ПРЕПРИНТ ПОМИ — 23/2006

АЛГОРИТМИЧЕСКИЕ СВОЙСТВА СЖАТЫХ ТЕКСТОВ *

Юрий Лифшиц
Петербургское отделение Математического института
им. В.А. Стеклова Российской академии наук
http://logic.pdmi.ras.ru/~yura/
yura@logic.pdmi.ras.ru

16 декабря 2006 г.

Аннотация

Можно ли эффективно (не проводя полную распаковку) изучать различные свойства сжатых текстов? В работе мы рассматриваем три фундаментальных задачи: (1) проверить равенство двух сжатых текстов, (2) определить, является ли один сжатый текст подстрокой другого и (3) найти количество несовпадающих символов (расстояние по Хэммингу) у двух сжатых текстов одинаковой длины.

В работе построен алгоритм, решающий задачу о равенстве за $O(n^3)$, а задачу поиска подстрок — за $O(n^2m)$ шагов. Здесь n — размер архива текста, m — размер архива шаблона. Мы также доказали, что очень близкая по формулировке задача (3) оказалась #P-полной. Алгоритмическая техника, которую мы применили для задач (1) и (2), также позволяет вычислять комбинаторные свойства сжатых текстов, такие как длина минимального периода и накрытия.

Завершает работу список открытых проблем по алгоритмам на сжатых текстах, их применению к мультимедиа-поиску, архивированию структур данных и верификации программ.

*Работа поддержана грантами НШ-8464.2006.1 и INTAS-04-77-7173

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, run-length encoding и т.д.). Можно ли объединить достоинства этих двух подходов? Начиная с середины девяностых годов ставится вопрос: можно ли вести быстрый поиск (или вычисление различных свойств) по сжатому тексту без полной распаковки?

История области. Исследования начались с построения алгоритмов для поиска явно заданных подстрок в сжатых текстах [1, 5]. 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 [24]. Мы подробно опишем прямолинейные программы в Разделе 2. Недавно было предложено некоторое обобщение прямолинейных программ под названием collage systems [12]. Для прямолинейных программ вслед за поиском подстрок, были представлены алгоритмы для поиска регулярных выражений, приближенного поиска подстрок, и поиска явно заданных подпоследовательностей (соответственно, [18], [11], [4]).

После построения ряда алгоритмов, имеющих полиномиальную сложность относительно *размера сжатых текстов*, исследования столкнулись с NP-трудными задачами: принадлежность сжатого текста контекстно-свободной грамматике [15], двумерный поиск подстрок [3], вложимость сжатого шаблона “как подпоследовательности” в сжатый текст [14]. Таким образом, представление текстов в виде прямолинейных программ позволяет решить далеко не все задачи без порождения исходного текста.

Неожиданно оказалось, что алгоритмы на сжатых текстах имеют прямое отношение к многим другим задачам теоретической информатики. Так, с их помощью удалось построить полиномиальный по памяти алгоритм решения уравнений в словах [22]; более быстрый алгоритм проверки эквивалентности программ (естественно, в ограниченном классе) [13]; полиномиальный алгоритм верификации диаграмм сообщений [8].

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 приложений является верификация программ и микросхем. Обычно для верификации нужно 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 проверить его свойства без явного порождения.

Экспериментальный анализ алгоритмов на сжатых текстах можно найти в статье [19], работы [23, 25] дают обзор всех алгоритмических исследований по сжатым текстам, книга [9] служит прекрасным введением в текстовые алгоритмы.

Постановка решаемых задач. Одной из центральных задач области является разработка алгоритмов сравнения двух сжатых текстов. Активно изучаются следующие вопросы:

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

2. Даны два сжатых текста. Определить, входит ли первый из них во второй, и если да, то найти место первого вхождения и общее число вхождений.

Добавим к этим задачам естественное продолжение:

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

Первая задача является частным случаем второй. Равенство сжатых текстов может использоваться для сверки разных архивных копий одной и той же системы, поиск подстрок — для верификации программ и поиска по медиа-данным (в них шаблон тоже может быть большим и требовать архивирования). Вычисление расстояния Хэмминга является самой простой формой приближенного поиска подстрок, который полезен для анализа биологических данных и, опять же, медиа-файлов.

Задача о равенстве впервые была решена в работе [21] в 1994 году за $O(n^4)$, алгоритм для поиска сжатой подстроки в сжатом тексте впервые появился в 1995 году в статье [16]. Вслед за этим удалось расширить алгоритм для вычисления различных комбинаторных свойств текста [7]. Наконец в 1997 году Миязаки, Шинохара и Такеда [17] построили алгоритм, требующий $O(n^2m^2)$ времени для поиска подстрок, где m и n — размеры сжатого шаблона и сжатого текста, соответственно. В 2000 еще удалось для одного специального случая провести поиск подстрок за $O(n^3)$.

Несмотря на естественность вопроса, полиномиальный алгоритм для вычисления расстояния Хэмминга между сжатыми текстами до сих пор не был найден.

Результаты. В Разделе 3 этой работы нам удалось улучшить алгоритмы [10, 17, 16, 21, 7] и решить задачу о равенстве за $O(n^3)$, а задачу о поиске подстрок за $O(n^2m)$ шагов. Здесь n — размер архива текста, m — размер архива шаблона.

Большим сюрпризом стала теорема из Раздела 4: вычисление расстояния Хэмминга для сжатых текстов является #P-полной задачей. Напомним, что #P — это класс функций, своеобразное расширение класса предикатов NP).

Алгоритмическая техника, которую мы применили для задачи о поиске подстроки, также позволяет вычислять комбинаторные свойства сжатых текстов, такие как длина минимального периода и накрытия. Это приложение и ряд открытых вопросов сформулированы в Разделе 5. В частности, одной из самых заманчивых задач для будущих исследований является построение *сжатого суффиксного дерева.*

2 Грамматика как модель сжатого текста

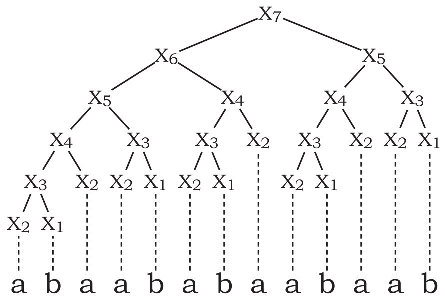
Мы рассматриваем конечный алфавит Σ, строкой (текстом) называется любая конечная последовательность его букв. В этой работе под сжатыми текстами мы понимаем строки, порожденные *прямолинейными программами* (ПП). Неформально, прямолинейная программа – это контекстно-свободная грамматика, порождающая только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

$$X_1 \to b,\;\; X_2 \to a,\;\; X_3 \to X_2X_1,\;\; X_4 \to X_3X_2,\;\; X_5 \to X_4X_3,\;\; X_6 \to X_5X_4,\;\; X_7 \to X_6X_5$$

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 замечательной работе Риттера [24] 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, мы не можем себе позволить восстанавливать исходный текст.

3 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 и вычислить "сжатое представление" всех вхождений. Мы собираемся решать задачу в последней постановке, то есть построить такую структуру данных, по которой можно быстро ответить на все остальные 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 решить все разновидности 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.

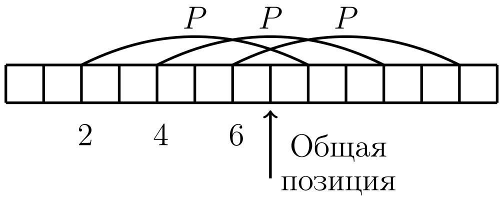
3.1 Поиск подстрок с помощью таблицы прогрессий

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.

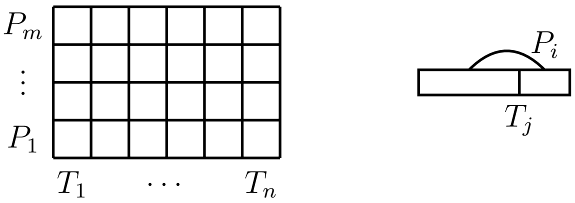
Пусть $P_1, \dots, P_m$ и $T_1, \dots, T_n$ — это нетерминальные символы прямолинейных программ, порождающих P и T, соответственно. Для каждого из этих промежуточных текстов мы определим специальную позицию *разреза.* В текстах, образованных по правилу $X_i = X_r X_s$, позицией разреза будет как раз точка, где заканчивается X_r и начинается X_s. Для полноты терминологии, будем считать позицию 0 разрезом в однобуквенных текстах. Для вышеприведенного примера, 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 [17]):

Лемма 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 определим $TP[i, j] = \varnothing_1$, а если текст достаточно велик, но не содержит ни одного вхождения, касающегося точки разреза, установим $TP[i, j] = \varnothing_2$.



Факт 1: Показать, что используя таблицу прогрессий (на самом деле, достаточно ее верхней строки) мы можем решить все варианты задачи поиска подстрок в сжатых текстах за $O(n)$ шагов.

Факт 2: Показать, что таблицу прогрессий можно заполнить с помощью метода динамического программирования за $O(n^2 m)$ шагов.

Алгоритм для Факта 1. 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. Если да, нужно посмотреть в TP-таблицу и проверить принадлежность соответствующей арифметической прогрессии. В противном случае нужно рекурсивно перейти к левой или правой части 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.

3.2 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 бо́льшей компоненты шаблона P_i вокруг разреза текста T_j;
   (b) Находим вхождения меньшей компоненты шаблона P_i, которые стыкуются уже найденных вхождений 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 можем составить массивы длин, позиций разрезов, первых и последних букв для текстов $P_1, \ldots, P_m, T_1, \ldots, T_n$ за время $O(n+m)$.

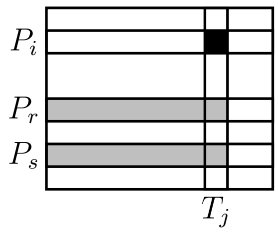
Шаг 2: заполняем "однобуквенные" строки/столбцы таблицы прогрессий. Пусть для какого-то i мы знаем, что $|P_i| = 1$. Тогда мы заполним i-ую строку таблицы по следующему принципу. Для каждого j мы сравним букву P_i либо с единственной буквой T_j (в случае $|T_j| = 1$), либо с последней буквой левой части 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 0, в противном случае сравниваем две буквы. При удаче — записываем в ответ прогрессию из одной позиции. Тут также достаточно константного времени на каждый элемент.

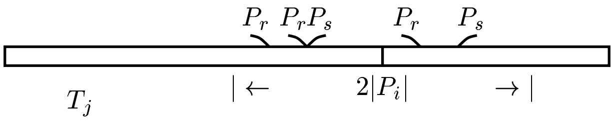
Шаг 3: вычисляем новый элемент таблицы прогрессий по "правилу пересечения". После "однобуквенных рядов" мы заполняем таблицу в алфавитном порядке пар (i,j). Пусть $P_i = P_r P_s$. Для вычисления $TP[i,j]$ мы будем использовать уже вычисленные значения $TP[r,1], \ldots, TP[r,j]$ и $TP[s,1], \ldots, TP[s,j]$. 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. [17] 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 $P_i = P_rP_s$, 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.*

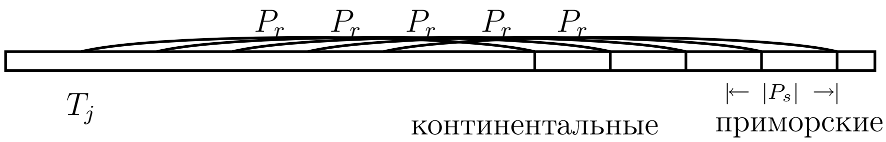
Для наших вычислений мы разработали вспомогательную процедуру *локального поиска*, которая извлекает нужную информацию из уже сосчитанных элементов таблицы прогрессий. Процедура $\text{Local}(i,j,[\alpha,\beta])$ возвращает вхождения P_i в T_j лежащие целиком внутри интервала $[\alpha,\beta]$. Важные свойства локального поиска:

- локальный поиск использует значения $TP[i,k]$ для $1 \leq k \leq j$,
- он корректно работает только при условии $|\beta - \alpha| \leq 3|P_i|$,
- его трудоемкость $O(j)$ шагов,
- 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. Возьмем или все континентальные окончания, или ни одно из них (в зависимости от результатов проверки). Возьмем подпрогрессию приморских окончаний (тех, которые соответствуют началам вхождений P_s), 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 время $O(1)$.

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 участков 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.

3.3 Процедура локального поиска

Локальный поиск: поиск вхождений P_i в подстроке $T_j[\alpha, \beta]$.

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
(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.

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 из не более чем $3j$ прогрессий. Здесь отсортированный означает, что последний элемент k-ой прогрессии находится левее или равен первому элементу $(k+1)$-ой прогрессии. Как раз 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 равных части. Если мы применим Основную лемму к позициям $\delta_1 = \frac{2\alpha+\beta}{3}$ и $\delta_2 = \frac{\alpha+2\beta}{3}$, мы узнаем, что все вхождения P_i в $[\alpha, \beta]$ образуют две (одна после другой) арифметические 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.

3.4 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) [12] и LZ77 [26] используют конкатенацию и *обрезку*. Конечно, как показал Риттер [24], 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. Мы планируем подробнее осветить этот вопрос в журнальной версии.

Мы также утверждаем, что по таблице прогрессий можно построить прямолинейную программу размера $O(n^2)$, описывающую все вхождения P в T.

4 Расстояние по Хэммингу между сжатыми текстами

Расстоянием Хэмминга называется количество несовпадающих букв на одинаковых позициях в текстах. Задача *расстояние 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 #P, если существует такая недетерминированная полиномиальная машина Тьюринга M, что значением функции будет число прижимающих веток M на соответствующих входных данных. Иначе говоря, существует полиномиально-вычислимая функция $G(x, y)$, такая что $f(x) = \#\{y | G(x, y) = \text{“да”}\}$. 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 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: $G(T, S; y) = \text{“да”}$, 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 другою полную задачу в рассматриваемом классе. Напомним формулировку #P-полной задачи *сумма размеров* [6]: даны натуральные числа $w_1, \ldots, w_n, t$ в двоичной записи. Требуется определить, сколько существует наборов $x_1, \ldots, x_n \in \{0, 1\}$, таких что $\sum_{i=1}^{n} x_i \cdot w_i = t$. Иначе говоря, сколько подмножеств $W = \{w_1, \ldots, w_n\}$ 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 $s+1$ символ 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 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 $\bar{x} \cdot \bar{w} = \sum x_i w_i$, а $\prod$ обозначает конкатенацию.

Строка S (будем называть ее *строка Лори*) впервые появилась в работе Маркуса Лори [15], затем использовалась и в работе [14]. Лори доказал [15], 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 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. Так, недавно [20] был введен класс TotP. Задача принадлежит классу TotP, если для нее 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 полиномиально разрешима, а считающий вариант — #P-полон.

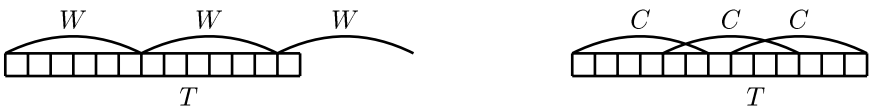
Можно показать, что расстояние Хэмминга между сжатыми текстами принадлежит классу TotP. В самом деле, можно сравнивать интервалы исходных текстов и разветвлять вычисления всякий раз, когда обе половины интервала из первого текста не равны аналогичным участкам второго.

5 Следствия и выводы

5.1 Применение таблицы прогрессий: вычисление 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 может помочь вычислить периоды и накрытия сжатого текста за полиномиальное (относительно сжатого размера) время. Здесь мы лишь напомним ключевые свойства строк и дадим наброски алгоритмов. Полное описание появится в журнальной версии статьи.

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 впервые введен в работе [2]) 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 следующих комбинаторных свойствах текстов:

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

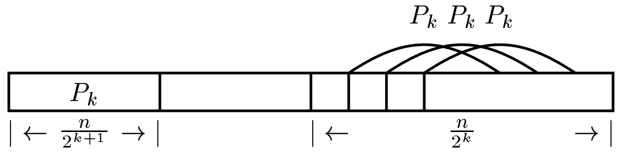
2. Каждое накрытие является *границей (border)*, т.е. одновременно суффиксом и префиксом текста. Если какая-то граница с длиной в интервале $[\frac{t}{2^{k+1}}, \frac{t}{2^k}]$ является накрытием, то все меньшие границы в этом интервале также являются накрытиями.

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 $T[\alpha, \beta]$.

Набросок алгоритма вычисления периодов сжатого текста. 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.

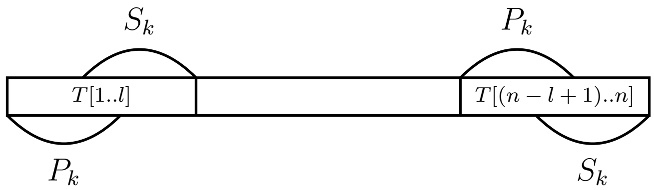


Шаг 2: возьмем $\frac{t}{2^{k+1}}$-суффикс текста T (обозначим его S_k) и найдем все его вхождения в $\frac{t}{2^k}$-префикс T. По Основной лемме эти вхождения образуют одну арифметическую прогрессию. Вторым списком кандидатов будут расстояния от начала текста T до концов вхождений S_k.

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, общая сложность алгоритма по всем интервалам составляет $O(n^3 \log |T|)$.

Задача о нахождении периода сжатого текста была поставлена в 1996 в расширенных тезисах [7]. К сожалению, полное описание алгоритма из этой работы (он требовал

$O(n^5 \log^3 |T|)$ шагов) так и не было опубликовано.

Набросок алгоритма 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. Уважаемся, 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, общая сложность составляет $O(n^3 \log^2 |T|)$ шагов.

5.2 Открытые проблемы и направления дальнейших исследований

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 $O(n^2 m)$. 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?

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 к 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. В то же время, 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 $A \cup B$?

Применения к медиа-поиску. Возможно, для поиска по изображениям, аудио и видео материалам грамматики являются неудачной моделью сжатых данных. Нам известны теоремы о вычислительной трудности двумерного поиска подстрок и поиске подпоследовательностей. Из Раздела 4 мы знаем, что даже расстояние Хэмминга (при предположении P$\neq$NP) не может быть вычислено за полиномиальное время. Необходимо найти новую модель сжатых данных, которая бы допускала эффективный поиск приближенного вхождения.

Список литературы

[1] Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in Z-compressed files. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1994.

[2] Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Inf. Process. Lett.*, 39(1):17–20, 1991.

[3] Piotr Berman, Marek Karpinski, Lawrence L. Larmore, Wojciech Plandowski, and Wojciech Rytter. On the complexity of pattern matching for highly compressed two-dimensional texts. *Journal of Computer and Systems Science*, 65(2):332–350, 2002.

[4] Patrick Cegielski, Irene Guessarian, Yury Lifshits, and Yuri Matiyasevich. Window subsequence problems for compressed texts. In *Proceedings of 1st International Symposium "Computer Science in Russia"*, LNCS 3967, pages 127–136. Springer-Verlag, 2006.

[5] Martin Farach and Mikkel Thorup. String matching in Lempel-Ziv compressed strings. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 703–712. ACM Press, 1995.

[6] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness.* Freeman, 1979. Русский перевод: М. Гэри и Д. Джонсон. Вычислительные машины и труднорешаемые задачи, Москва, Мир, 1982.

[7] Leszek Gąsieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996)*, LNCS 1097, pages 392–403. Springer-Verlag, 1996.

[8] Blaise Genest and Anca Muscholl. Pattern matching and membership for hierarchical message sequence charts. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN 2002)*, LNCS 2286, pages 326–340. Springer-Verlag, 2002.

[9] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997. Русский перевод: Д.Гасфилд. Строки, деревья и последовательности в алгоритмах: информатика и вычислительная биология. BHV, Санкт-Петербург, 2003.

[10] M. Hirao, A. Shinohara, M. Takeda, and S. Arikawa. Fully compressed pattern matching algorithm for balanced straight-line programs. In *Proc. of 7th International Symposium on String Processing and Information Retrieval (SPIRE2000)*, pages 132–138. IEEE Computer Society, 2000.

[11] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proceedings of the 11st Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1848, pages 195–209. Springer-Verlag, 2000.

[12] Takuya Kida, Tetsuya Matsumoto, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

[13] Sławomir Lasota and Wojciech Rytter. Faster algorithm for bisimulation equivalence of normed context-free processes. In *Proc. MFCS'06*, LNCS 4162, pages 646–657. Springer-Verlag, 2006.

[14] Yury Lifshits and Markus Lohrey. Quering and embedding compressed texts. In *Proc. of MFCS'06*, LNCS 4162, pages 681–692. Springer-Verlag, 2006.

[15] Markus Lohrey. Word problems on compressed word. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, LNCS 3142, pages 906–918. Springer-Verlag, 2004.

[16] Ayumi Shinohara Marek Karpinski, Wojciech Rytter. Pattern-matching for strings with short descriptions. In *Proc. of CPM'95*, LNCS 937, pages 205–214. Springer-Verlag, 1995.

[17] Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An improved pattern matching algorithm for strings in terms of straight line programs. In *CPM '97: Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, LNCS 1264, pages 1–11. Springer-Verlag, 1997.

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

[19] Gonzalo Navarro and Mathieu Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pages 14–36. Springer-Verlag, 1999.

[20] Aris Pagourtzis and Stathis Zachos. The complexity of counting functions with easy decision version. In *Proc. of MFCS'06*, LNCS 4162, pages 741–752. Springer-Verlag, 2006.

[21] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *Second Annual European Symposium on Algorithms (ESA'94)*, LNCS 855, pages 460–470. Springer-Verlag, 1994.

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

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

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

[25] Wojciech Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, LNCS 3142, pages 15–27. Springer-Verlag, 2004.

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