САНКТ-ПЕТЕРБУРГСКОЕ ОТДЕЛЕНИЕ
МАТЕМАТИЧЕСКОГО ИНСТИТУТА ИМ. В.А.СТЕКЛОВА
РОССИЙСКОЙ АКАДЕМИИ НАУК

На правах рукописи

Лифшиц Юрий Михайлович

Алгоритмы и анализ трудоемкости обработки сжатых текстов

05.13.17 – Теоретические основы информатики

Диссертация
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
член-корр. РАН, д.ф.-м.н. Ю.В. Матиясевич

Санкт-Петербург
2007

Я посвящаю эту работу
своим замечательным родителям —
Михаилу Анатольевичу Лифшицу
и Надежде Алексеевне Комаровой.

Благодарности

Я благодарю своего научного руководителя Юрия Владимировича Матиясевича за постоянное внимание, поддержку и конструктивную критику моих исследований. Мои соавторы Ирен Гессарян, Юхани Карьюмяки, Маркус Лори, Патрик Сигельски знакомили меня с интересными открытыми проблемами и были самыми первыми критиками полученных мной результатов. Многочисленные читатели черновых версий этой работы Михаил Брудно, Николай Гравин, Михаил Дворкин, Вадим Джанмухамедов, Алексей Ильин, Шунсуке Иненага, Арист Кожевников, Александр Кузнецов, Денис Кулагин, Иван Лагунов, Свен Лаур, Илья Миронов, Александр Охотин, Надежда Поликарпова, Юрий Притыкин, Матье Раффино, Войцех Риттер, Анатолий Федуков, Алексей Хворост, Арсений Шур прислали много важных замечаний и комментариев. Наконец, весь коллектив лаборатории математической логики ПОМИ РАН создавал замечательную научную атмосферу, в которой было легко и приятно работать.

Исследования, лежащие в основе диссертации, были поддержаны грантами INTAS № 04-77-7173, INTAS YSF 1000014-6233, грантом РФФИ

№ 06-01-00584-а, контрактами Федерального агентства по науке и инновациям № 02.442.11.7290 и № 02.442.11.7291, президентскими программами “Ведущие научные школы” НШ-8464.2006.1 и НШ-2203.2003.1, а также специальными программами компаний Intel и Яндекс.

Оглавление

Глава 1

Введение

Как хранить данные, чтобы одновременно иметь быстрый доступ к нужной информации и максимально уменьшить их размер? Для решения первой задачи разработано множество структур данных (сбалансированные деревья, суффиксные массивы и т.д.), ориентированных на быструю обработку конкретных видов запросов. Для уменьшения размера данных используются алгоритмы архивирования (LZ77, LZW, кодирование длин серий и т.д.). Можно ли объединить достоинства этих двух подходов? В представляемой работе построены алгоритмы вычисления ряда свойств сжатых текстов без распаковки, получены нижние оценки на трудоемкость некоторых других задач на сжатых текстах, и, наконец, предложен новый подход к архивированию текстов.

1.1 Алгоритмы на сжатых текстах

1.1.1 Постановка решаемых задач

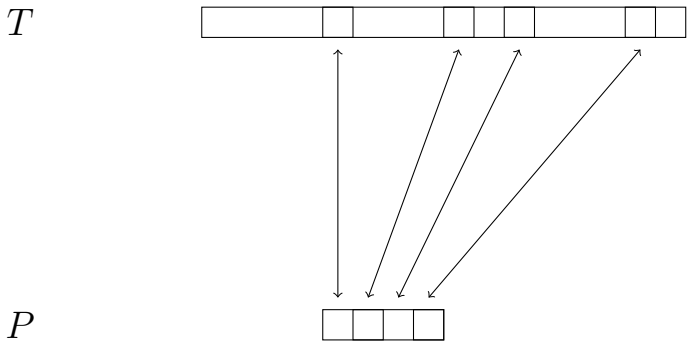
Исследования начались с построения алгоритмов для поиска явно заданных подстрок в сжатых текстах [8, 18]. Очень быстро от рассмотрения конкретных алгоритмов архивирования исследователи перешли к общей модели сжатого текста — *прямолинейной программе* (ПП). Неформально, ПП является грамматикой, которая порождает ровно один текст. Оказалось, что тексты, сжатые практическими архиваторами, быстро и без значительного увеличения размера могут быть переведены в грамматику, описывающую тот же текст [42]. Мы подробно опишем прямолинейные программы в главе 2. Недавно было предложено некоторое обобщение прямолинейных программ под названием collage systems [29].

Мы будем строить алгоритмы для следующих трех задач:

1. Даны два сжатых текста, представленные в виде прямолинейных программ. Определить, совпадают ли исходные тексты.

2. Даны два сжатых текста. Определить, входит ли первый из них во второй. При положительном ответе найти место первого вхождения и общее число вхождений.

3. Дан шаблон (явно заданный) и сжатый текст. Определить, образуют ли буквы шаблона подпоследовательность в тексте. Другими словами, можно ли вычеркнуть часть букв в тексте так, чтобы остался шаблон?



Вхождение шаблона P в текст T в виде подпоследовательности

Первая задача является частным случаем второй. Равенство сжатых текстов может использоваться для сверки различных архивных копий одной и той же системы, поиск подстрок — для верификации программ и поиска по медиа-данным (в них шаблон также может быть большим и требовать архивирования).

Задача о равенстве впервые была решена в работе [39] в 1994 году. Была доказана оценка $O(n^4)$ на время работы этого алгоритма, где n равно сумме размеров прямолинейных программ, порождающих два сравниваемых текста. Алгоритм для поиска сжатой подстроки в сжатом тексте впервые появился в 1995 году в статье [27]. Вслед за этим удалось расширить алгоритм для вычисления различных комбинаторных свойств текста [20]. Далее, в 1997 году Миязаки, Шинохара и Такеда [35] построили алгоритм, требующий $O(n^2m^2)$ времени для поиска подстрок, где m и n — размеры сжатого шаблона и сжатого текста, соответственно. Наконец, в 2000 году удалось провести поиск подстрок за $O(nm)$ для еще одного специального случая [24].

Поиск подпоследовательностей является естественным “родственником” поиска подстрок. Тем не менее, до сих пор не было предложено ни одного алгоритма поиска подпоследовательностей напрямую в сжатом

тексте.

1.1.2 Результаты

В разделе 3.1 этой работы нам удалось улучшить алгоритмы [24, 35, 27, 39, 20] и решить задачу о равенстве за $O(n^3)$, а задачу о поиске подстрок за $O(n^2m)$ шагов.

Алгоритмическая техника, которую мы применили для задачи о поиске подстроки, позволяет вычислять и комбинаторные свойства сжатых текстов, например, длины минимального периода и накрытия. Это приложение и ряд открытых вопросов сформулированы в разделе 3.4. В частности, одной из самых заманчивых задач для будущих исследований является построение *сжатого суффиксного дерева*.

Кроме того, мы построили алгоритм, который определяет вложимость шаблона в текст. Наш алгоритм также выдает количество минимальных подстрок и количество подстрок определенной длины, в которые вкладывается шаблон. Трудоемкость алгоритма равна $mk^2 \log k$, где k — длина шаблона, а m — размер прямолинейной программы, порождающей текст. Таким образом, время работы нашего алгоритма линейно относительно размера сжатого текста.

1.2 Нижние оценки трудоемкости обработки сжатых текстов

Чтобы показать все преимущества хранения текстовой информации в виде прямолинейных программ, мы ставим следующую общую задачу. Для каждого классического свойства текста мы ищем алгоритм, кото-

рый (1) на вход получает прямолинейную программу, (2) на выходе сообщает, обладает ли порождаемый этой программой текст данным свойством, (3) время исполнения алгоритма полиномиально ограничено относительно размера ПП.

В процессе исследований мы столкнулись с двумя задачами, для которых построить такой алгоритм никак не удавалось:

1. Дано два сжатых текста одинаковой длины. Определить количество несовпадающих символов (расстояние Хэмминга) между ними.

2. Дано два сжатых текста. Определить, является ли один текст подпоследовательностью второго.

Вычисление расстояния Хэмминга является самой простой формой приближенного поиска подстрок, который полезен для анализа как биологических данных, так и медиа-файлов. Кроме того, вычисление расстояния Хэмминга между сжатыми текстами является естественным продолжением задачи о равенстве сжатых текстов, для которой мы уже построили эффективный алгоритм.

Как было уже отмечено выше, для поиска в сжатых текстах существуют полиномиальные алгоритмы, которые находят (1) явно заданные подстроки, (2) подстроки, представленные в сжатом виде (3) вхождения в виде подпоследовательности явно заданного шаблона. Вычислительная трудность четвертой задачи — проверки вложимости "как подпоследовательность" одного *сжатого* текста в другой сжатый текст, оставалась совершенно неясной. Определение вложимости сжатого шаблона в сжатый текст могло оказаться как решаемым за полиномиальное время, так и PSPACE-полной задачей.

В представляемой работе удалось разработать специальную технику оценки трудности вышеприведенных задач на сжатых текстах. Мы научились *сводить* решение общепризнанно-трудных задач к вычислению определенных свойств сжатых текстов.

Так, в главе 4 мы докажем, что вычисление расстояния Хэмминга для сжатых текстов является #P-полной задачей. Напомним, что #P — это класс функций, своеобразное расширение класса предикатов NP.

Мы также показали, что задача определения вложимости сжатого шаблона в сжатый текст не только NP-трудна, но и лежит вне класса NP(при предположении NP≠coNP). Последний результат стал большой неожиданностью, так как задача по своей природе очень напоминает представителей класса NP.

1.3 Новый подход к архивированию: разреженная периодичность

В первых двух частях нашего исследования мы изучали архивы, которые легко могут быть преобразованы в прямолинейную программу, порождающую тот же текст. Базовой операцией восстановления текста из такого описания является конкатенация двух уже определенных фрагментов. Оставался открытым вопрос о том, есть ли такие системы представления текстов, которые используют более широкий класс операций. Мы представим новое понятие — *разреженную периодичность*, — которое может быть использовано для компактного описания некоторых длинных текстов.

1.3.1 Основное понятие

Слово S называется *чисто периодическим*, если $S = W^k = W \dots W$. Другими словами, чистая периодичность соответствует делимости $p|n$ и равенству $s_i = s_{i+p}$ для всех $1 \leq i < i + p \leq n$. В этой работе мы будем иметь дело только с чистой периодичностью.

Два слова на иллюстрации внизу не являются чисто периодическими. Однако, они обладают некоторой структурной закономерностью. В пятой главе диссертации мы вводим новое обобщенное понятие периодичности, в соответствии с которым эти строки становятся периодическими.

Неформально, слово T называется *разреженно периодическим*, если существует такое частично определенное слово S, что T может быть разделено на несколько параллельных копий S. Мы представим формальное определение со всеми подробностями в первых же абзацах главы 5. На следующей иллюстрации показаны разбиения, доказывающие разреженную периодичность исходных примеров.

1.3.2 Рассматриваемые вопросы

Глава 5 посвящена исследованию основных свойств разреженной периодичности и сопоставлению их с классическим определением. Нас интересуют следующие вопросы: (1) Как перечислить все разреженные пе-

риоды? (2) Каждое ли слово имеет единственный *примитивный* разреженный период? (3) Как найти все разреженные периоды минимального размера? (4) Сколько разреженных периодов может иметь слово длины n? (5) Каково отношение между примитивным классическим периодом и примитивным разреженным периодом?

У нас есть несколько причин изучать разреженные периоды. Во-первых, это естественное обобщение классического понятия, так как любой классический период является также и разреженным. Обратное необязательно верно. Во-вторых, в случае, когда разреженный период относительно мал, мы можем полностью описать длинную строку, сообщив только ее длину и разреженный период. Таким образом, мы получаем новый формализм для компактного описания слов из некоторый класса. И, таким образом, все слова в этом классе имеют малую колмогоровскую сложность.. Разреженная периодичность может быть полезна для построения новых алгоритмов архивирования, особенно для обобщения *кодирования длин серий* (RLE). Заметим здесь, что отношение между "размером" разреженного периода и длиной классического периода может быть сколь угодно мало. В-третьих, понятие разреженной периодичности дает геометрический взгляд на структуру текстов. Далее, в разделе 5.3 мы формулируем гипотезу, утверждающую что разреженная периодичность невыразима при помощи уравнений в словах. Еще одним стимулом для изучения разреженной периодичности является надежда обнаружить новые закономерности в реальных данных. Представляется интересным провести экспериментальные исследования в этом направлении.

Покажем один естественный источник разреженной периодичности.

Рассмотрим многомерный параллелепипед $n_1 \times \cdots \times n_d$. Пусть каждая его клетка покрашена в некоторый цвет. Отсортируем все клетки в алфавитном порядке их координат (от $(0, \ldots, 0)$ до $(n_1 - 1, \ldots, n_d - 1)$) и выпишем все их цвета в единую последовательность. Предположим теперь, что исходный параллелепипед был замощен меньшим $m_1 \times \cdots \times m_d$, причем $m_1 | n_1, \ldots, m_d | n_d$ и каждая копия маленького параллелепипеда была раскрашена одинаково. Тогда последовательность цветов большого параллелепипеда будет обладать разреженной периодичностью. Мы вернемся к этому примеру после введения ряда необходимых терминов. Другими источниками разреженной периодичности могут стать XML-файлы и автоматически порожденные тексты.

1.3.3 Результаты

В главе 5 мы ввели понятие разреженной периодичности. Оно выглядит очень просто и естественно, но, насколько нам известно, никогда не было явно сформулировано прежде.

Далее мы определили частичный порядок на разреженных периодах и обнаружили, что, в отличие от классического случая, у текста может оказаться несколько примитивных разреженных периодов. Наименьший из известных нам примеров (24 буквы), обладающих этим свойством, мы представим в разделе 5.1. Этот пример позволил опровергнуть гипотезу Теро Харью об *общем подразбиении*.

Тем не менее, в подразделе 5.2.2 мы докажем, что каждый примитивный разреженный период слова T также является разреженным периодом классического примитивного периода T. Из этого свойства следует, что разреженная периодичность живет “внутри” классического прими-

тивного периода. Доказательство основано на расширенном алгоритме Евклида.

В подразделе 5.2.1 мы представим классификацию всех возможных разреженных периодов. Наиболее трудным техническим шагом является доказательство полноты нашей конструкции, то есть того факта, что наша классификация содержит *все* разреженные периоды. Мы получили рекурсивную формулу для вычисления функции $L(n)$, которая дает максимальное количество разреженных периодов для слов длины n. Значение $L(n)$ может оказаться даже больше длины исходного текста. Например, слово длиной 36 символов может иметь до $L(36) = 52$ разреженных периодов.

Наконец, в подразделе 5.2.3 мы представим алгоритм для нахождения (всех) разреженных периодов минимального размера для данного текста. Трудоемкость алгоритма составляет $n^{1+o(1)}$ шагов.

1.4 Обзор литературы

1.4.1 Исследования по алгоритмам обработки сжатых текстов

Для прямолинейных программ, помимо поиска подстрок, были представлены алгоритмы для поиска регулярных выражений и приближенного поиска подстрок (соответственно, [36] и [26]).

Неожиданно оказалось, что алгоритмы на сжатых текстах имеют прямое отношение ко многим другим задачам теоретической информатики. Так, с их помощью впервые удалось построить полиномиальный

по памяти алгоритм решения уравнений в словах [40]; более быстрый алгоритм проверки эквивалентности программ (естественно, в ограниченном классе) [31]; полиномиальный алгоритм верификации диаграмм сообщений [21].

Быстрый поиск по сжатым данным имеет и прикладное значение. Архивирование индексов поисковых систем важно для интернет-приложений, коллекций аудио и видео, биологических баз данных. Хочется найти такое представление данных, при котором, по возможности, одновременно минимизируются и размер, и время доступа. Второй областью приложения является верификация программ и микросхем. Обычно для верификации необходимо проверить некое свойство множества допустимых состояний программы и графа переходов. Для современных систем число состояний не поддается никаким переборным алгоритмам. Единственный выход — хранить его в неявном виде и проверять его свойства без явного порождения.

После построения ряда алгоритмов, имеющих полиномиальную сложность относительно *размера сжатых текстов*, исследования столкнулись с NP-трудными задачами: принадлежность сжатого текста контекстно-свободной грамматике [33], двумерный поиск подстрок [10]. Таким образом, представление текстов в виде прямолинейных программ позволяет решить далеко не все задачи без порождения исходного текста. Кроме того, недавно удалось доказать [34], что принадлежность регулярному языку является P-полной (то есть плохо поддается распараллеливанию).

Экспериментальный анализ алгоритмов на сжатых текстах можно найти в статье [37], работы [41, 43] дают обзор всех алгоритмических исследований по сжатым текстам, книга [1] служит прекрасным введением

в текстовые алгоритмы.

1.4.2 Исследования по периодичности текстов и по обобщениям этого понятия

Одним из важнейших результатов для классической периодичности является теорема Файна и Вильфа [19]. Они изучали необходимые и достаточные условия, при которых из p-периодичности и q-периодичности следует НОД(p, q)-периодичность (НОД = наибольший общий делитель). Мы предприняли попытку найти аналогичное свойство для разреженных периодов. К нашему удивлению, теорема Файна и Вильфа не обобщается на разреженную периодичность. Даже два *чистых* разреженных периода одного текста могут не иметь никакого общего подпериода.

Наше обобщение понятия периодичности является далеко не первым. Недавно Симпсон и Тайдеман обобщили теорему Файна и Вильфа на многомерные периодичности [45]. Они считают, что некоторый вектор $\bar{v}$ является периодом многомерной раскрашенной клетчатой фигуры, если любая клетка x покрашена в тот же цвет, что и $x + \bar{v}$. Другое обобщение теоремы Файна и Вильфа можно найти в статье [17].

Берстель и Боассон [14], а вслед за ними Шур и Коновалова (Гамзова) [7, 44] и Бланше–Садри [11, 12, 13] подробно изучили частичные слова и ввели понятие периодичности для них. Однако, в этой постановке не допускается перекрытие периодов (копии периода должны идти одна вслед за другой). Таким образом, в этой модели только частичные слова могут иметь разреженные периоды (то есть периоды с пропусками букв).

В работе [9] Апостолико, Фарач и Илиопулос ввели понятие накрытия (cover). Слово C называется накрытием для слова T, если каждая

буква из T находится внутри некоторого вхождения C в T. Таким образом, эта модель отличается от нашего понятия в двух аспектах: накрытия связны (никаких пропусков), и накрытия могут перекрываться.

Катона и Шаш в своей работе [28] предложили вариант разреженной периодичности на плоскости. К сожалению, они рассмотрели только периоды (шаблоны) из двух клеток.

Наконец, понятие периодичности было обобщено на бесконечные слова под именем *почти периодических последовательностей* [6].

1.5 Структура работы

В главе 2 мы подробно обсудим, что мы понимаем под "сжатым текстом". Мы опишем абстрактную модель сжатого текста (прямолинейная программа), и сформулируем теорему Риттера о преобразовании архивов общепринятых стандартов в прямолинейные программы.

Далее, в главе 3, мы подробно представим алгоритмы для поиска явно заданных подпоследовательностей и сжатых подстрок в сжатом тексте. Четвертая глава посвящена доказательству вычислительной трудности нахождения расстояния Хэмминга между сжатыми текстами и проверки вложимости одного сжатого текста в другой. В пятой главе мы введем понятие разреженной периодичности и докажем ряд его свойств.

В конце каждой главы подводятся краткие итоги и ставятся задачи для дальнейших исследований.

Глава 2

Грамматика как модель сжатого текста

Мы рассматриваем конечный алфавит Σ, строкой (текстом, словом) называется любая конечная последовательность его букв. В этой работе под сжатыми текстами мы понимаем строки, порожденные *прямолинейными программами* (ПП). Неформально, прямолинейная программа — это контекстно-свободная грамматика, порождающая только одно слово.

Определение: прямолинейной программой называется контекстно-свободная грамматика $\mathcal{P}$, в которой нетерминальные символы $X_1, \ldots, X_m$ упорядочены (X_m — стартовый символ), и где у каждого нетерминального символа есть только одно правило: $X_i \to a$, где a — терминал, или $X_i \to X_j X_k$ для некоторых $j, k < i$.

Пример: текст *abaababaabaab* может быть представлен следующей прямолинейной программой:

$$X_1 \to b,$$

$$X_2 \to a,$$

$$X_3 \to X_2 X_1,$$

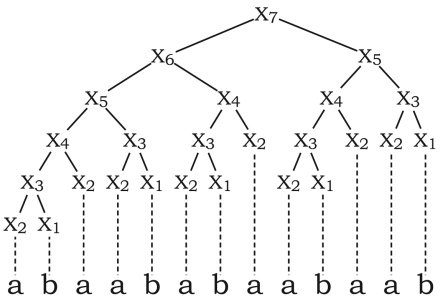
$$X_4 \to X_3 X_2,$$

$$X_5 \to X_4 X_3,$$

$$X_6 \to X_5 X_4,$$

$$X_7 \to X_6 X_5.$$

Восстановление этого текста из ПП можно проиллюстрировать в виде дерева:



Мы позволим себе некоторую вольность и будем пользоваться обозначением X_k как для символа грамматики, так и для *соответствующего текста*, получаемого при распаковке X_k. Таким образом, мы можем написать $T = X_m$.

В работе Риттера [42] доказано, что архив, полученный почти каждым прикладным архиватором (включая LZ1, LZ77, LZ78, LZW, run-length encoding, все словарные методы), может быть эффективно преобразован в прямолинейную программу почти того же размера. Важно отметить, что прямолинейные программы являются моделью *декомпрессора*, то есть моделируют лишь получение исходного текста из сжатого

представления (набора правил). В данной работе мы совершенно не заботимся о том, как были получены сжатые тексты.

Трудность обработки сжатых текстов связана с тем, что соотношение между размером прямолинейной программы и длиной порождаемого текста может быть экспоненциальным. Таким образом, любой алгоритм, который восстанавливает текст, имеет по крайней мере экспоненциальною сложность (по памяти, а следовательно и по времени).

Глава 3

Алгоритмы обработки сжатых текстов

3.1 Поиск сжатой подстроки в сжатом тексте

Формулировка "да/нет" варианта поиска сжатого шаблона в сжатом тексте выглядит следующим образом:

ВХОДНЫЕ ДАННЫЕ:
Две прямолинейные программы, порождающие P и T
ОТВЕТ:
Да/Нет (входит ли P в T как подстрока?)

Другие варианты: найти первое вхождение, найти количество всех вхождений, проверить, есть ли вхождение, начиная с данной позиции. Мы построим такую структуру данных, пользуясь которой можно быстро ответить на все поставленные вопросы.

Мы будем использовать одно свойство модели вычислений, которое неявно предполагалось и во всех предыдущих работах. А именно, при анализе алгоритма мы считаем, что каждая арифметическая операция с позициями в тексте занимает один шаг. С теоретической точки зрения, позиции в тексте могут быть натуральными числами длиной до $\log |T|$ бит в двоичной записи. Таким образом, чтобы получить оценку трудоемкости в *побитовых операциях* нужно умножить наш результат $O(n^2 m)$ на некоторый множитель, полиномиально зависящий от $\log |T|$.

Изложение алгоритма разбито на три этапа. На первом мы представим нашу структуру данных — *таблицу прогрессий*. Мы также продемонстрируем, как с помощью этой таблицы быстро решаются все разновидности задачи поиска подстрок в сжатых текстах. На втором этапе будет построен алгоритм вычисления таблицы прогрессий, основанный на процедуре локального поиска. Наконец, в третьей части мы опишем алгоритм для локального поиска.

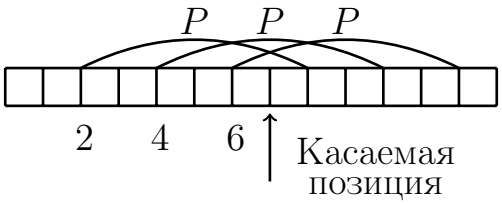
3.1.1 Поиск подстрок с помощью таблицы прогрессий

Обозначения и терминология. Мы называем *позициями* в тексте координаты воображаемых точек между последовательными буквами. Таким образом, в тексте $a_1 \dots a_n$ есть позиции $0, \dots, n$, где 0 находится перед первой буквой, а позиция n — сразу за последней. Мы говорим, что некоторая подстрока *касается* данной позиции, если позиция находится внутри или на границе подстроки. Под термином *вхождение* мы подразумеваем и соответствующую подстроку, и ее стартовую позицию. Надеемся, что точный смысл всегда будет ясен из контекста.

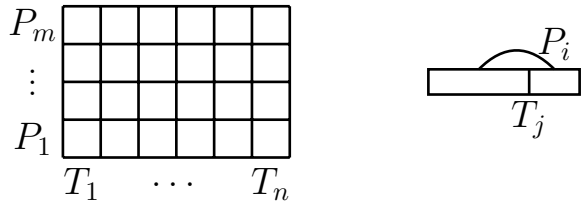
Пусть $P_1, \ldots, P_m$ и $T_1, \ldots, T_n$ — это нетерминальные символы прямолинейных программ, порождающих P и T, соответственно. Для каждого из этих промежуточных текстов мы определим специальную позицию *разреза*. В текстах, образованных по правилу $X_i = X_r X_s$, позицией разреза будет как раз точка, где заканчивается X_r и начинается X_s. Для полноты терминологии будем считать позицию 0 разрезом в однобуквенных текстах. Для примера из Главы 2, позицией разреза в тексте X_6 будет точка между пятой и шестой буквой: $abaab|aba$, так как X_6 был получен конкатенацией $X_5 = abaab$ и $X_4 = aba$.

Наш алгоритм основан на одном комбинаторном свойстве строк (оно также было использовано в [35]):

Лемма 3.1.1 (Основная лемма)**.** *Рассмотрим все вхождения P в T, касающиеся некоторой фиксированной позиции в тексте. Тогда их стартовые позиции образуют арифметическую прогрессию.*



Определим *таблицу прогрессий* следующим образом. Для каждых $1 \le i \le m, 1 \le j \le n$ значением $TP[i,j]$ будет код арифметической прогрессии тех вхождений P_i в T_j, которые касаются точки разреза T_j. Заметим, что любая арифметическая прогрессия может быть описана тремя числами: первым элементом, шагом, количеством элементов. Если $|T_j| < |P_i|$, мы формально определим $TP[i,j] = \varnothing_1$, а если текст достаточно велик, но не содержит ни одного вхождения, касающегося точки разреза, установим $TP[i,j] = \varnothing_2$.



Алгоритмическая задача 1: Используя таблицу прогрессий (на самом деле, достаточно ее верхней строки), массивы длин и позиций разреза всех промежуточных текстов решить каждый вариант задачи поиска подстрок в сжатых текстах за $O(n)$ шагов.

Алгоритмическая задача 2: Вычислить массивы длин и позиций разреза всех промежуточных текстов и заполнить таблицу прогрессий за $O(n^2m)$ шагов.

Построим алгоритмы для первой задачи. Для "да/нет" варианта мы можем воспользоваться следующим правилом: P входит в T тогда и только тогда, когда существует j, для которого $TP[m, j]$ не пусто. Другими словами, хотя бы один из текстов T_j содержит вхождение $P = P_m$, касающееся его точки разреза. Считающая и проверяющая версии поиска подстрок решаются чуть сложнее.

Рекурсивный алгоритм для проверки вхождения с данной позиции: проверить, касается ли тестируемая подстрока точки разреза текущего текста T_j. Если да, нужно посмотреть в TP-таблицу и проверить принадлежность соответствующей арифметической прогрессии. В противном случае нужно рекурсивно перейти к левой или правой компоненте рассматриваемого текста.

Мы будем считать количество вхождений P в тексты $T_1, \ldots, T_n$ по индукции. Для однобуквенных текстов мы просто берем мощность соответствующей арифметической прогрессии из TP-таблицы. Индукционный

переход: сложить число вхождений для правой части, для левой части текста и добавить мощность арифметической прогрессии “центральных вхождений”, исключив “только-касающиеся” вхождения.

Решение второй задачи приводится в следующих двух подразделах.

3.1.2 Вычисление таблицы прогрессий

Схема вычисления таблицы прогрессий:

1. Предвычисления: считаем длины, точки разрезов, первые и последние буквы всех промежуточных текстов;
2. Вычисляем строки и столбцы таблицы прогрессий, соответствующие однобуквенным текстам;
3. От самого маленького текста к самому большому, от самого маленького шаблона к самому большому последовательно вычисляем элементы таблицы $TP[i, j]$:
   (a) Находим вхождения бóльшей компоненты шаблона P_i вокруг разреза текста T_j;
   (b) Находим вхождения меньшей компоненты шаблона P_i, которые стыкуются с уже найденными вхождениями большей половины;
   (c) Пересекаем вхождения большей и меньшей компонент, приводим ответ к коду одной прогрессии.

Шаг 1: проводим предвычисления свойств промежуточных текстов. Индуктивно, от меньших номеров к большим, мы вычисляем

массивы длин, позиций разрезов, первых и последних букв для текстов $P_1, \ldots, P_m, T_1, \ldots, T_n$ за время $O(n+m)$.

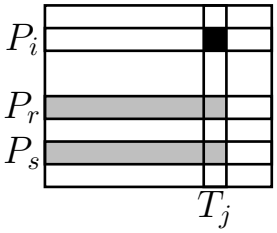
Шаг 2: заполняем "однобуквенные" строки/столбцы таблицы прогрессий. Пусть для какого-то i мы знаем, что $|P_i| = 1$. Тогда мы заполним i-ую строку таблицы по следующему принципу. Для каждого j мы сравним букву P_i либо с единственной буквой T_j (в случае $|T_j| = 1$), либо с последней буквой левой части T_j и первой буквой правой части T_j (эти две буквы мы знаем из предвычислений). В ответ запишем код прогрессии, кодирующей найденные совпадения (в данном случае ноль, одну или две соседние позиции). Таким образом, на один элемент таблицы мы тратим константное время.

Пусть теперь для некоторого j оказалось, что $|T_j| = 1$. В этом случае мы заполним j-ый столбец следующим образом. Если $|P_i| > 1$, мы сразу пишем $\varnothing_1$, в противном случае сравниваем две буквы. При удаче — записываем в ответ прогрессию из одной позиции. Здесь также достаточно константного времени на каждый элемент.

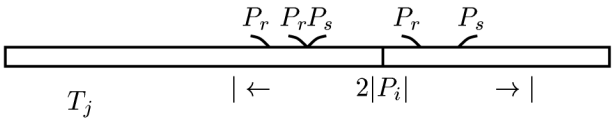
Шаг 3: вычисляем новый элемент таблицы прогрессий по "правилу пересечения". После "однобуквенных рядов" мы заполняем таблицу в алфавитном порядке пар (i, j). Пусть $P_i = P_r P_s$. Для вычисления $TP[i, j]$ мы будем использовать уже вычисленные значения $TP[r, 1], \ldots, TP[r, j]$ и $TP[s, 1], \ldots, TP[s, j]$. Другим словами, нам потребуется только информация о вхождении правой и левой части шаблона в текущий текст и все предшествующие тексты.

Схема использования
уже вычисленных элементов
при динамическом программировании

$P_i = P_r P_s$



Опишем *правило пересечения* для вычисления очередного элемента таблицы прогрессий при составном шаблоне и составном тексте. Наиболее естественным является следующий способ: (1) найти все вхождения P_r "вокруг" точки разреза T_j, (2) найти все вхождения P_s "вокруг" точки разреза T_j, (3) сдвинуть вторые вхождения на $|P_r|$ и пересечь с первыми.



В работе Миязаки и др. [35] представлен алгоритм, который реализует правило пересечения за $O(mn)$. Мы же не будем целиком следовать предложенной схеме, что позволит нам в итоге затратить лишь $O(n)$ шагов на вычисление $TP[i, j]$.

Итак, пусть $P_i = P_r P_s$, обозначим через γ позицию разреза T_j (мы знаем это значение из предвычислений). Не умаляя общности, будем считать $|P_r| \geq |P_s|$. Заметим, что любое вхождение P_i, касающееся разреза γ в тексте T_j, состоит из вхождения шаблона P_r и вхождения P_s внутри $|P_i|$-окрестности разреза. При этом P_r заканчивается в точке, откуда начинается P_s. Первый шаг правила пересечения мы оставим без изменений. Но на втором шаге мы будем искать *только вхождения P_s, которые начинаются с окончаний уже найденных вхождений P_r.*

Для наших вычислений мы разработали вспомогательную процедуру *локального поиска*, которая извлекает нужную информацию из уже со-

считанных элементов таблицы прогрессий. Процедура Local$(i, j, [\alpha, \beta])$ возвращает вхождения P_i в T_j, лежащие целиком внутри интервала $[\alpha, \beta]$. Важные характеристики алгоритма локального поиска:

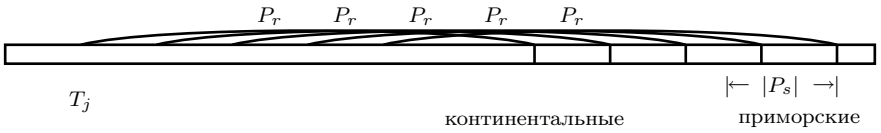
1. он корректно работает только при условии $|\beta - \alpha| \leq 3|P_i|$,
2. локальный поиск использует значения $TP[i, k]$ для $1 \leq k \leq j$,
3. его трудоемкость $O(j)$ шагов,
4. на выходе локальный поиск выдает пару арифметических прогрессий, причем внутри каждой прогрессии все вхождения имеют общую точку, и все вхождения из первой прогрессии находятся левее всех вхождений из второй.

Теперь покажем, как вычислить $TP[i, j]$ за 5 запусков локального поиска.

Шаг 3a: ищем бóльшую часть шаблона. С помощью одного вызова локального поиска мы найдем все вхождения P_r в интервале $[\gamma - |P_i|, \gamma + |P_r|]$, длина которого $|P_i| + |P_r| \leq 3|P_r|$. В ответе мы получим две арифметические прогрессии, представляющие все потенциальные стартовые позиции для вхождений P_i, касающихся разреза. К сожалению, мы не можем поступить таким же образом с P_s, так как длина соответствующего интервала поиска может *не быть константной* относительно $|P_s|$. Поэтому мы будем искать только те вхождения P_s, которые начинаются с позиций из двух арифметических прогрессий *окончаний* вхождений P_r.

Шаг 3b: ищем меньшую часть шаблона. Будем обрабатывать каждую прогрессию по отдельности. Назовем точку окончания вхождения P_r *континентальной*, если она находится на расстоянии хотя бы

$|P_s|$ от последнего окончания в прогрессии. Все остальные окончания будем называть *приморскими*.



По свойству 4 локального поиска, все вхождения P_r из одной прогрессии имеют общую точку. Следовательно, все подстроки длины $|P_s|$, стартующие с континентальных окончаний, полностью совпадают. Таким образом, нам необходимо проверить все приморские точки и *одну* континентальную. Для поиска вхождений в приморском районе мы применим локальный поиск P_s в $|P_s|$-окрестности последнего окончания и пересечем ответ с прогрессией приморских окончаний вхождений P_r. Пересечение двух прогрессий занимает $O(\log |T|)$ шагов, по технике эта операция очень похожа на расширенный алгоритм Евклида. Для проверки первого континентального окончания мы применим локальный поиск к $|P_s|$-строке, начинающейся с этого окончания.

Шаг 3c: приводим ответ к одной прогрессии. Чтобы получить множество всех вхождений P_i, касающихся разреза в T_j, остается сделать следующее. В зависимости от результатов проверки, возьмем или все континентальные окончания, или ни одно из них. Возьмем подпрогрессию приморских окончаний (тех, которые соответствуют началам вхождений P_s). Также, возьмем аналогичные множества для второй прогрессии вхождений P_r. Эти четыре части являются арифметическими прогрессиями, идущими одна после другой. Следовательно, мы можем упростить ответ (мы должны получить лишь одну прогрессию по Основной лемме) за константное время.

Анализ трудоемкости. Давайте оценим трудоемкость вычисления одного элемента таблицы прогрессий. Мы применили один локальный поиск для P_r, четыре локальных поиска для P_s и дважды вычисляли пересечения арифметических прогрессий. Таким образом, мы имеем 7 процедур по $O(n)$ шагов.

Замечание. В представленном алгоритме для вычисления нового элемента таблицы мы пользуемся локальным поиском, а в реализации локального поиска — значениями из таблицы прогрессий. Убедимся, что в нашем случае нет никакого порочного круга. Действительно, пусть мы вычисляем элемент таблицы в строке i, и $P_i = P_r P_s$. Тогда мы пользуемся локальным поиском для шаблонов P_r и P_s. Эти процедуры активно используют значения таблицы из строк r и s. Но, согласно определенному нами порядку заполнения таблицы, эти строки находятся ниже строки i и уже заполнены.

3.1.3 Процедура локального поиска

Локальный поиск находит в хождения P_i в подстроке $T_j[\alpha, \beta]$. Наше решение состоит из *процедуры обхода* и *процедуры упрощения*, которые мы подробно опишем ниже.

Первый этап: запускаем рекурсивную процедуру обхода с параметрами (i, j, α, β). После завершения работы всех веток получаем отсортированный список прогрессий вхождений P_i в T_j в пределах $[\alpha, \beta]$.

Второй этап: запускаем процедуру упрощения и приводим список прогрессий к двум прогрессиям.

Процедура обхода. На входных данных $(i, j, [\alpha, \beta])$ мы делаем следующее.

1. Берем (из таблицы прогрессий) вхождения P_i в T_j, которые касаются разреза.

2. Обрезаем прогрессию этих вхождений, оставляя лишь находящиеся целиком внутри $[\alpha, \beta]$.

3. Записываем эту усеченную прогрессию в список ответов.

4. Проверяем, имеет ли пересечение $[\alpha, \beta]$ с левой/правой частью T_j длину хотя бы $|P_i|$.

5. Если да, то делаем рекурсивный вызов процедуры обхода с тем же i, индексом левой/правой части T_j, и интервалом пересечения.

Процедура обхода формирует динамический двунаправленный список групп вхождений шаблона (список ответов). В дополнение к основным параметрам локального поиска, мы вводим два вспомогательных параметра. Это указатель на элемент динамического списка ответов и "глобальный сдвиг". Когда мы находим очередные вхождения при "внутреннем" вызове процедуры обхода, мы добавляем к найденной позиции текущее значение сдвига. При этом, полученный ответ мы вписываем в прямо перед тем элементом, куда ведет указатель. При последующих рекурсивных вызовах для левой половины сдвиг не меняется, для правой к сдвигу нужно добавить длину левой половины. Указатели для левого/правого рекурсивного вызовов указывают, соответственно, на только что записанный ответ и на следующий за ним элемент.

В тексте T_j рассмотрим положение всех интервалов, с которыми мы работали во время процедуры обхода. Любая пара из них или вложена, или не пересекается. Причем для каждого k интервалы, на которых процедура обхода запускалась с параметром T_k, не могут быть вложенными. Напомним, мы требуем, чтобы исходный интервал был не больше $3|P_i|$. Следовательно, для каждого k во время обхода было не больше трех интервалов, ассоциированных с T_k. Таким образом, при обходе мы сделали не более $3j$ рекурсивных вызовов, и общее время работы не превосходит $O(j)$.

Мы предполагаем, что процедура обхода поддерживает указатель на соответствующее место в списке ответов. Это означает, что в конце работы мы получим *отсортированный* список не более, чем из $3j$ прогрессий. Здесь отсортированный означает, что последний элемент k-ой прогрессии находится левее или равен первому элементу $(k+1)$-ой прогрессии. Заметим, что, по построению процедуры обхода, выдаваемые прогрессии могут пересекаться только по граничным элементам.

Процедура упрощения. В процедуре упрощения мы пройдем по всему полученному списку прогрессий. А именно, мы будем сравнивать расстояние между последним элементом текущей прогрессии и первым элементом следующей с шагом каждой из этих прогрессий. Если все эти три числа равны друг другу, мы объединяем прогрессии. В противном случае мы объявляем новую прогрессию ответов.

Давайте мысленно вобьем два "колышка" в интервал $[\alpha, \beta]$ так, чтобы разделить его на три приблизительно равных части. Если мы применим Основную лемму к позициям $\delta_1 = \lfloor \frac{2\alpha+\beta}{3} \rfloor$ и $\delta_2 = \lfloor \frac{\alpha+2\beta}{3} \rfloor$, мы узнаем, что все вхождения P_i в $[\alpha, \beta]$ образуют две (одна после другой) арифмети-

ческие прогрессии. А именно, вхождения, касающиеся δ_1, и вхождения, не касающиеся δ_1, но касающиеся δ_2. Здесь мы пользуемся неравенством $\beta - \alpha \leq 3|P_i|$, из которого следует, что каждое вхождение P_i должно задеть δ_1 или δ_2. Таким образом, при процедуре упрощения мы будем объявлять новую прогрессию не больше одного раза.

3.1.4 Обсуждение алгоритма

Поиск сжатого шаблона в сжатом тексте (общий вид):

1. Предвычисления: находим длины промежуточных текстов и точки разрезов;
2. Последовательно, элемент за элементом, заполняем таблицу прогрессий;
3. Получаем нужные ответы (существование вхождения, первое вхождение, число вхождений).

Отметим два возможных улучшения алгоритма. Посмотрим внимательно на процедуру вычисления нового элемента таблицы прогрессий. Заметим, что локальный поиск работает за время $O(h)$, где h — *высота* прямолинейной программы (точнее, высота соответствующего дерева распаковки), порождающей T. Пересечение арифметических прогрессий требует всего $O(\log |T|)$ шагов. Таким образом, если бы нам удалось "сбалансировать" любую прямолинейную программу до $O(\log |T|)$ высоты, мы немедленно получили бы $O(nm \log |T|)$ оценку времени работы алгоритма.

Теперь взглянем на правила порождения текста. Коллаж-системы (collage systems) [29] и LZ77 [47] используют конкатенацию и *обрезку*.

Конечно, как показал Риттер [42], мы можем оставить только конкатенации, расширив размер архива лишь в $O(\log |T|)$ раз. Но есть надежда, что представленный алгоритм может работать напрямую с текстами, построенными по правилам конкатенации/обрезки.

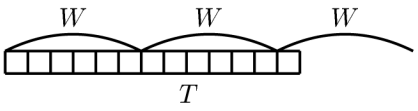
Еще одной ближайшей задачей является преобразование таблицы прогрессий в прямолинейную программу размера $O(n^2)$, описывающую все вхождения P в T. Здесь под описанием мы понимаем бинарную строку, в которой единицы стоят в точности на местах первых символов вхождений шаблона в текст.

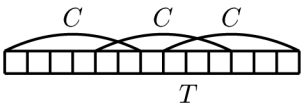
3.2 Алгоритм вычисления периодов и накрытий

Основной идеей нашего алгоритма поиска подстрок в сжатых текстах было вычисление таблицы прогрессий. По сути, мы проводили частичную распаковку текста, извлекая лишь существенную для поиска подстроки информацию. В этом разделе мы покажем, что быстрый алгоритм построения таблицы прогрессий помогает вычислить периоды и накрытия сжатого текста за полиномиальное (относительно сжатого размера) время.

Периодом строки T назовем такую строку W (и ее длину $|W|$), что T является префиксом W^k для некоторого натурального k. *Накрытием* (от "cover", термин введен в работе [9]) строки T называется такая строка C, что каждый символ в T накрыт каким-то вхождением C в T. Каждый период и каждое накрытие однозначно задаются своей длиной, так как, по определению, они являются префиксами T. В этом разделе мы будем

использовать обозначение $t = |T|$.





Задача об определении *периода/накрытия сжатого текста:* дан текст T, представленный прямолинейной программой. Найти длину минимального периода/накрытия и вычислить "сжатое" представление всех периодов и накрытий.

Наши алгоритмы основаны на комбинаторных свойствах текстов, которые были использованы при решении классических постановок (без сжатия) рассматриваемых задач:

1. У строки длины t длины периодов в интервале $[t - \frac{t}{2^k}, t - \frac{t}{2^{k+1}}]$ образуют одну арифметическую прогрессию.

2. Каждое накрытие является *границей* (border), то есть одновременно суффиксом и префиксом текста. Если какая-то граница с длиной в интервале $[\frac{t}{2^{k+1}}, \frac{t}{2^k}]$ является накрытием, то все меньшие границы в этом интервале также являются накрытиями.

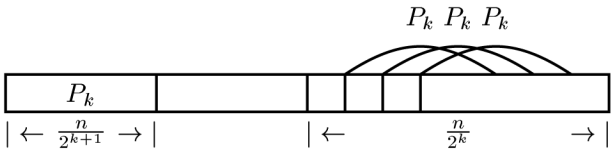
3. Каждый текст имеет период длины u тогда и только тогда, когда у него есть граница $t - u$.

4. По прямолинейной программе размера n, порождающей текст T, и по числам α, β мы можем построить за линейное (от n) время ПП линейного размера, порождающую подстроку $T[\alpha, \beta]$.

Алгоритм вычисления периодов сжатого текста. В этом разделе мы часто используем позиции в тексте, представленные в виде дробей.

Если такая дробь несократима, то мы имеем в виду ее нижнюю целую часть, но для компактности записи мы опускаем знаки целой части. Используя Свойство 3, мы будем искать границы вместо периодов, причем будем вести отдельный поиск в каждом интервале $[\frac{t}{2^{k+1}}, \frac{t}{2^k}]$. Зафиксируем k. Теперь составим два списка "кандидатов в границы".

Шаг 1: возьмем $\frac{t}{2^{k+1}}$-префикс текста T (обозначим его P_k) и найдем все его вхождения в $\frac{t}{2^k}$-суффикс T. По Основной лемме эти вхождения образуют одну арифметическую прогрессию. Первым списком кандидатов будут расстояния от начал вхождений P_k до конца текста T.

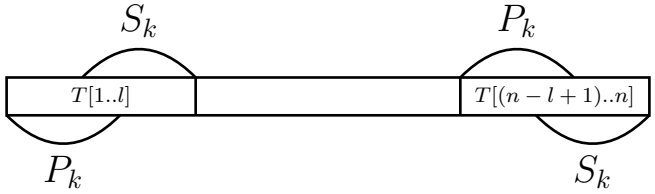


Шаг 2: возьмем $\frac{t}{2^{k+1}}$-суффикс текста T (обозначим его S_k) и найдем все его вхождения в $\frac{t}{2^k}$-префикс T. По Основной лемме эти вхождения образуют одну арифметическую прогрессию. Вторым списком кандидатов будут расстояния от начала текста T до концов вхождений S_k.

Шаг 3: пересечем две прогрессии, полученные на первых двух шагах. Мы получим в точности длины всех границ текста в интервале $[\frac{t}{2^{k+1}}, \frac{t}{2^k}]$.

Поиск периодов в сжатом тексте:

1. Для каждого k от 0 до $\lceil \log |T| \rceil$ мы
   (a) Строим прямолинейные программы для описания P_k, S_k;
   (b) Строим таблицы прогрессий для пар (T, S_k) и (T, P_k);
   (c) Запускаем процедуру локального поиска P_k в $\frac{t}{2^k}$-суффиксе T;
   (d) Запускаем процедуру локального поиска S_k в $\frac{t}{2^k}$-префиксе T;
   (e) Пересекаем две полученные прогрессии.
2. Чтобы получить длины периодов, надо из t вычесть полученные длины границ.



Корректность: пусть число $l \in [\frac{t}{2^{k+1}}, \frac{t}{2^k}]$ попало в оба списка кандидатов. Тогда у строк $T[1..l]$ и $T[(t-l+1)..t]$ совпадают первые $\frac{t}{2^{k+1}}$ и последние $\frac{t}{2^{k+1}}$ букв. Таким образом, они полностью совпадают и l является границей. Сложность: согласно Свойству 4, длина прямолинейных программ, описывающих суффиксы и префиксы, линейна относительно n. Следовательно, построение двух таблиц прогрессий займет $O(n^3)$. Локальный поиск и пересечение прогрессий занимает $O(n)$ шагов. Таким образом, суммарное время работы алгоритма по всем интервалам составляет $O(n^3 \log |T|)$.

Задача о нахождении периода сжатого текста была поставлена в 1996 в расширенных тезисах [20]. К сожалению, полное описание алгоритма

из этой работы (он требовал $O(n^5 \log^3 |T|)$ шагов) так и не было опубликовано.

Алгоритм нахождения накрытий сжатого текста. Будем вести поиск по отдельности внутри каждого интервала $[\frac{t}{2^{k+1}}, \frac{t}{2^k}]$. Как было показано выше, мы можем найти все границы в таком интервале за время $O(n^3)$. Согласно Свойству 2, достаточно применить бинарный поиск с помощью "проверки накрывания", чтобы найти все накрытия в интервале.

Проверка накрывания (проверить по сжатым текстам C и T, является ли C накрытием для T):

1. Строим таблицу прогрессий для C и T.
2. Для каждого промежуточного текста T_j с помощью процедуры локального поиска делаем следующую проверку. Рассматриваем $|C|$-окрестность точки разреза T_j, исключая лишь позиции, которые находятся ближе чем $|C|$ к краям T_j. Убеждаемся, что этот интервал полностью покрыт C-вхождениями.
3. С помощью локального поиска проверяем, что $|C|$-префикс и $|C|$-суффикс текста T полностью покрыты C-вхождениями.

Корректность: по индукции можно доказать, что C является накрытием тогда и только тогда, когда мы получили ответ "да" на все проверки шагов 2 и 3. Сложность: поиск границ по всем интервалам занимает $O(n^3 \log |T|)$ время. Проверка накрывания требует $O(n^3)$ шагов, так как первый шаг требует $O(n^3)$, второй — $O(n^2)$, а третий — $O(n)$. Мы вызываем процедуру проверки накрывания не более $\log |T|$ раз в каждом из

$\log |T|$ интервалов. Таким образом, суммарное время работы алгоритма по всем интервалам составляет $O(n^3 \log^2 |T|)$ шагов.

3.3 Алгоритм поиска оконных подпоследовательностей

В этом разделе мы опишем алгоритм для проверки вложимости шаблона P в текст T, представленный прямолинейной программой.

На практике часто требуется найти "достаточно компактную" подпоследовательность. Минимальным окном в тексте T назовем такую подстроку S текста T, которая 1) содержит P, как подпоследовательность, и 2) никакая собственная подстрока S не обладает свойством 1. Мы будем также называть w-окном подстроку текста T длины w.

Мы рассматриваем следующие пять задач:

1. Определить, является ли шаблон P подпоследовательностью в тексте T.

2. Сосчитать количество минимальных окон в тексте T, в которые вкладывается шаблон P.

3. Определить, вкладывается ли шаблон P в какое-нибудь w-окно текста T.

4. Сосчитать количество w-окон в тексте T, в которые вкладывается P.

5. Сосчитать количество минимальных окон в T размером не больше w, в которые вкладывается P.

Для решения пяти вышеописанных задач мы определим вспомогательные структуры данных. Далее мы покажем, как вычислить эти структуры и обсудим итоговый алгоритм.

3.3.1 Вспомогательные структуры данных

С этого момента мы рассматриваем текст T, сжатый прямолинейной программой $\mathcal{X}$ размера m. Пусть $|P| = k$ и $P_1, \ldots, P_l$ — все различные подстроки шаблона P. Заметим, что $l < k^2$.

Теперь мы определим пять вспомогательных структур данных.

Левые вложения. Для каждого нетерминала X_i прямолинейной программы $\mathcal{X}$ и подстроки шаблона P_j мы определяем $L_{i,j}$ как длину минимального *префикса* строки X_i, в которую вкладывается P_j. Если такого префикса нет, мы устанавливаем $L_{i,j} = \infty$.

Правые вложения. Для каждого нетерминала X_i прямолинейной программы $\mathcal{X}$ и подстроки шаблона P_j мы определяем $R_{i,j}$ как длину минимального *суффикса* строки X_i, в который вкладывается P_j. Если такого префикса нет, мы устанавливаем $R_{i,j} = \infty$.

Минимальные окна. Для каждого нетерминала X_i прямолинейной программы $\mathcal{X}$ мы определим MW_i как число минимальных окон в строке X_i, в которые вкладывается P.

Фиксированные окна. Для каждого нетерминала X_i прямолинейной программы $\mathcal{X}$ мы определим FW_i как число w-окон в строке X_i, в которые вкладывается P.

Ограниченные минимальные окна. Для каждого нетерминала X_i прямолинейной программы $\mathcal{X}$ мы определим BMW_i как число окон размером не больше w, в строке X_i, в которые вкладывается P.

3.3.2 Вычисление вспомогательных структур данных

Покажем, как вычислить значения элементов пяти вышеописанных массивов.

Левые вложения. Будем проводить вычисления вдоль конструкции прямолинейной программы. Мы легко можем вычислить значение L для однобуквенных символов и всех возможных P_i. Пусть теперь $X_i \to X_pX_q$. Если $L_{p,j} \neq \infty$, тогда просто $L_{i,j} = L_{p,j}$. В противном случае, мы должны найти такое разбиение P_j на две части P_uP_v, что и $L_{p,u}$, и $L_{q,v}$ конечны. Нам нужно найти разбиение, максимизирующее первую часть (то есть $|P_u|$). Мы будем делать это бинарным поиском. Если мы нашли такое разбиение, то $L_{i,j} = |X_p| + L_{q,v}$, иначе $L_{i,j} = \infty$. Таким образом, время работы на одном шаге — $O(\log k)$, общее время работы — $O(ml \log k) = O(mk^2 \log k)$.

Правые вложения. Вычисляем аналогично левым вложениям.

Минимальные окна. Будем использовать вычисления вдоль конструкции прямолинейной программы, а также данные о левых и правых вложениях. Для подсчета минимальных окон в $X_i \to X_pX_q$ мы должны сложить уже вычисленные значения для X_p и X_q и прибавить некоторое количество *пограничных* окон. Заметим, что для каждого разбиения $P = P_uP_v$ может быть не более одного минимального пограничного окна, в котором P_u вкладывается внутри X_p, P_v внутри X_q. Используя данные о правых и левых вложениях, мы определим, для каких разбиений такие пограничные минимальные окна есть. В этом месте нужно быть аккуратным. Одному минимальному окну может соответствовать несколько последовательных разбиений. Поэтому будем действовать следующим образом. Мы будем рассматривать все разбиения шаблона от “все в X_p”

до “все в X_q” и увеличивать счетчик пограничных окон только в случае, когда 1) первая часть разбиения вкладывается в X_p, а вторая в X_q; и 2) получившееся минимальное окно сдвинуто относительно предыдущего успешного вложения. Таким образом, трудоемкость вычисления этого массива равна $O(mk)$.

Фиксированные окна. Схема вычисления такая же, как и для минимальных окон. Здесь мы лишь покажем, как сосчитать пограничные фиксированные окна для $X_i \to X_pX_q$. Основное наблюдение: любое пограничное w-окно, в которое вкладывается P, также содержит *минимальное* окно, в которое вкладывается P. Также, как и в предыдущем абзаце, мы рассматриваем по очереди все разбиения шаблона P. Для каждого разбиения, пользуясь данными о правых и левых вложениях, мы находим минимальное окно, соответствующее этому разбиению. Сравнивая его с предыдущим, мы прибавляем к счетчику количество w-окон, которые содержат новое минимальное окно, но не старое. Трудоемкость вычисления этого массива также равна $O(mk)$

Ограниченные минимальные окна. Используем ту же технику, что и для минимальных окон. Просто при подсчете мы игнорируем слишком большие пограничные минимальные окна.

3.3.3 Итоговый алгоритм и его трудоемкость

Наши структуры содержат ответы для всех пяти задач:

1. Шаблон P является подпоследовательностью в тексте T тогда и только тогда, когда $L_{1,m} \neq \infty$ (мы считаем $P_1 = P$),

2. Количество минимальных окон в T, в которые вкладывается P, рав-

но MW_m,

3. Шаблон P является подпоследовательностью некоторого w-окна тогда и только тогда, когда $FW_m \neq 0$,

4. Количество w-окон, в которые вкладывается P, равно FW_m,

5. Количество окон размера не больше w, в которые вкладывается P, равно BMW_m.

Таким образом, итоговая трудоемкость нашего алгоритма для шаблона длины k и текста, представленного прямолинейной программой размера m, равна $O(mk^2 \log k)$.

Замечание. По теореме Риттера [43] по тексту, сжатому алгоритмом LZ78, можно построить прямолинейную программу, лишь линейно увеличив размер архива. Сама процедура преобразования также линейна. Это значит, что наши задачи могут быть решены за время $O(mk^2 \log k)$, уже где m — размер LZ78-сжатого текста.

Замечание. Для LZ77-сжатия также можно осуществить переход к прямолинейной программе [43]. Трудоемкость и размер получившейся программы оценивается как $O(m \log n)$, где m — размер LZ77-сжатого текста, а n — оригинального. Таким образом, применив сначала переход к прямолинейной программе, а затем наш основной алгоритм, мы получим трудоемкость $O(mk^2 \log k \log n)$ для LZ77-сжатия.

3.4 Выводы и открытые вопросы

Алгоритмические свойства грамматик. Желательно охарактеризовать все свойства, которые могут быть вычислены по сжатому тексту

за полиномиальное время, и найти для них оптимальные алгоритмы.

- Для поиска сжатого шаблона (размер архива m) в сжатой строке (размер архива n) мы построили алгоритм, работающий за время $O(n^2m)$. Можно ли решить эту задачу быстрее? *Гипотеза:* достаточно $O(nm \log |T|)$ времени. Более точно, мы надеемся, что каждый элемент таблицы прогрессий может быть вычислен за время $O(\log |T|)$.

- Можно ли ускорить вычисление редакторского расстояния (расстояния Левенштейна) в случае, когда один из текстов хорошо сжимается? Формально, можно ли вычислить редакторское расстояние за время $O(nm)$, где n — это длина T_1, а m — размер грамматики, порождающей T_2? Такой результат позволил бы ускорить вычисление редакторского расстояния для любого "сверхлогарифмического сжатия". Напомним, классический алгоритм требует $O(\frac{n^2}{\log n})$ шагов для решения этой задачи.

- Можно ли вычислить длину наибольшей общей подстроки двух текстов, представленных прямолинейными программами, за полиномиальное (от размера ПП) время?

- Можно ли ускорить предложенный в этой работе алгоритм для поиска оконных подпоследовательностей? Точнее, можно ли уменьшить k-зависимый сомножитель в оценке времени работы алгоритма?

Архивирование суффиксного дерева. Алгоритмы поиска явно заданного шаблона в сжатом тексте работают за время, близкое к линейному от размера архива. Но как провести предварительные вычисления,

если мы хотим подготовить текст ко многократному поиску шаблонов? Более формально, существует ли такая структура данных, которая (1) поддерживает поиск шаблона за время, линейное *относительно длины шаблона*, и (2) для "хорошо-сжимаемых" текстов имеет размер меньше, чем длина самого текста?

Применения к верификации. Одним из главных прорывов в верификации моделей программ является алгоритм символьной верификации. Он оперирует не с множествами состояний программы, а с описаниями этих множеств в виде упорядоченных двоичных решающих диаграмм (OBDD). Можно показать, что OBDD-представление можно перевести в представление грамматикой того же размера. С другой стороны, есть множества, для которых представление грамматикой будет экспоненциально короче. Чтобы встроить более выразительное описание множеств грамматиками в алгоритм символьной верификации, необходимо ответить на следующий вопрос. Как по двум множествам A и B, представленных грамматиками, получить *близкую к минимальной* грамматику, представляющую $A \cap B$, и грамматику, представляющую $A \cup B$?

Глава 4

Нижние оценки на трудоемкость обработки сжатых текстов

4.1 Расстояние Хэмминга между сжатыми текстами

Расстоянием Хэмминга называется количество различий между двумя текстами равной длины. Задача о *расстоянии Хэмминга между сжатыми текстами* (считающий вариант): даны два текста одинаковой длины T и S, представленные в виде прямолинейных программ. Найти расстояние Хэмминга $HD(T, S)$ между ними.

В теории сложности говорят, что функция принадлежит классу $\#\mathsf{P}$, если существует такая недетерминированная полиномиальная машина Тьюринга M, что значением функции будет число принимающих веток

M на соответствующих входных данных. Иначе говоря, существует такая полиномиально-вычислимая функция $G(x, y)$ и многочлен p, что

$$f(x) = \#\{y | G(x, y) = \text{“да”, и } |y| \leq p(|x|)\}.$$

Мы говорим, что функция f имеет [1]-Тьюринг сведение к функции g, если существуют такие полиномиально-вычислимые функции E и D, что $f(x) = D(g(E(x)))$. Функция называется #P-полной (относительно [1]-Тьюринг сведений), если она принадлежит классу #P, и любая другая функция этого класса имеет к ней [1]-Тьюринг сведение. Фразы “функция f является #P-полной” и “задача вычисления f является #P-полной” являются синонимами.

Теорема 4.1.1. *Задача о расстоянии Хэмминга между сжатыми текстами является #P-полной.*

Доказательство. Принадлежность #P. Для расстояния Хэмминга в качестве функции G мы можем взять сравнение текстов по одной позиции: $G(T, S; y) =$ “да”, если $T_y \neq S_y$. Тогда количество y, дающих ответ “да”, в точности равно расстоянию Хэмминга. Функция G — полиномиально вычислима, так как, зная длины всех вспомогательных текстов, мы можем пройти грамматику “сверху вниз”. Каждый раз спускаясь в нужную ветку, мы придем к символу T_y.

#P-полнота. Для доказательства полноты задачи достаточно свести к ней другую полную задачу в рассматриваемом классе. Напомним формулировку #P-полной задачи *сумма размеров* [2]: даны натуральные числа $w_1, \ldots, w_n, t$ в двоичной записи. Требуется определить, сколько существует таких наборов $x_1, \ldots, x_n \in \{0, 1\}$, что $\sum_{i=1}^{n} x_i \cdot w_i = t$. Иначе

говоря, сколько подмножеств $\bar{w} = \langle w_1, \ldots, w_n \rangle$ имеют сумму элементов, равную t.

Построим [1]-Тьюринг сведение от суммы размеров к расстоянию Хэмминга между сжатыми текстами. Зафиксируем входные данные для суммы размеров. Построим описания двух текстов прямолинейными программами так, чтобы по расстоянию между ними можно было определить ответ для суммы размеров.

Идея конструкции заключается в следующем. Пусть $s = w_1 + \cdots + w_n$. Оба текста будут иметь длину $(s+1)2^n$. Мысленно мы будем представлять их себе как последовательность 2^n блоков по $s+1$ символу каждый. Первый текст T будет кодировать число t. Все его блоки будут одинаковы. Все символы, кроме одного, — "0", единственная "1" стоит на $t+1$ месте. Блоки второго текста S будут соответствовать всем возможным подмножествам $\bar{w}$. В каждом из них единственная "1" будет стоять непосредственно после позиции, равной сумме элементов подмножества. Формулами тексты T и S можно записать так:

$$T = (0^t 10^{s-t})^{2^n}, \quad S = \prod_{x=0}^{2^n-1} (0^{\bar{x}\cdot\bar{w}} 10^{s-\bar{x}\cdot\bar{w}}).$$

Здесь $\bar{x} \circ \bar{w} = \sum x_i w_i$, а $\prod$ обозначает конкатенацию.

Строка S (будем называть ее *строка Лори*) впервые появилась в работе Маркуса Лори [33]. Лори доказал [33], что зная входные параметры задачи о сумме размеров, мы можем построить ПП полиномиального размера, описывающие строки S и T.

Заметим теперь, что $HD(T, S)$ равно удвоенному числу тех подмножеств $\bar{w}$, чья сумма не равна t. Таким образом, ответ для суммы размеров можно получить по формуле $2^n - \frac{1}{2}HD(T, S)$.

□

Оказывается, в классе #P-полных задач можно выделять подклассы, несводимые друг к другу по Карпу. Напомним здесь, что функция $f(x)$ сводится по Карпу к функции $g(x)$, если существует такая полиномиально вычислимая функция t, что $f(x) = g(t(x))$. Недавно [38] был введен интересный класс функций TotP. Функция f принадлежит классу TotP, если существует такая недетерминированная полиномиальная машина Тьюринга M, что $f(x)$ равно числу *всех* веток вычисления $M(x)$ минус один. В класс TotP попадают многие задачи, для которых "да/нет" версия (то есть проверка, равно ли $f(x)$ нулю) полиномиально разрешима, а считающий вариант — #P-полон.

Можно показать, что расстояние Хэмминга между сжатыми текстами принадлежит классу TotP. В самом деле, можно сравнивать интервалы исходных текстов и разветвлять вычисления всякий раз, когда обе половины интервала из первого текста не равны аналогичным участкам второго. Для каждой пары неравных текстов нужно также добавить дополнительную искусственную ветку вычислений, чтобы компенсировать вычитание единицы из определения принадлежности TotP.

4.2 Сжатые подпоследовательности в сжатых текстах

В этом разделе мы дадим две нижние оценки на вычислительную сложность следующей задачи:

- *Поиск сжатой подпоследовательности в сжатом тексте* (для краткости — *Вложимость*). Даны прямолинейные программы, представляющие строку P (шаблон) и строку T (текст). Требуется определить, являются ли буквы шаблона подпоследовательностью в строке T (обозначение: $P \hookrightarrow T$).

В этом разделе представлено два основных результата. Мы начнем со сведения известной NP-полной задачи (*Сумма размеров*) к задаче *Вложимости*. Таким образом, мы получим NP-трудность последней. Следующим шагом будет сведение *Вложимости* к *Невложимости* и наоборот. Немедленным следствием из этой симметричности станет coNP-трудность изучаемой задачи.

4.2.1 NP-трудность

На этот раз нам понадобится "да/нет" версия NP-полной задачи *Сумма размеров* (см. [2]):

- Дана двоичная запись натуральных чисел $w_1, \ldots, w_n, t$. Требуется определить, существуют ли $x_1, \ldots, x_n \in \{0, 1\}$ такие, что

$$\sum_{i=1}^{n} x_i \cdot w_i \quad = \quad t.$$

Теорема 4.2.1. *Сумма размеров сводится к Вложимости.*

Доказательство. Пусть $t, \bar{w} = \langle w_1, \ldots, w_n \rangle$ – входные данные *Суммы размеров* (будем считать, что $n > 1$). Мы построим такие прямолинейные программы $\mathcal{F}$ и $\mathcal{G}$ (порождающие тексты F, G), что подмножество $\bar{w}$ с суммой t существует тогда и только тогда, когда $F \hookrightarrow G$.

Введем обозначения $s = w_1 + \cdots + w_n$, $N = 2^n s$. Каждому подмножеству $\bar{w}$ можно поставить в соответствие строчку $x = \overline{x_1 \ldots x_n}$ длины n из нулей и единиц, то есть целое число от 0 до $2^n - 1$. По-прежнему, будем пользоваться обозначением $x \circ \bar{w} = \sum_{i=1}^{n} x_i w_i$. Фактически, $x \circ \bar{w}$ дает сумму подмножества $\bar{w}$, кодируемого числом x.

Конструкция. Наша задача "закодировать" каждый набор входных данных для *Суммы размеров* через входные данные для *Вложимости*. Опишем текстовые строки F и G:

$$G = G_0^{5N} \qquad G_0 = G_1 G_2 G_3 G_4$$

$$G_1 = \prod_{x=0}^{2^n-1} (10^s) = (10^s)^{2^n} \qquad G_2 = 0^{2N}$$

$$G_3 = \prod_{x=0}^{2^n-1} (0^{x \circ \bar{w}} 10^{s - x \circ \bar{w}}) \qquad G_4 = 0^{t+1}$$

$$F = F_0^{5N-1} \qquad F_0 = 10^{3N+t} 10^{N+1}$$

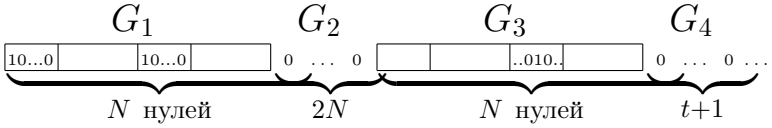
Мы использовали знак $\prod$ для обозначения конкатенации соответствующих слов в порядке изменения индекса от нижнего предела к верхнему.

Этими равенствами мы определили тексты F и G. Ниже мы докажем, что они могут быть порождены прямолинейными программами $\mathcal{F}$ и $\mathcal{G}$. Но сначала убедимся, что вложимость текста F в G равносильна существованию подмножества $\bar{w}$ с суммой t.

Равносильность существования подмножества с заданной суммой и вложимости.

"Существует подмножество $\Rightarrow$ вложимость". Пусть x — код подмножества $\bar{w}$ с суммой t, то есть $x \circ \bar{w} = t$. Рассмотрим начало G, то есть

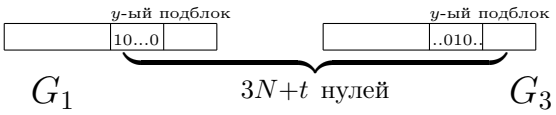
$G_1G_2G_3G_4G_1\ldots$, и выделим следующее вложение F_0. Отметим 1 в x-ом блоке G_1, затем $3N+t$ нулей, далее (в точности потому что $x \circ \bar{w} = t$) стоит 1 и, отметив следующие $N+1$ нулей, мы окажемся перед x-й единицей во втором G_1. Таким образом, имея $5N$ блоков $G_0 = G_1G_2G_3G_4$, мы сможем вложить $5N-1$ копий F_0. Что, собственно, и требовалось проверить: $F \hookrightarrow G$.



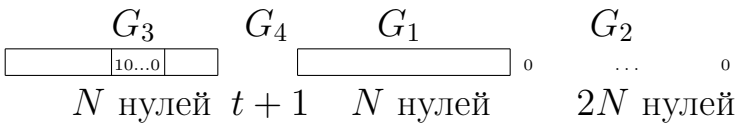
"Вложимость $\Rightarrow$ существует подмножество". Пусть $F \hookrightarrow G$. Предположим также, что ответ в *Сумме размеров* отрицательный. Выведем из этих двух утверждений противоречие.

Мы будем искать нули в G, которые не задействованы во вложении F в G. Это вложение представляет собой $5N-1$ непересекающихся вложений F_0 в G. Слово F_0 содержит две единицы, между которыми расположено ровно $3N+t$ нулей.

Убедимся, от противного, что в G нет двух единиц, между которыми расположено в точности столько же нулей. Рассмотрим два случая: первая единица находится в блоке G_1 (скажем, в подблоке номер y). Тогда, сдвинувшись на $3N+t$ нулей, мы попадем в y-ый подблок G_3. Если бы в этом подблоке после t нулей стояла единица, то мы бы получили $y \circ \bar{w} = t$ — противоречие с несуществованием подмножества с суммой t.



Второй случай: левая единица лежит в блоке G_3. Тогда, сдвинувшись на $3N + t$ нулей, мы попадем внутрь блока G_2, где вообще нет единиц.



Следовательно, при каждом вложении F_0 хотя бы один ноль (между двух единиц) остается незадействованным. Осталось лишь оценить с двух сторон количество нулей в G. По построению, их там $5N{\cdot}(4N{+}t{+}1)$. С другой стороны, каждое слово F_0 содержит $4N + t + 1$ нулей и еще хотя бы $5N - 1$ ноль не задействован. Таким образом, общее количество нулей в G должно быть не меньше, чем

$$(5N{-}1){\cdot}(4N{+}t{+}1){+}5N{-}1 = 5N{\cdot}(4N{+}t{+}1){+}(N{-}t{-}2) > 5N{\cdot}(4N{+}t{+}1).$$

Поясним последнее неравенство: $N = s2^n \geq 4s > t + 2$. То есть мы получили больше нулей в G, чем там есть по построению, что и дает нам противоречие.

Реализация F и G в виде прямолинейных программ. Заметим, что, за одним исключением, F и G строятся из 0 и 1 только с помощью полиномиального числа конкатенаций и возведения в степень (двоичная запись всех использованных степеней – полиномиального размера). Такие конструкции напрямую реализуются сжатыми словами полиномиального размера. Единственным нетривиальным блоком является G_3. Этот блок мы уже представили как *строку Лори* в предыдущем разделе, посвященном расстоянию Хэмминга. Конструкция сжатой версии G_3 полиномиального размера впервые была предложена Маркусом Лори в работе [33].

Полиномиальность сведения. Для завершения доказательства необходимо убедиться, что построение прямолинейных программ, вычисляющих F и G, полиномиально относительно размера входных данных задачи *Сумма размеров*. Чтобы убедиться в этом, достаточно заметить, что в построении F и G мы использовали лишь конкатенации и возведение в степени, являющиеся результатами арифметических действий над t и $w_1, \ldots, w_n$. □

Следствие 4.2.1. *Вложимость NP-трудна.*

4.2.2 coNP-трудность

Теорема 4.2.2. *Вложимость сводится (по Карпу) к Невложимости. Невложимость сводится к Вложимости.*

Доказательство. Мы будем доказывать следующее утверждение: существует полиномиальный алгоритм, позволяющий для любых сжатых слов F и G построить такие сжатые слова F_1 и G_1 (естественно, что в размер их сжатого представления может быть лишь полиномиально больше, чем у F и G), что

$$F \hookrightarrow G \Leftrightarrow F_1 \not\hookrightarrow G_1. \qquad (*)$$

В случае унарного алфавита обе задачи лежат в P. Будем считать, что в алфавите хотя бы 2 буквы. Заметим, что за полиномиальное время можно вычислить последнюю букву F и дописать в конец к G другую букву (получив G') и при этом $F \hookrightarrow G \Leftrightarrow F \hookrightarrow G'$. Так что, не умаляя общности, можно считать, что F и G заканчиваются на разные буквы.

Пусть $F = f_1 \dots f_k$, $G = g_1 \dots g_m$. Для каждой буквы алфавита a введем обозначение $X_a = (\Sigma/a)^{m+1}$, где (Σ/a) – конкатенация всех букв алфавита, кроме a, в любом порядке.

Конструкция:

$$F_1 = G = g_1 \dots g_m$$

$$G_1 = X_{f_1} f_1 \dots X_{f_k} f_k$$

Проверка свойства $(*)$: Мы можем представить G в следующем виде: $R_1 f_1 \dots R_l f_l R_{l+1}$, где R_i не содержит буквы f_i. Утверждение $F \hookrightarrow G$ равносильно выполнению равенства $l = k$ для нашего представления.

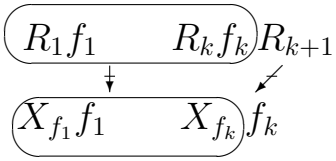
Если $l < k$, то $F_1 \hookrightarrow G_1$, так как для каждого $1 \le i \le l+1$ выполнено $R_i \hookrightarrow X_{f_i}$ и $f_i = f_i$.

$$\begin{array}{ll} R_1 f_1 R_2 X_2 & R_l f_l R_{l+1} \\ \downarrow \quad \downarrow & \downarrow \quad \downarrow \quad \downarrow \\ X_{f_1} f_1 & X_{f_l} f_l X_{f_{l+1}} \dots \end{array}$$

Если $l = k$, то R_{l+1} не пусто, так как g_m по предположению не равно f_k. По индукции можно убедиться в том, что

$$R_1 f_1 \dots R_i f_i \not\hookrightarrow X_{f_1} f_1 \dots X_{f_i}.$$

Действительно, при $i = 1$ это следует из факта $f_1 \not\hookrightarrow X_{f_1}$. Переход $i \to i+1$: если бы вложение существовало бы для $i+1$, то, так как $f_{k+1} \not\hookrightarrow X_{f_{k+1}}$, то f_{k+1} попадало бы не правее f_k, а значит было бы вложение и для i, что противоречит предположению индукции. Рассмотрим наше утверждение при $i = k$ и в сумме с фактом $R_{k+1} \not\hookrightarrow f_k$ мы получим $F_1 \not\hookrightarrow G_1$.



Полиномиальность конструкций: Заметим, что X_a строится за полиномиальное от $\log m$ время. Для построения G_1 остается лишь добавить правила вида $A \to X_a a$ для всего алфавита и использовать в конструкции эти нетерминальные символы вместо соответствующих терминалов. □

Следствие 4.2.2. *Вложимость coNP-трудна.*

Доказательство. Согласно Теореме 1 *Вложимость* NP-трудна. Следовательно, так как мы используем сведение по Карпу, *Невложимость* — coNP-трудна. Так как *Невложимость* сводится к *Вложимости* (Теорема 2), то *Вложимость* также является coNP-трудной. □

4.3 Выводы и открытые вопросы

Для поиска сжатой подпоследовательности удалось получить нижние оценки на вычислительную сложность. Мы построили доказательство (при предположении NP≠coNP), что эта задача лежит за пределами класса NP. С другой стороны, известна только тривиальная верхняя оценка сложности — легко построить алгоритм из класса PSPACE. Главным открытым вопросом остается сближение этих оценок.

Одной из причин интереса к поиску в сжатых текстах являются применения разрабатываемых алгоритмов к медиа-поиску. К сожалению, для поиска по изображениям, аудио и видео материалам грамматики являются, по всей видимости, неудачной моделью сжатых данных. Нам

известны теоремы о вычислительной трудности двумерного поиска подстрок [10]. В этой главе мы доказали, что поиск подпоследовательностей и даже вычисление расстояния Хэмминга (при предположении $\mathsf{P}\neq\mathsf{NP}$) не может быть вычислено за полиномиальное время. Таким образом, необходимо найти новую модель сжатых данных, которая бы допускала эффективный поиск приближенного вхождения шаблона.

Глава 5

Разреженная периодичность

Частично определенное слово — это слово в алфавите $\Sigma \cup \{\diamond\}$, где $\diamond$ — это специальная прозрачная (или неопределенная) буква. Другими словами, частично определенное слово представляет собой последовательность обычных слов (блоков), разделенных пропусками фиксированной (но необязательно одинаковой) длины.

Частично определенное слово S называется *разреженным периодом* (обычного) слова T, если мы можем разделить T на одну или несколько параллельно сдвинутых копий S, которые будут удовлетворять следующим условиям:

- Все определенные (видимые) буквы S-копий совпадают с соответствующими буквами в тексте;
- Каждая буква текста покрыта *в точности одной* определенной (видимой) буквой из S-копий.

Представим, что у нас есть несколько копий частично определенного слова, напечатанных на прозрачной бумаге. Тогда это слово будет разреженным периодом некоторого текста, если мы можем сложить наши копии в стопку так, чтобы образовалась одна связная строка без перекрытий видимых букв.

Пример: X$\diamond$Y является разреженным периодом для XXYY.

5.1 Примитивный разреженный период не единственен

Частичный порядок на частично определенных словах. Будем говорить, что одно частичное слово S *меньше* другого частичного слова Q, если Q может быть разделено на несколько параллельно сдвинутых копий S так, чтобы были выполнены следующие свойства:

- Все определенные (видимые) буквы S-копий совпадают с соответствующими буквами Q
- Каждая буква Q покрыта *в точности одной* определенной буквой из S-копий.

В частности, любой разреженный период меньше, чем сам текст. На самом деле, это определение просто расширяет разреженную периодичность на частично определенные слова.

Пример:

меньше чем

В классической периодичности для каждого текста существует единственный примитивный чистый период. Здесь под словом “примитивный” мы имеем в виду, что любой другой период является всего лишь последовательным повторением нескольких копий примитивного периода. Для обнаружения разреженной периодичности было бы крайне важно иметь такое же свойство.

Вопрос о примитивном разреженным периоде: верно ли, что для каждого обычного слова существует единственный примитивный разреженный период (то есть он меньше любого другого)? Переформулировка: верно ли, что любые два разреженных периода одного слова имеют общий разреженный “подпериод”?

Удивительно, но эта столь естественно выглядящая гипотеза неверна. Здесь мы представляем наименьший из известных нам (24 буквы) контрпримеров. Следующие разреженные периоды

и

несравнимы между собой. Дадим лишь краткий набросок объяснения, почему у них не может быть общего подпериода. Предположим, что он все же есть. Тогда это частично определенное слово имеет только одну букву B. Так как расстояние между первой и второй буквами B в тексте равно трем, сдвиг между первой и второй копией нашего подпериода тоже равен трем. Таким образом, это частично определенное слово начинается с трех подряд идущих A. Но с другой стороны, сдвиг третьей копии относительно второй равен единице. Следовательно, стартовые блоки второй и третьей копии перекрываются.

Как получить больше двух попарно несравнимых разреженных периодов? Мы можем использовать для этой цели "рекурсивную конструкцию". Пусть T — это текст вышеприведенного контрпримера, а тексты $T_{A_1B_1}$ и $T_{A_2B_2}$ получены из T просто использованием других букв. Построим текст T_2, заменяя каждую букву A в T на $T_{A_1B_1}$ и каждую B в T на $T_{A_2B_2}$. Теперь мы можем описать четыре несравнимых периода для T_2. Сам текст T_2 имеет длину 24^2. Сгруппируем его буквы на 24 блока по 24 буквы. Оставим лишь 12 блоков, используя как шаблон одно из двух вышеприведенных частично определенных слов. Теперь внутри каждого блока тоже оставим лишь 12 символов, используя один из двух разреженных периодов исходного контрпримера (одинаково в каждом блоке).

Мы можем повторять эту конструкцию несколько раз. Тогда текст T_k будет иметь длину 24^k и у него будет 2^k попарно несравнимых периодов. Асимптотически, эта конструкция дает нижнюю оценку $n^{\frac{\log 2}{\log 24}} > \sqrt[5]{n}$ на количество примитивных разреженных периодов.

В 2003 году в своем курсе лекций [22] Теро Харью поставил следующий вопрос. Предположим, что некоторая раскрашенная клетчатая фигура имеет разбиение на одинаковые копии одного шаблона и может быть разбита на копии другого шаблона. Верно ли, что всегда существует такой третий шаблон, что на его копии можно разбить и первый, и второй? Этот вопрос был мотивирован изучением defect effect в комбинаторике на словах [23]. Наш пример дает отрицательный ответ на вопрос Харью даже для одномерных (но необязательно связных!) фигур.

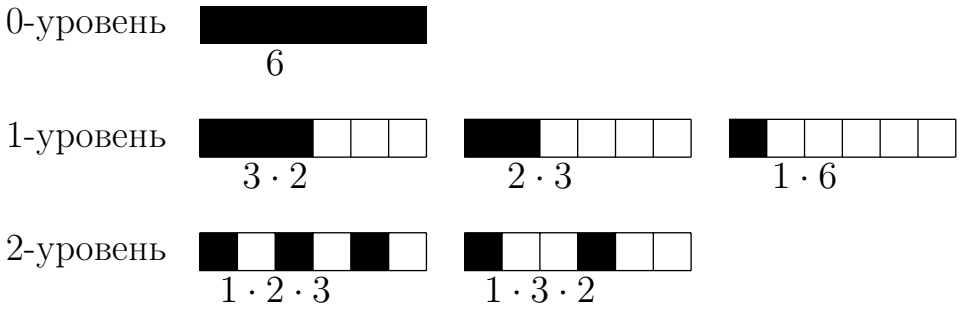
5.2 Свойства разреженной периодичности

Начнем наше исследование с классификации всех возможных разреженных периодов унарного (используется только одна буква) слова длины n. Так мы получим точную верхнюю оценку на количество всех разреженных периодов для текста длины n. Далее мы переформулируем разреженную периодичность в алгебраических терминах (аналогично правилу $s_i = s_{i+p}$ для классической периодичности). Пользуясь этой переформулировкой мы докажем, что каждый примитивный разреженный период меньше классического примитивного периода. Наконец, мы представим алгоритм поиска всех разреженных периодов минимального размера.

5.2.1 Количество разреженных периодов

Теорема 5.2.1. *Существует биективное соответствие между разреженными периодами унарного слова длины n и упорядоченными разложениями $n = n_1 \cdot \ldots \cdot n_k$, где $n_2, \ldots, n_k \geq 2$.*

Доказательство. Мы начнем с описания множества разреженных периодов (иерархическая конструкция) унарного слова длины n. После этого мы докажем полноту этого множества, а именно: любой разреженный период в него входит.



Мы разделяем множество всех разреженных периодов на уровни. Каждому периоду в нашей системе будет поставлен в соответствие специальный *код*. Весь текст будет единственным периодом 0-уровня и ему будет соответствовать код n. Для каждого разложения $n = n_1 \cdot n_2$, где $n_2 \geq 2$, блок из первых n_1 букв будет периодом 1-уровня с кодом $n_1 \cdot n_2$. На самом деле, 0-уровень и 1-уровень вместе составляют множество всех классических (то есть связных) периодов.

Объясним теперь, как, отталкиваясь от периода P из k-уровня и с кодом $n_1 \cdot \dots \cdot n_k$, построить новый разреженный период Q из $k+2$-уровня. Нужно взять разложение $n_1 = m_1 \cdot m_2 \cdot m_3$, где $m_2, m_3 \geq 2$. В течение всего процесса построения первое число в коде равно длине непрерывных блоков в периоде (в рамках наших построений все блоки в периоде имеют равную длину). Для построения нового разреженного периода Q мы возьмем каждый блок P, разделим его на m_3 групп по $m_1 \cdot m_2$ символов, и в каждой такой группе оставим только m_1. Заметим здесь, что P может быть покрыто m_2 копиями Q (со сдвигами $0, m_1, \dots, m_1 \cdot (m_2 - 1)$). Таким образом, Q также является разреженным периодом. Мы присвоим ему код $m_1 \cdot m_2 \cdot m_3 \cdot n_2 \cdot \ \dots \ \cdot n_k$. Подчеркнем, что наше построение сохраняет неравенство $n_2, \dots, n_k \geq 2$ для всех кодов.

Докажем теперь, что каждый разреженный период включен в наше построение. Возьмем произвольный разреженный период P. Все его блоки должны иметь одинаковую длину s, и длина каждого пропуска должна быть кратна s. В самом деле, в замощении текста вторая копия P сдвинута на длину первого блока s_1. Таким образом, чтобы избежать перекрытия, все другие блоки по длине не превосходят s_1. Предположим теперь, что какой-то блок b оказался первым из тех, чья длина все же

строго меньше чем s_1. Тогда длина пропуска между блоком b из первой копии P и b из второй копии P строго больше s_1 и строго меньше $2s_1$. Следовательно, этот пропуск не может быть полностью заполнен. Каждый пропуск в P заполнен несколькими связными блоками из других копий. Следовательно, все пропуски кратны $s = s_1$.

Нам потребуется определить *простую степень* для каждого разреженного периода P. Пусть s — это размер блока, а $g \cdot s$ — длина первого пропуска. Простой степенью P будет объединение $g+1$ копии P со сдвигами $0, s, 2s, \ldots, g \cdot s$.

Факт: Простая степень произвольного разреженного периода P также является разреженным периодом.

Доказательство: возьмем разбиение текста на P-копии. Пусть s — это размер блока, а $g \cdot s$ — длина первого пропуска в P. Разделим все копии P на группы по $g + 1$ экземпляров следующим образом. Будем рассматривать все копии слева направо. Пропуск между первыми двумя блоками первой копии P может быть заполнен *только первыми* блоками других P-копий. Первый экземпляр P и участники заполнения первого пропуска составят нашу первую группу. Предположим теперь, что у нас уже создано несколько групп. Рассмотрим самую левую из еще не использованных копий P. Посмотрим на ее первый пропуск. Все P-копии, включенные в уже созданные группы, идут чересчур длинными (не менее $s \cdot (g + 1)$) непрерывными участками. Таким образом, рассматриваемый пропуск тоже заполнен новыми, еще не использованными копиями. Так как мы обрабатываем копии строго слева направо, для заполнения нашего пропуска используются только первые блоки других копий. Следовательно, мы можем объединить их в очередную группу из

$g+1$ копии, идущей непрерывно одна за другой. Отметим, что каждая группа в точности является простой степенью P. Таким образом, мы доказали, что исходный текст имеет разбиение на копии простой степени. Утверждение доказано.

Если существует хоть один разреженный период P вне нашей конструкции, то будет и такой разреженный период P', что (1) сам он не включен в наше построение и (2) его простая степень Q входит в нашу иерархию. Выведем противоречие из этих посылок. В самом деле, пусть $n_1 \cdot \ldots \cdot n_k$ — код Q, длина блоков P' равна s, а $s \cdot g$ равно длине первого пропуска. Тогда размер блока n_1 в разреженном периоде Q кратен $s \cdot (g+1)$. Пусть $m_1 = s, m_2 = g+1, m_3 = n_1/(s \cdot (g+1))$. Остается заметить, что P' на самом деле является разреженным периодом нашей иерархии с кодом $m_1 \cdot m_2 \cdot m_3 \cdot n_2 \cdot \ldots \cdot n_k$.

Теорема доказана. □

Следствие. Пусть $L(n)$ — это количество разреженных периодов унарного слова длины n. Только что доказанная теорема утверждает, что $L(n)$ равно количеству разложений $n = n_1 \cdot \cdots \cdot n_k$, где $n_2, \ldots, n_k \geq 2$. Группируя все разложения по крайне правому сомножителю, мы получим следующую рекуррентную формулу:

$$L(1) = 1; \qquad L(n) = 1 + \sum_{d|n, d \neq n} L(d).$$

Замечание. Энциклопедия целочисленных последовательностей [46], поддерживаемая Нейлом Слоаном, содержит две интересные для нас последовательности. Последовательность A067824 определяется в точности формулой из Следствия 1. Последовательность A107736 выражает

количество многочленов p с коэффициентами из $\{0,1\}$, которые делят x^n-1, и таких, что коэффициенты частного при делении $(x^n-1)/((x-1)p(x))$ также принадлежат $\{0,1\}$. Но это число в точности равно количеству разреженных периодов унарного слова длины n. В самом деле, умножая p на многочлен с коэффициентами из $\{0,1\}$, мы проводим разбиение n "единиц" на несколько параллельно сдвинутых копий p без перекрытий. Таким образом, из Теоремы 1 и Следствия вытекает, что последовательности A067824 и A107736 совпадают. Действительно, количество многочленов равно количеству разреженных периодов, количество разреженных периодов равно числу разложений (Теорема 1), а число разложений удовлетворяет нужной рекурсивной формуле (Следствие 1).

Независимо от представленного здесь доказательства, за полгода до публикации работы о разреженной периодичности [3], в статье [15] было дано другое доказательство равенства A067824 и A107736. Сравнивая эти работы, отметим два достоинства доказательства, приведенного в этой диссертации. Во-первых, мы установили конструктивное соответствие между разложениями длины текста на множители и разреженными периодами. Во-вторых, мы внесли определенную структуру в множество всех периодов, разделив их на уровни.

Функция $L(n)$ и число разложений также появлялись в книге Дональда Кнута "Искусство программирования" [30]. Тем не менее, никакой замкнутой формулы для вычисления $L(n)$ пока не известно. Простыми вычислениями мы можем показать, что для всяких простых чисел p и q верны следующие равенства:

- $L(p) = 2$,

- $L(p^k) = 2^k$,
- $L(pq) = 6$,
- $L(p^2q) = 1 + L(1) + L(p) + L(q) + L(pq) + L(p^2) = 16$,
- $L(p^2q^2) = 52$, в частности $L(36) = 52$.

Как мы видим, слово длины 36 может иметь до 52 разреженных периодов. Следовательно, найдутся два разных периода с одинаковым "размахом" (расстоянием между первой и последней видимой буквой).

Теперь мы можем сказать больше и о примере во введении. Если d-мерный параллелепипед разбит на копии меньшего параллелепипеда, то последовательность цветов будет обладать разреженным периодом $2d$-уровня. Размер этого периода (число видимых букв) будет равно объему меньшего параллелепипеда.

5.2.2 Соотношение между разреженными и классическими периодами

Определение. Будем говорить, что слово $T = t_0 \dots t_{n-1}$ имеет *чистый ограниченный период* (a, b) (или период a с ограничением b), если $b|n$, $a|b$, $a < b$ и выполнено следующее утверждение:

$$\left\lfloor \frac{i}{b} \right\rfloor = \left\lfloor \frac{i+a}{b} \right\rfloor \Rightarrow t_i = t_{i+a}.$$

Другими словами, если мы разделим весь текст на блоки длины b, то каждый из блоков будет иметь чистый период a. Классический период p можно описать как ограниченный период (p, n).

A|A|B|B|A|A|B|B|C|C|D|D|C|C|D|D

Этот текст имеет ограниченные периоды $(1, 2)$ и $(4, 8)$.

Лемма 5.2.1. *У слова T есть разреженный период P с кодом $n_1 \cdot \ldots \cdot n_{2k}$ тогда и только тогда, когда текст обладает ограниченными периодами*

$$(n_1, n_1 n_2), \ldots, (\prod_{i=1}^{2k-1} n_i, \prod_{i=1}^{2k} n_i).$$

У слова T есть разреженный период P с кодом $n_1 \cdot \ldots \cdot n_{2k+1}$ тогда и только тогда, когда текст обладает ограниченными периодами

$$(n_1, n_1 n_2), \ldots, (\prod_{i=1}^{2k-1} n_i, \prod_{i=1}^{2k} n_i).$$

Доказательство. Шаг 1: от разреженного периода к ограниченным периодичностям. Будем использовать индукцию по уровню разреженного периода. Рассмотрим соответствующее разбиение текста. Напомним, что все копии P могут быть разделены на группы по n_2 экземпляров, со сдвигами внутри группы $0, n_1, \ldots, n_1(n_2 - 1)$. Таким образом, если мы разделим весь текст на блоки по $n_1 n_2$ букв, каждый блок будет покрыт P-копиями из одной группы. Следовательно, внутри каждый блок будет n_1-периодическим. Таким образом, мы доказали $(n_1, n_1 n_2)$-ограниченную периодичность. Все остальные ограниченные периодичности следуют из индукционного предположения для простой степени P.

Шаг 2: от ограниченных периодичностей к разреженному периоду. Рассмотрим *верхнюю* ограниченную периодичность $(\prod_{i=1}^{2k-1} n_i, \prod_{i=1}^{2k} n_i)$. Возьмем частичное слово Q, соответствующее коду $(\prod_{i=1}^{2k-1} n_i) \cdot n_{2k}$ для четного варианта леммы и соответствующее коду $(\prod_{i=1}^{2k-1} n_i) \cdot n_{2k} \cdot n_{2k+1}$

для нечетного варианта. Напрямую из верхней ограниченной периодичности следует, что каждая копия Q на одинаковых позициях имеет одинаковые буквы. Следовательно, Q является разреженным периодом текста. Продолжая рассуждение, с помощью второй ограниченной периодичности можно построить следующий разреженный период R внутри Q. В конце концов, из последней ограниченной периодичности мы сможем вывести существование разреженного периода с требуемым кодом. □

Лемма 5.2.2. *Пусть у текста T есть классический период p и ограниченный период (a, b), тогда или $b|p$ или у текста есть также классический период НОД(a, p).*

Доказательство. Возьмем произвольную букву текста T_i. Мы будем стараться сделать несколько прыжков величиной $\pm p$ и $+a$, чтобы в итоге достичь позиции $i + \text{НОД}(a, p)$. Нам хочется все время находиться на позициях с одинаковыми буквами. Поэтому мы запретим себе делать прыжок $+a$ из последнего a-блока в каждом b-блоке. Согласно свойствам расширенного алгоритма Евклида, существуют такие натуральные числа k и l, что $\text{НОД}(a, p) = ka - lp$. Мы будем использовать следующую *жадную* стратегию. Если мы имеем право сделать прыжок $+a$, мы его делаем. В противном случае, попытаемся сделать несколько прыжков $\pm p$. После того, как мы сделаем в точности k прыжков $+a$, нам останется лишь провести все недостающие передвижения $\pm p$, чтобы очутиться на желаемой позиции. Единственной преградой на пути жадной стратегии может стать ситуация, когда для некоторого $j < p$ для каждой из позиций $j, j + p, \ldots j + (\frac{n}{p} - 1)p$ нам запрещен прыжок $+a$. То есть слу-

чай, когда все эти позиции попали в последние a-блоки в своих b-блоках. Предположим теперь, что p не делится на b. Тогда $u = \text{НОД}(p, b) \leq \frac{b}{2}$. Так как $p|n$ и $b|n$, среди чисел $j \bmod b, \ldots, [j + (\frac{n}{p} - 1)p] \bmod b$ встречаются все такие остатки h по модулю b, что $h \equiv j (\text{mod } u)$. Следовательно, один из них не больше u, что в свою очередь не превосходит $b/2$. Но это значит, что соответствующая позиция никак не могла попасть в последний a-блок своего b-блока. □

Теорема 5.2.2. *Любой примитивный разреженный период Q слова T является также разреженным периодом для классического примитивного периода T.*

Доказательство. Воспользуемся индукцией по длине текста. Для однобуквенных слов теорема верна. Пусть p — длина классического периода, а (a, b) — верхняя ограниченная периодичность (полученная из кода Q по Лемме 1). Если $p = n$, теорема верна, поэтому будем считать, что $p < n$. Согласно иерархической конструкции, все видимые буквы частично определенного слова Q находятся в первых a-блоках своих b-блоков. Применим Лемму 2. Если $b|p$, то мы можем получить новый разреженный период Q', оставив от Q только ту часть, которая входит в первые p символов текста. Согласно p-периодичности, Q может быть разбито на несколько копий Q' со сдвигами $0, p, 2p, \ldots, (\frac{n}{p} - 1)p$. Мы получили противоречие с примитивностью Q.

Таким образом, верно второе утверждение из Леммы 2: текст обладает классическим НОД(a, p)-периодом. Но по условию теоремы, p является примитивным классическим периодом. Следовательно, $p|a$. А

раз текст является p-периодическим, то он также и a-периодичен и b-периодичен. На этот раз мы можем оставить от Q только буквы, входящие в первый a-блок текста. Этим мы или получим меньший разреженный период Q' (противоречие с примитивностью Q), или $b = n$ и все видимые буквы Q расположены в первом a-блоке текста. В последнем случае, и классический период p, и разреженный Q являются периодами первого a-блока. Так как $a \leq b/2 \leq n/2$, мы можем применить индукционное предположение. □

Следствие 2. Для каждого разреженного периода Q и классического периода p найдется общий разреженный подпериод.

Доказательство. Возьмем примитивный разреженный период Q', который меньше (то есть разбивает) Q. Рассмотрим классический примитивный период p'. Из единственности чистого классического примитивного периода следует $p'|p$. По Теореме 2 разреженный период Q' является также разреженным периодом для первого p'-блока текста. Таким образом, Q' является искомым разреженным подпериодом для Q и p. □

Замечание. Используя технику из Леммы 1, Леммы 2 и Теоремы 2, можно показать, что примитивный разреженный период единственен в случае $n = 2^k$. От противного: возьмем два несравнимых примитивных разреженных периода. Рассмотрим их верхние ограниченные периодичности. Применим рассуждения, подобные Лемме 2. Либо один из этих периодов не примитивен, либо эти верхние ограниченные периодичности совпадают. Двигаясь дальше вниз, мы либо докажем их равенство, либо получим противоречие с примитивностью.

5.2.3 Алгоритм поиска разреженных периодов минимального размера

Определим размер разреженного периода как число его видимых букв. Как найти все разреженные периоды минимального размера? Будем пользоваться Леммой 1 для ответа на этот вопрос. Мы знаем, что каждому разреженному периоду соответствует цепочка *вложенных* ограниченных периодичностей $(a_1, b_1), \ldots, (a_k, b_k)$. Под "вложенностью" мы понимаем делимость $b_i | a_{i+1}$ для каждого i. Заметим, что размер такого разреженного периода равен произведению $n \prod_{i=1}^{k} \frac{a_i}{b_i}$.

Алгоритм поиска разреженных периодов минимального размера:

1. Для каждой пары $a < b$, такой что $a|b$ и $b|n$, проверить, обладает ли текст (a, b)-ограниченной периодичностью.
2. Построить ациклический граф из найденных ограниченных периодичностей. Пары (a, b) будут вершинами, и мы будем проводить ребро от (a, b) к (c, d), если $b|c$.
3. В каждую вершину (a, b) записать значение a/b.
4. Найти все пути в графе с наименьшим произведением значений в вершинах.

Анализ алгоритма. Первый шаг требует $O(d^2(n)n)$ времени при переборном подходе. Здесь $d(n)$ обозначает количество делителей n. Четвертый шаг лишь незначительно отличается от поиска кратчайших путей в ациклических графах. Сначала нужно совершить один проход по всему графу сверху (маленький период, маленькое ограничение) вниз (большой период, большое ограничение). При проходе мы будем стирать все

ребра, которые не участвуют в получении минимальных произведений по путям. В итоге мы получим несколько *стоков* (вершин без исходящих ребер), и оставим только тот (или те) из них, которые дают минимальное произведение по пути, и удалим все остальные стоки и ребра ведущие к ним. После этого нужно пройти весь граф обратно снизу вверх и стереть все ребра, не участвующие в минимальных путях. В оставшемся графе каждый путь из некоторого истока (вершины без входящих ребер) к некоторому стоку соответствует разреженному периоду минимального размера. Таким образом, мы получили сжатое представление всех разреженных периодов минимального размера. Четвертый шаг алгоритма требует лишь линейное время относительно размера графа. Следовательно время работы алгоритма составляет $O(d^2(n)n + d^2(n)) = n^{1+o(1)}$.

5.3 Выводы и открытые вопросы

Мы можем предложить ряд естественных, важных и, возможно, не столь сложных вопросов для дальнейшего исследования разреженных периодов. Приведем список этих задач.

1. Изучить *не обязательно чистую* разреженную периодичность.
2. Найти точную асимптотическую верхнюю оценку $L(n)$.
3. Построить более эффективный (линейный?) алгоритм для нахождения разреженного периода минимального размера.
4. Вычислить, как часто случайное слово (в разных моделях) обладает собственным разреженным периодом. Сравнить полученные результаты с классическим случаем.

5. Проверить, можно ли выразить свойство *"слово имеет разреженный квадратный корень"* через уравнения в словах. Мы отсылаем читателя к работе [25] для знакомства с исследованиями по выразительной силе уравнений в словах.

6. Проверить, верно ли, что все примитивные периоды имеют одинаковое число видимых букв.

7. Найти другие естественные источники разреженной периодичности.

8. Определить и изучить разреженную периодичность на деревьях и других объектах.

9. Определить и изучить приближенную разреженную периодичность (допускающую небольшие ошибки).

10. Определить и изучить разбиения на копии двух (или более) частичных слов. Этот тип разбиений может оказаться еще более подходящим для применений в архивировании.

На наш вкус, первый вопрос является наиболее важным, так как такое расширение понятия разреженной периодичности позволит описать намного больше структурных свойств текстов. Кроме решения вышеприведенных вопросов, необходимо найти близкие результаты и техники в комбинаторике. Такое естественное понятие, как разреженная периодичность, просто обязано иметь предшественников в дискретной математике.

Литература

[1] **Гасфилд Д.** Строки, деревья и последовательности в алгоритмах: информатика и вычислительная биология // Пер. с англ.— BHV, Санкт-Петербург, 2003.

[2] **Гэри М., Джонсон Д.** Вычислительные машины и труднорешаемые задачи // Пер. с англ.— Москва, Мир, 1982.

[3] **Карьюмяки Ю., Лифшиц Ю.М.** Разреженная периодичность // Препринт ПОМИ 22/06, 2006.

[4] **Лифшиц Ю.М.** Алгоритмические свойства сжатых текстов // Препринт ПОМИ 23/06, 2006.

[5] **Лифшиц Ю.М.** Обработка сжатых текстов // Материалы XVI Международной школы-семинара "Синтез и сложность управляющих систем", стр. 64–68, Изд-во механико-математического факультета МГУ, 2006.

[6] **Притыкин Ю.Л.** Конечно-автоматные преобразования строго почти периодических последовательностей // Математические заметки, 2006, 80:5, 751-–756.

[7] **Шур А.М., Гамзова Ю.В.** Частичные слова и свойство взаимодействия периодов // Известия РАН. Серия математическая, 2004, 68:2, 191–214.

[8] **Amir A., Benson G., Farach M.** Let sleeping files lie: Pattern matching in Z-compressed files // *SODA'94*, 1994.

[9] **Apostolico A., Farach M., Iliopoulos C.S.** Optimal superprimitivity testing for strings // *Information Processing Letters*, 39(1):17–20, 1991.

[10] **Berman P., Karpinski M., Larmore L., Plandowski W., and Rytter W.** On the complexity of pattern matching for highly compressed two-dimensional texts // *Journal of Computer and Systems Science*, 65(2):332–350, 2002.

[11] **Blanchet-Sadri F.** Periodicity on partial words // *Computers and Mathematics with Applications*, 47(1):71–82, 2004.

[12] **Blanchet-Sadri F.** Primitive partial words // *Discrete Applied Mathematics*, 148:195–213, 2005.

[13] **Blanchet-Sadri F., Hegstrom R.** Partial words and a theorem of Fine and Wilf revisited // *Theoretical Computer Science*, 270(1/2):401–419, 2002.

[14] **Boasson L., Berstel J.** Partial words and a theorem of Fine and Wilf // *Theoretical Computer Science*, 218(1):135–141, 1999.

[15] **Bodini O., Rivals E.** Tiling an interval of the discrete line // *CPM'06*, LNCS 4009, pages 117–128, Springer-Verlag, 2006.

[16] **Cégielski P., Guessarian I., Lifshits Y., and Matiyasevich Y.** Window subsequence problems for compressed texts // *CSR'06*, LNCS 3967, pages 127–136, Springer-Verlag, 2006.

[17] **Constantinescu S., Ilie L.** Generalised Fine and Wilf's theorem for arbitrary number of periods // *Theoretical Computer Science*, 339(1):49–60, 2005.

[18] **Farach M., Thorup M.** String matching in Lempel-Ziv compressed strings // *STOC '95*, pages 703–712, ACM Press, 1995.

[19] **Fine N., Wilf H.** Uniqueness theorems for periodic functions // *Proc. Amer. Math. Soc.*, 16:109–114, 1965.

[20] **Gąsieniec L., Karpinski M., Plandowski W., and Rytter W.** Efficient algorithms for Lempel-Ziv encoding (extended abstract) // *SWAT'96*, LNCS 1097, pages 392–403, Springer-Verlag, 1996.

[21] **Genest B., Muscholl A.** Pattern matching and membership for hierarchical message sequence charts // *LATIN'02*, LNCS 2286, pages 326–340, Springer-Verlag, 2002.

[22] **Harju T.** Defect theorem // Lecture notes of "Combinatorics of words" Tarragona course, 2002/2003.

[23] **Harju T., Karhumäki J.** Many aspects of defect theorems // *Theoretical Computer Science*, 324(1):35–54, 2004.

[24] **Hirao M., Shinohara A., Takeda M., and Arikawa S.** Fully compressed pattern matching algorithm for balanced straight-line programs // *SPIRE'00*, pages 132–138, IEEE Computer Society, 2000.

[25] **Karhumäki J., Mignosi F., and Plandowski W.** The expressibility of languages and relations by word equations // *ICALP'97*, number 1256 in LNCS, pages 98–109, Springer-Verlag, 1997.

[26] **Kärkkäinen J., Navarro G., and Ukkonen E.** Approximate string matching over Ziv-Lempel compressed text //*CPM'00*, LNCS 1848, pages 195–209, Springer-Verlag, 2000.

[27] **Karpinski M., Rytter W., and Shinohara A.** Pattern-matching for strings with short descriptions // *CPM'95*, LNCS 937, pages 205–214, Springer-Verlag, 1995.

[28] **Katona G., Szász D.** Matching problems // *J. of Combinatorial Theory Ser B*, 10(1):60–92, 1971.

[29] **Kida T., Matsumoto T., Shibata Y., Takeda M., Shinohara A., and Arikawa S.** Collage system: a unifying framework for compressed pattern matching // *Theoretical Computer Science*, 298(1):253–272, 2003.

[30] **Knuth D.** The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions / D. Knuth // Addison-Wesley, 2005.

[31] **Lasota S., Rytter W.** Faster algorithm for bisimulation equivalence of normed context-free processes // *MFCS'06*, LNCS 4162, pages 646–657, Springer-Verlag, 2006.

[32] **Lifshits Y., Lohrey M.** Querying and embedding compressed texts // *MFCS'06*, LNCS 4162, pages 681–692, Springer-Verlag, 2006.

[33] **Lohrey M.** Word problems on compressed word // *ICALP'04*, LNCS 3142, pages 906–918, Springer-Verlag, 2004.

[34] **Markey N., Schnoebelen P.** A PTIME-complete matching problem for SLP-compressed words // *Information Processing Letters*, 90(1):3–6, 2004.

[35] **Miyazaki M., Shinohara A., and Takeda M.** An improved pattern matching algorithm for strings in terms of straight line programs // *CPM '97*, LNCS 1264, pages 1–11, Springer-Verlag, 1997.

[36] **Navarro G.** Regular expression searching on compressed text // *J. of Discrete Algorithms*, 1(5-6):423–443, 2003.

[37] **Navarro G., Raffinot M.** A general practical approach to pattern matching over Ziv-Lempel compressed text // *CPM'99*, LNCS 1645, pages 14–36, Springer-Verlag, 1999.

[38] **Pagourtzis A., Zachos S.** The complexity of counting functions with easy decision version // *MFCS'06*, LNCS 4162, pages 741–752, Springer-Verlag, 2006.

[39] **Plandowski W.** Testing equivalence of morphisms on context-free languages // *ESA'94*, LNCS 855, pages 460–470, Springer-Verlag, 1994.

[40] **Plandowski W.** Satisfiability of word equations with constants is in PSPACE // *J. ACM*, 51(3):483–496, 2004.

[41] **Rytter W.** Compressed and fully compressed pattern matching in one and two dimensions // *Proceedings of the IEEE*, 88(11):1769–1778, 2000.

[42] **Rytter W.** Application of Lempel-Ziv factorization to the approximation of grammar-based compression // *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[43] **Rytter W.** Grammar compression, LZ-encodings, and string algorithms with implicit input // *ICALP'04*, LNCS 3142, pages 15–27, Springer-Verlag, 2004.

[44] **Shur A., Konovalova Y.** On the periods of partial words // *MFCS'01*, LNCS 2136, pages 657–665, Springer-Verlag, 2001.

[45] **Simpson R., Tijdeman R.** Multi-dimensional versions of a theorem of Fine and Wilf and a formula of Sylvester // *Proc. Amer. Math. Soc.*, 131:1661–1667, 2003.

[46] **Sloane N.J.A.** Sequence A067824 from The On-Line Encyclopedia of Integer Sequences // [Электронный ресурс] `http://www.research.att.com/~njas/sequences/A067824`.

[47] **Ziv J., Lempel A.** A universal algorithm for sequential data compression // *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.