: <http://dx.doi.org/10.1006/jctb.1995.1006>.
- [114] N. Robertson and P. D. Seymour. “Graph Minors. XX. Wagner’s conjecture”. In: *J. Comb. Theory, Ser. B* 92.2 (2004), pp. 325–357. URL: <http://dx.doi.org/10.1016/j.jctb.2004.08.001>.

- [115] N. Robertson and P. D. Seymour. “Graph Minors. XXII. Irrelevant vertices in linkage problems”. In: *J. Comb. Theory, Ser. B* 102.2 (2012), pp. 530–563. URL: <http://dx.doi.org/10.1016/j.jctb.2007.12.007>.
- [116] W. J. Savitch. “Relationships Between Nondeterministic and Deterministic Tape Complexities”. In: *J. Comput. Syst. Sci.* 4.2 (1970), pp. 177–192. URL: [http://dx.doi.org/10.1016/S0022-0000\(70\)80006-X](http://dx.doi.org/10.1016/S0022-0000(70)80006-X).
- [117] N. Saxena. “Progress on Polynomial Identity Testing”. In: *Bulletin of the EATCS* 99 (2009), pp. 49–79. URL: <http://www.cse.iitk.ac.in/users/nitin/papers/pit-survey09.pdf>.
- [118] N. Saxena. “Progress on Polynomial Identity Testing - II”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 20 (2013), p. 186. URL: <http://www.cse.iitk.ac.in/users/nitin/papers/pit-survey13.pdf>.
- [119] M. Schaefer and C. Umans. “Completeness in the Polynomial-Time Hierarchy: A Compendium”. In: *ACM SIGACT News* 33.3 (), pp. 32–49. URL: <https://www.researchgate.net/publication/245726393>.
- [120] U. Schöning and R. Pruim. *Gems of Theoretical Computer Science*. Springer-Verlag, 1998.
- [121] J. Shallit. “Randomized Algorithms in “Primitive” Cultures or What is the Oracle Complexity of a Dead Chicken?” In: *SIGACT News* 23.4 (Oct. 1992), pp. 77–80. URL: <https://cs.uwaterloo.ca/~shallit/Papers/ch.pdf>.
- [122] A. Shamir. “IP=PSPACE”. In: *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume I*. 1990, pp. 11–15. URL: <http://dx.doi.org/10.1109/FSCS.1990.89519>.
- [123] C. E. Shannon. “The synthesis of two-terminal switching circuits”. In: *Bell Systems Technical Journal* 28.1 (1949), pp. 59–98.
- [124] A. Shen. “IP=PSPACE: simplified proof”. In: *Journal of the ACM* 39.4 (1992). URL: <http://www.lirmm.fr/~ashen/mathtext/ip/1992/ip-pspace-simplified.pdf>.
- [125] M. Sipser. “A Complexity Theoretic Approach to Randomness”. In: *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*. 1983, pp. 330–335. URL: <http://doi.acm.org/10.1145/800061.808762>.
- [126] M. Sipser. “The History and Status of the P versus NP Question”. In: *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*. 1992, pp. 603–618. URL: <http://doi.acm.org/10.1145/129712.129771>.
- [127] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.
- [128] C. Smoryński. “The varieties of arboreal experience”. In: *The Mathematical Intelligencer* 4.4 (1982), pp. 182–189.
- [129] R. Solovay and V. Strassen. “A Fast Monte-Carlo Test for Primality”. In: *SIAM J. Comput.* 6.1 (1977), pp. 84–85. URL: <http://dx.doi.org/10.1137/0206006>.
- [130] L. J. Stockmeyer. “The Polynomial-Time Hierarchy”. In: *Theoretical Computer Science* 3.1 (1976), pp. 1–22. URL: <http://www.sciencedirect.com/science/article/pii/030439757690061X>.



- [131] L. J. Stockmeyer and A. R. Meyer. “Word Problems Requiring Exponential Time: Preliminary Report”. In: *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*. 1973, pp. 1–9. URL: <http://doi.acm.org/10.1145/800125.804029>.
- [132] R. Szelepcsényi. “The Method of Forced Enumeration for Nondeterministic Automata”. In: *Acta Inf.* 26.3 (1988), pp. 279–284. URL: <http://dx.doi.org/10.1007/BF00299636>.
- [133] T. Tao. “The Erdős discrepancy conjecture”. In: *Discrete Analysis* 1 (2016). URL: <https://arxiv.org/abs/1509.05363>.
- [134] A. G. Thomason. “Hamilton cycles and uniquely edge-colourable graphs”. In: *Ann. of Discrete Math.* 3 (1978), pp. 259–268.
- [135] S. Toda. “PP is as Hard as the Polynomial-Time Hierarchy”. In: *SIAM J. Comput.* 20.5 (1991), pp. 865–877. URL: <http://dx.doi.org/10.1137/0220053>.
- [136] W. T. Tutte. “On Hamilton circuits”. In: *J. London Math. Soc.* 21 (1946), pp. 98–101.
- [137] S. P. Vadhan. “Pseudorandomness”. In: *Foundations and Trends in Theoretical Computer Science* 7.1-3 (2012), pp. 1–336. URL: <http://people.seas.harvard.edu/~salil/pseudorandomness/>.
- [138] L. G. Valiant. “The Complexity of Computing the Permanent”. In: *Theor. Comput. Sci.* 8 (1979), pp. 189–201. URL: [http://dx.doi.org/10.1016/0304-3975\(79\)90044-6](http://dx.doi.org/10.1016/0304-3975(79)90044-6).
- [139] L. G. Valiant and V. V. Vazirani. “NP is as Easy as Detecting Unique Solutions”. In: *Theor. Comput. Sci.* 47.3 (1986), pp. 85–93. URL: [http://dx.doi.org/10.1016/0304-3975\(86\)90135-0](http://dx.doi.org/10.1016/0304-3975(86)90135-0).
- [140] K. W. Wagner. “The Complexity of Combinatorial Problems with Succinct Input Representation”. In: *Acta Inf.* 23.3 (1986), pp. 325–356. URL: <http://dx.doi.org/10.1007/BF00289117>.
- [141] Wikipedia contributors. *List of long mathematical proofs*. URL: [https://en.wikipedia.org/wiki/List\\_of\\_long\\_mathematical\\_proofs](https://en.wikipedia.org/wiki/List_of_long_mathematical_proofs).
- [142] H. C. Williams and J. O. Shallit. “Factoring Integers Before Computers”. In: *Mathematics of Computation 1943–1993: A Half-Century of Computational Mathematics. Mathematics of Computation 50th Anniversary Symposium, August 9–13, 1993, Vancouver, British Columbia*. Ed. by W. Gautschi. Vol. 48. 1993, pp. 481–531.
- [143] D. Zeps. *Forbidden Minors for Projective Plane are Free-Toroidal or Non-Toroidal*. URL: <http://www.ltn.lv/~dainize/MathPages/ForbMinProjPl.pdf>.
- [144] M. Zimand. *Computational Complexity: A Quantitative Perspective*. Elsevier, 2004.
- [145] С. А. Абрамов. *Лекции о сложности алгоритмов*. Москва: МЦНМО, 2012.
- [146] В. Босс. *Лекции по математике. Т. 10: Перебор и эффективные алгоритмы*. Москва: Издательство ЛКИ, 2008.

- [147] Н. К. Верещагин, В. А. Успенский и А. Шень. *Колмогоровская сложность и алгоритмическая случайность*. Москва: МЦНМО, 2013. URL: <http://www.mccme.ru/free-books/shen/kolmbook.pdf>.
- [148] Н. Верещагин. *Лекции по математической криптографии*.
- [149] Э. А. Гирш. *Структурная сложность I: конспект лекций*. URL: <http://logic.pdmi.ras.ru/~hirsch/students/complexity1/>.
- [150] М. Гэри и Д. Джонсон. *Вычислительные машины и труднорешаемые задачи*. Москва: Мир, 1982.
- [151] В. А. Емеличев, О. И. Мельников, В. И. Сарванов и Р. И. Тышкевич. *Лекции по теории графов*. Москва: Наука, 1990.
- [152] Г. Кантор. «Об одном элементарном вопросе учения о многообразиях». В: *Труды по теории множеств*. Москва: Наука, 1985, с. 170—172.
- [153] Р. М. Карп. «Сводимость комбинаторных проблем». В: *Кибернетический сборник. Новая серия. Выпуск 12*. Под ред. О. Б. Лупанов. Москва: Мир, 1975, с. 16—38.
- [154] А. Китаев, А. Шень и М. Вялый. *Классические и квантовые вычисления*. Москва: МЦНМО, ЧеРо, 1999. URL: <http://www.mccme.ru/free-books/qcomp/qps00205.zip>.
- [155] Д. Кнут. *Искусство программирования. Том 2: получисленные алгоритмы*. Москва: Вильямс, 2017.
- [156] А. Коняев. *Утверждение о запрещённом миноре*. 2013. URL: <https://lenta.ru/articles/2013/08/26/matroid/>.
- [157] Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест и К. Штайн. *Алгоритмы: построение и анализ*. Москва: Вильямс, 2016.
- [158] В. Королёв. *Доказательство теоремы о тройках заняло 200 терабайт*. 2016. URL: <https://nplus1.ru/news/2016/05/27/math>.
- [159] В. Н. Крупский. *Введение в сложность вычислений*. Москва: Факториал, 2006.
- [160] Н. Н. Кузюрин и С. А. Фомин. *Эффективные алгоритмы и сложность вычислений*. Москва: МФТИ, 2007. URL: <http://discopal.ispras.ru/ru.book-advanced-algorithms.htm>.
- [161] С. А. Кук. «Сложность процедур вывода теорем». В: *Кибернетический сборник. Новая серия. Выпуск 12*. Под ред. О. Б. Лупанов. Москва: Мир, 1975, с. 5—15.
- [162] Л. А. Левин. «Универсальные задачи перебора». В: *Проблемы передачи информации* 9.3 (1973), с. 115—116. URL: <http://mi.mathnet.ru/ppi914>.
- [163] О. Б. Лупанов. «О синтезе контактных схем». В: *Доклады Академии Наук СССР* 119.1 (1958), с. 23—26.
- [164] С. Немалевич. *Всё, расходимся*. 2015. URL: <https://nplus1.ru/material/2015/10/30/polymath>.
- [165] А. А. Разборов. *Алгебраическая сложность*. Москва: МЦНМО, 2016.

- [166] А. С. Станкевич и Студенты НИУ ИТМО. *Вики-конспекты: теория сложности*. URL: [http://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F\\_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8](http://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8).
- [167] К. Шеннон. «Синтез двухполюсных переключательных схем». В: *Работы по теории информации и кибернетике*. Москва: Изд-во иностр. лит-ры, 1963, с. 59—105.