Содержание

1 Введение

В современном мире объемы информации растут очень быстрыми темпами и, как следствие, растут размеры решаемых задач. На данный момент задачи на больших объемах данных возникают в различных областях: информационный поиск, микробиология, социальные сети. Также в некоторых областях привлечение больших объемов данных может повысить точность получаемых моделей. Например, эта идея активно используется в теории машинного обучения.

С ростом размера входа для классических задач меняются и алгоритмы, способные их эффективно решать. Так, существует целый класс алгоритмов, ориентированных на работу с данными, не помещающимися в оперативную память, и оптимизирующих количество операций обращения к жесткому диску. Такие алгоритмы называются *алгоритмами эффективного ввода-вывода*.

С другой стороны, для ускорения работы с данными, в особенности редко меняющимися, разумно использовать предварительную обработку. В частности, в случае текстовой информации для этих целей можно использовать алгоритмы сжатия. При таком подходе мы экономим место и накапливаем особенности текста, а затем "моделируем" запуск строкового алгоритма на сжатом представлении.

Существуют модели сжатия, поддерживающие возможность обращения к исходным данным без распаковки. Такими моделями являются, например, прямолинейные программы (ПП) [?, ?, ?], коллаж-системы [?], представления с помощью антисловарей [?]. Мы будем рассматривать сжатие текста с помощью прямолинейных программ. Данная модель хорошо структурирована и, кроме того, интересна благодаря связи с алгоритмом Лемпеля-Зива, широко применяемом на практике. Эта связь описана в работе Риттера [?] и будет подробно обсуждаться в данной работе. Также существует широкий спектр классических строковых задач, сформулированных в терминах сжатых представлений и разрешимых за полиномиальное время от размера ПП. Такими задачами являются, например, **поиск сжатого образца в сжатом тексте** [?], **поиск наибольшей общей подстроки двух сжатых строк** [?], **поиск всех палиндромов в сжатой строке** [?], **поиск квадратов в сжатой строке** [?].

Несмотря на широкий спектр теоретически решенных задач, до сих пор актуален вопрос о практической пригодности ПП как модели сжа-

тия. Известно, что задача построения минимальной по размеру ПП для заданной строки является NP-полной. Поэтому вызывают интерес алгоритмы, позволяющие построить некоторое приближение к минимальной ПП. В данной работе мы хотим ответить на следующие вопросы: насколько сложно строить ПП и насколько размер построенных ПП отличается от размера архива, построенного классическими алгоритмами сжатия.

Первая часть работы посвящена классическому алгоритму Лемпеля-Зива сжатия текстов. Мы опишем широко известную реализацию данного алгоритма, основанную на суффиксном дереве. Также мы предложим альтернативную реализацию, основанную на суффиксном массиве. Она имеет более плохую теоретическую оценку сложности, однако более перспективна с точки зрения практического использования.

Вторая часть работы посвящена преобразованию архива, являющегося результатом алгоритма Лемпеля-Зива, в прямолинейную программу. Мы обсудим алгоритм Риттера, решающий эту задачу. Также мы предложим новый алгоритм, который сокращает количество дорогих операций алгоритма Риттера и на практике работает быстрее.

В заключительной части работы мы представим практические результаты по степени сжатия текстов с помощью ПП в сравнении со степенью сжатия алгоритмом Лемпеля-Зива. Также мы сравним практические результаты работы двух алгоритмов построения ПП. Поскольку природа исходного текста влияет на степень сжатия и время обработки, в работе рассматриваются следующие типы текстов:

- **ДНК** – строки, свойства которых интересны на практике и активно изучаются;
- **случайные строки** – предположительно худший вход для алгоритмов сжатия;
- **строки Фибоначчи** – предположительно один из лучших входов алгоритмов сжатия.

2 Обозначения и определения

Зафиксируем произвольный непустой конечный алфавит Σ. *Строкой или словом* будем называть элемент из множества Σ^+. *Конкатенацию* двух строк T и S будем обозначать символом "$\cdot$": $A = T \cdot S$. *Размером* или *длиной* строки T будем называть количество символов в ней и обозначать $|T|$. Символ, находящийся в позиции i заданной строки T, будем обозначать через $T[i]$. *Подстроку*, расположенную в строке T, начиная с позиции i и заканчивая позицией j (включительно), будем обозначать через $T[i \dots j]$. *Суффиксом с номером* i будем называть подстроку $T[i \dots |T|]$.

Строками Фибоначчи будем называть последовательность строк $\{f_i\}, i \geq 1$, определяемую следующими соотношениями:

1. $f_1 =$ "b";
2. $f_2 =$ "a";
3. $f_i = f_{i-1} \cdot f_{i-2}$, для всех $i \geq 3$.

Пример. 7-е слово Фибоначчи $f_7 =$ «*abaababaabaab*».

3 LZ77-факторизация

Определение. Под *факторизацией* F строки T будем понимать некоторое разбиение строки T на слова $f_1, f_2, \ldots, f_k$, обладающее следующими свойствами:

1. $T = f_1 \cdot f_2 \cdot \ldots \cdot f_k$;
2. для любого $1 \leq i \leq k$ верно, что строка f_i либо состоит из одного символа, либо встречается как подстрока в строке $f_1 \cdot \ldots \cdot f_{i-1}$.

Слова $f_1, f_2, \ldots, f_k$ будем называть *факторами*.

В частности, алгоритм Лемпеля-Зива $LZ77$ задает следующую факторизацию строки:

1. $f_1 = T[1]$;
2. Пусть уже построена факторизация для префикса строки T длины k: $T[1 \ldots k] = f_1 \cdot f_2 \cdot \ldots \cdot f_{i-1}$. Тогда $f_i = T[k+1]$, если символ $T[k+1]$ не встречается в строке $T[1 \ldots k]$. В противном случае f_i равен наибольшему префиксу строки $T[k+1 \ldots |T|]$, который встречается как подстрока в $T[1 \ldots k]$.

Обозначим $LZ77$-факторизацию строки T через $LZ77(T)$, а ее *размер* положим равным числу факторов и обозначим через $|LZ77(T)|$.

Пример: Рассмотрим $LZ77$ и некоторые факторизации F_1 и F_2 7-го слова Фибоначчи.

$$LZ77(\text{«}abaababaabaab\text{»}) = f_1\ f_2\ f_3\ f_4\ f_5\ f_6 = a\ b\ a\ aba\ baaba\ ab;$$

$$F_1(\text{«}abaababaabaab\text{»}) = f_1\ f_2\ f_3\ f_4\ f_5\ f_6\ f_7 = a\ b\ a\ a\ b\ abaab\ aab$$

$$F_2(\text{«}abaababaabaab\text{»}) = f_1\ f_2\ f_3\ f_4\ f_5\ f_6\ f_7\ f_8 = a\ b\ a\ aba\ b\ a\ aba\ ab.$$

Лемма 1 (о минимальности $LZ77$-факторизации)**.**
Среди всех возможных факторизаций заданной строки T минимальной по количеству факторов является $LZ77$-факторизация.

Доказательство: Пусть $g_1 \cdot g_2 \cdot \ \ldots \ \cdot g_r$ – некоторая факторизация строки T, а $f_1 \cdot f_2 \cdot \ldots f_k$ – ее $LZ77$-факторизация. Предположим, что $k > r$. Докажем по индукции, что для любого $i \leq r$ $|f_1 \cdot \ldots f_i| \geq |g_1 \cdot \ldots g_i|$:
База индукции $f_1 = g_1 = T[1]$;
Шаг индукции Пусть утверждение доказано для всех $i = [1 \ldots s]$. Докажем для $s + 1$. Рассмотрим два случая:

1. $|f_1 \cdot f_2 \cdot \ \ldots \ \cdot f_s| = |g_1 \cdot g_2 \cdot \ \ldots \ \cdot g_s|$. В этом случае $|f_{s+1}| \geq |g_{s+1}|$ в силу построения $LZ77$-факторизации (выбирается наидлиннейший из возможных факторов), следовательно, $|f_1 \cdot f_2 \cdot \ \ldots \ \cdot f_{s+1}| \geq |g_1 \cdot g_2 \cdot \ \ldots \ \cdot g_{s+1}|$;

2. $|f_1 \cdot f_2 \cdot \ldots \cdot f_s| > |g_1 \cdot g_2 \cdot \ldots \cdot g_s|$. Предположим, что $n = |f_1 \cdot f_2 \cdot \ldots \cdot f_{s+1}| < |g_1 \cdot g_2 \cdot \ \ldots \ \cdot g_{s+1}| = m$. Но это значит, что суффикс строки g_{s+1} длиной $m - n$ входит как подстрока в строку $g_1 \cdot g_2 \cdot \ \ldots \ \cdot g_s$, а значит, и в строку $f_1 \cdot f_2 \cdot \ \ldots \ \cdot f_s$. Это значит, что фактор f_{s+1} можно продолжить суффиксом строки g_{s+1} длиной $m - n$. Значит, предположение неверно.

Таким образом, $|T| = |g_1 \cdot \ldots g_r| \leq |f_1 \cdot \ldots f_r| < |f_1 \cdot \ldots f_k| < |T|$. Полученное противоречие завершает доказательство. □

Сформулируем задачу построения $LZ77$-факторизации по заданной строке T.
Задача: Построение $LZ77$-факторизации.
Вход: Строка T длины n.
Выход: $LZ77$-фактризация $f_1 \cdot f_2 \cdot \cdots \cdot f_k$ строки T.

В следующем разделе мы опишем алгоритм построения $LZ77(T)$, имеющий минимально возможную теоретическую оценку сложности.

3.1 Построение LZ77-факторизации с помощью суффиксного дерева

3.1.1 Определение и основные свойства суффиксного дерева

Пусть T – строка длины n.

Неявное суффиксное дерево для строки T – это способ представления этой строки. Неформально говоря, чтобы построить неявное суффиксное

дерево для строки T, нужно взять все n суффиксов, подвесить их за начала и склеить все ветки, идущие по одинаковым буквам.

Чтобы построить *суффиксное дерево* для строки T, нужно вначале к строке T приписать специальный символ, не равный ни одному символу исходной строки (будем обозначать его "\$"). Затем для строки $T\$$ требуется построить неявное суффиксное дерево. Суффиксное дерево для строки T будем обозначать через $ST(T)$.

Заметим, что из-за приписанного в конец специального символа ни один суффикс не может полностью лежать в другом суффиксе. Поэтому в суффиксном дереве будет ровно $n + 1$ листьев.

Суффиксное дерево (как и неявное суффиксное дерево) можно хранить, используя $O(n)$ памяти. Для этого оставим в дереве только те вершины, которые имеют не менее двух сыновей, а на ребрах будем вместо строк хранить ссылку на подстроку $T[i \dots j]$. Дерево в таком виде называется *сжатым*.

Заметим, что в сжатом суффиксном дереве не может быть более n внутренних узлов, поскольку каждый внутренний узел добавляет к своему поддереву как минимум 1 лист. Но листьев всего $n + 1$.

Пример. На рисунке ниже изображен пример суффиксного дерева. Для наглядности на ребрах изображены строки, хотя на самом деле там должны храниться только пары чисел i, j – сылки на соответствующие подстроки $T[i \dots j]$. Около каждого листа написан номер суффикса, который в нем заканчивается.

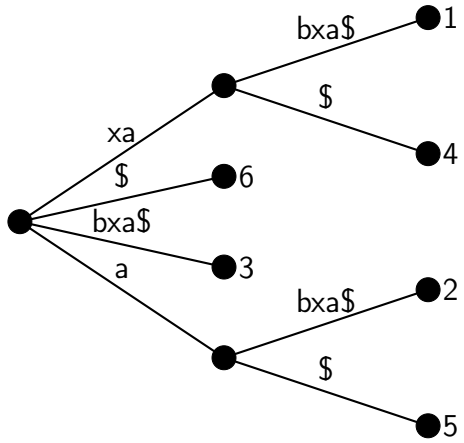


Рисунок 1. Суффиксное дерево для строки «$xabxa$».

Примечание В дальнейшем мы будем отождествлять понятия “суффиксное дерево”, “неявное суффиксное дерево” и “сжатое суффиксное дерево”. Все три объекта будем называть просто суффиксным деревом. Также в дальнейших рассуждениях нам будет удобно считать, что мы работаем с несжатым деревом. Это мысленные допущения, которые не влияют на суть алгоритма, но упрощают его описание.

Утверждение([**?**]). Существует алгоритм, который позволяет построить $ST(T)$ за время $O(|T|)$.

При этом алгоритм состоит из $|T|$ итераций, и после i-й итерации результатом алгоритма является суффиксное дерево ST_i для строки $T[1..i]$. Алгоритм называется *алгоритмом Укконена*.

3.1.2 Описание алгоритма построения LZ77-факторизации

Алгоритм построения $LZ77(T)$ следующий:
База: Положим $f_1 = T[1]$ и построим дерево ST_1 для первого символа строки.

Шаг:

1. Пусть уже построены факторизация $f_1 \cdot f_2 \cdot \ldots f_{i-1}$, такая, что $f_1 \cdot f_2 \cdot \ldots f_{i-1} = T[1 \ldots k]$, и суффиксное дерево ST_k для префикса $T[1 \ldots k]$.

2. С помощью суффиксного дерева найдем наибольший префикс P строки $T[k{+}1 \ldots |T|]$, входящий как подстрока в $T[1 \ldots k]$. Для этого начнем спуск с корня дерева. Найдем ребро, помеченное символом $T[k{+}1]$, и спустимся по этому ребру. Повторим процедуру для этого узла и символа $T[k+2]$ и т.д до тех пор, пока не придем в узел, из которого не исходит ребра, помеченного нужным нам символом.

   Если длина такого префикса равна нулю, то в качестве P возьмем символ $T[k+1]$.

3. $f_i = P$

4. Достроим дерево ST_k до дерева $ST_{k+|f_i|}$ над префиксом $f_1 \cdot f_2 \cdot \ldots f_i$.

5. Будем продолжать эту итерацию до тех пор, пока не построим полную факторизацию строки.

Сложность:

1. Алгоритм состоит ровно из k итераций, где k – количество факторов в построенной факторизации.

2. Пункт 2 i-й итерации выполняется за время $O(|f_i|)$, поскольку во время спуска алгоритм посетит $|f_i|+1$ узел дерева (считая корень). Таким образом, время выполнения пункта 2 в течение всех итераций есть $O(|T|)$.

3. Пункт 4 i-й итерации выполняется за некоторое время t_i. Результатом выполнения данного пункта на всех итерациях будет суффиксное дерево для строки T. Поскольку мы используем алгоритм Укконена, то общее время построения дерева есть $O(|T|)$. Следовательно, общее время выполнения пункта 4 в течение всех итераций есть $O(|T|)$.

Таким образом, общее время работы алгоритма есть $O(|T|)$.

Недостатки алгоритма:

1. Алгоритм имеет линейную оценку сложности, но за оценкой $O(|T|)$ скрывается достаточно большая константа, что существенным образом сказывается на реальной производительности.

2. Если обрабатываемая строка перестает помещаться в оперативную память, возникают большие трудности с адаптацией этого алгоритма. Суффиксное дерево является ссылочной структурой. При переносе этой структуры во внешнюю память любой переход по ссылке может означать обращение к диску.

В следующем разделе мы представим другой алгоритм построения $LZ77$-факторизации. Он имеет более плохую теоретическую оценку сложности, но является более удобным для использования на практике.

3.2 Построение LZ77-факторизации с помощью суффиксного массива

3.2.1 Определение и основные свойства суффиксного массива

Пусть T – строка длины n.

Суффиксный массив для строки T – это массив чисел длины n такой, что в позиции i в этом массиве расположен номер того суффикса, который был бы на позиции i среди всех суффиксов строки T, отсортированных лексикографически. Проще говоря, суффиксный массив – это лексикографически отсортированный массив суффиксов строки. Но у каждого суффикса есть номер, поэтому в массиве хранятся только номера соответствующих суффиксов. Суффиксный массив для строки T будем обозначать $SA(T)$.

Пример: Отсортированный массив суффиксов для строки "$abaab$" имеет вид ["aab","ab","$abaab$","b","$baab$"]. Следовательно, суффиксный массив для этой строки имеет вид $[3, 4, 1, 5, 2]$.

Наибольший общий префикс двух строк A и B – это строка P, такая, что:

- P является префиксом как строки A, так и строки B.
- Символы $A[|P| + 1]$ и $B[|P| + 1]$ различны (при условии, что оба существуют).

Длину наибольшего общего префикса двух суффиксов, расположенных в суффиксном массиве в позициях с номерами i и j, будем обозначать $lcp(i, j)$.

Массив наибольших общих префиксов LCP для строки T – это массив чисел длины n, в котором в позиции i записана длина наибольшего общего префикса суффиксов, расположенных в позициях i и $i - 1$ суффиксного массива. Массив наибольших общих префиксов для строки T будем обозначать $LCP(T)$.

Таким образом, $lcp(i, i - 1) = LCP(i)$.

Пример: Массивы SA и LCP для 7-го слова Фибоначчи.

$$SA(\text{«}abaababaabaab\text{»}) = [11, 8, 3, 12, 9, 6, 4, 1, 13, 10, 7, 2, 5]$$

$$LCP(\text{«}abaababaabaab\text{»}) = [0, 3, 4, 1, 2, 5, 3, 3, 0, 1, 4, 5, 2]$$

Когда говорят о суффиксном массиве как о структуре данных, то нередко имеют в виду связку "суффиксный массив" + "массив наибольших общих префиксов".

Лемма 2 (о наибольшем общем префиксе [?])**.**
Пусть T – строка длины n, построены $SA(T)$ и $LCP(T)$. Тогда для любой пары чисел i, j, таких, что $i < j \leq n$ верно, что $lcp(i, j) = min(LCP[i + 1], \ldots, LCP[j])$.

Доказательство: Докажем индукцией по второму аргументу функции lcp. Пусть первый аргумент зафиксирован и равен i.

База:
Если $j = i + 1$, то $lcp(i, j) = LCP[i + 1]$ и утверждение, очевидно, верно.

Шаг:
Пусть утверждение доказано для $j = i + 1, i + 2, \ldots i + k - 1$. Докажем для $j = i + k$. $lcp(i, i + k - 1) = min(LCP[i + 1], \ldots, LCP[i + k - 1])$ (по индукции). Обозначим $lcp(i, i+k-1)$ через s_1. $lcp(i+k-1, i+k)$ обозначим через s_2. Обозначим суффикс, расположенный в позиции i, через A, суффикс в позиции $i + k - 1$ через B, а суффикс в позиции j через C. Рассмотрим три случая:

1. $s_1 = s_2 = s$. Это значит, что $A[1 \ldots s] = B[1 \ldots s] = C[1 \ldots s]$. При этом $A[s + 1] \neq B[s + 1]$, а $B[s + 1] \neq C[s + 1]$. В таком случае $A[s + 1]$ не может быть равно $C[s + 1]$, поскольку в силу лексикографической сортировки $A[s + 1] < B[s + 1] < C[s + 1]$. Таким образом, $lcp(i, j) = s = min(lcp(i, i+k-1), LCP[j]) = min(LCP[i+1], \ldots, LCP[j])$.

2. $s_1 < s_2$. Это значит, что $A[1 \ldots s_1] = B[1 \ldots s_1]$, а $B[1 \ldots s_2] = C[1 \ldots s_2]$. При этом $A[s_1 + 1] \neq B[s_1 + 1]$, но $B[s_1 + 1] = C[s_1 + 1]$. Следовательно, $A[s_1 + 1] \neq C[s_1 + 1]$. Таким образом, длина наибольшего общего префикса между A и C равна s_1. Итак, $lcp(i, j) = lcp(i, i + k - 1) = min(lcp(i, i + k - 1), LCP[j]) = min(LCP[i + 1], \ldots, LCP[j])$.

3. $s_1 > s_2$. Рассмотрение этого пункта аналогично предыдущему. □

Следствие (о монотонности функции lcp).
Зафиксируем некоторую позицию i в суффиксном массиве. Тогда из того, что $j < k < i$ следует, что $lcp(j, i) \leq lcp(k, i)$. Аналогично, из того, что $j > k > i$ следует, что $lcp(j, i) \leq lcp(k, i)$.

3.2.2 Структура данных "разреженная таблица" для поиска минимума на отрезке массива

Пусть нам дан массив чисел A размера n. Требуется построить структуру данных, позволяющую для любых $i, j : i \leq j \leq n$ за $O(1)$ вычислять минимум среди элементов $A[i], \ldots, A[j]$.

Для простоты будем считать, что $n = 2^k$ для некоторого k. Легко понять, что такое допущение никак не влияет на исходную задачу, поскольку мы можем дополнить исходный массив до нужного размера, дописав в конец несколько элементов со значением "бесконечность".

Будем строить двумерный массив ST размера $n \times k$ по следующему правилу:

$$ST[i][j] = \begin{cases} A[i], \text{если } j = 0 \\ \min(ST[i][j-1], ST[i + 2^{j-1}][j-1]), \text{иначе.} \end{cases}$$

Данный массив будем называть *разреженной таблицей.*

Из определения массива очевидно, что $ST[i][j]$ – минимум среди чисел $A[i], A[i+1], \ldots, A[i + 2^j - 1]$.

Лемма 3 (о вычислении минимума на отрезке массива)**.**
Пусть нам дан массив чисел A размера $n = 2^k$ и для него построен разреженная таблица ST. Тогда для $i, j : i \leq j \leq n$ верно, что $min(A[i], \ldots, A[j]) = min(ST[i][r], ST[j - 2^r + 1][r])$, где r – максимальное число такое, что $2^r \leq j - i + 1$.

Доказательство: $ST[i][r]$ "покрывает" отрезок от i до $i + 2^r - 1$. $ST[j - 2^r + 1][r]$ "покрывает" отрезок от $j - 2^r + 1$ до j. В силу выбора значения r выполнено неравенство $j - 2^r + 1 \leq i + 2^r - 1$. □

Таким образом, требуемая структура данных строится за время $O(n \log n)$ и требует $O(n \log n)$ памяти.

3.2.3 Массив “Ближайших меньших”

Пусть нам дан массив чисел A размера n, все элементы которого различны. Рассмотрим массив чисел B, где $B[i]$ равен индексу j ближайшего к i слева элемента массива A, такого, что $A[j] < A[i]$. Если такого индекса не существует, то $B[i] = 0$. Такой массив будем называть массивом “ближайших меньших”.

Опишем алгоритм построения такого массива. Заведем *стек* S, который будет содержать пары чисел (a, b). Верхушку стека будем обозначать $TOP(S)$, а первый и второй элементы пары верхушки стека будем обозначать $TOP(S).First$ и $TOP(S).Second$, соответственно. Операцию удаления верхушки стека будем обозначать $POP(S)$. Операцию добавления элемента (a, b) в стек будем обозначать $PUSH(S, a, b)$. Описание алгоритма приведем в виде псевдокода:

Построение массива “ближайших меньших”

```
1  PUSH(S, 0, −∞)
2  for i ← 1 to n
3        do
4           while TOP(S).Second > A[i]
5                 do POP(S)
6
7           B[i] ← TOP(S).First
8           PUSH(S, i, A[i])
```

Утверждение. Описанный алгоритм корректно строит массив “ближайших меньших” за время $O(n)$ с использованием $O(n)$ дополнительной памяти.

Доказательство: Асимптотика работы пропорциональна количеству добавлений и удалений элементов из стека. Поскольку каждый элемент массива добавляется в стек ровно один раз и удаляется из стека не более одного раза, то алгоритм, очевидно, выполняет $O(n)$ действий. В качестве дополнительной памяти используется только стек, размер которого не может стать больше размера исходного массива. Таким образом, сложность алгоритма доказана.

Для доказательства корректности рассмотрим i-ю итерацию алгоритма. Возможны два случая:

1. $A[i] > TOP(S).Second$. Поскольку после предыдущего шага в верхушке стека расположен предыдущий элемент массива, то он, очевидно, является ближайшим для i элементом, меньшим $A[i]$. Следовательно, в таком случае алгоритм корректно присвоит в $B[i]$ значение $i - 1$.

2. $A[i] < TOP(S).Second$. Следовательно, $A[i] < A[i-1]$, а значит $B[i] \leq B[i-1]$. То есть, ответом для текущего шага является либо тот же элемент, что был ответом на предыдущем шаге (а он остался в стеке), либо какой-то элемент левее (он тоже остался в стеке после предыдущего шага). Значит, на текущем шаге алгоритм корректно найдет ответ. $\square$

3.2.4 Эффективный алгоритм построения $LZ77$-факторизации

Алгоритм условно разделим на две части: стадия предварительной обработки и стадия вычисления основного результата. Стадия предварительной обработки состоит из 6 частей.

1. Построим суффиксный массив SA и массив LCP для строки T.
2. Построим массив P такой, что $P[i]$ равен позиции суффикса с номером i в суффиксном массиве.
3. Построим массив $SARev$ – массив SA, записанный в обратном порядке.
4. Построим разреженную таблицу ST для массива LCP.
5. Построим массив "ближайших меньших" NS для массива SA
6. Построим массив "ближайших меньших" $NSRev$ для массива $SARev$.

Основной алгоритм опишем индуктивно.
База: $f_1 = T[1]$.

Шаг:
Пусть уже построена факторизация $f_1 \cdot f_2 \cdot \ldots \cdot f_s$ такая, что $f_1 \cdot f_2 \cdot \ldots \cdot f_s = T[1 \ldots k]$. Опишем процесс получения очередного фактора.

1. Пусть $P[k+1] = i$.
2. Далее алгоритм делится на 2 независимые части: поиск ответа слева от позиции i и поиск ответа справа от позиции i. В следующих двух пунктах будет описан поиск ответа для "левой" части алгоритма.
3. Пусть j – "ближайший меньший" элемент слева от позиции i, то есть, $NS[i] = j$.
4. Возможны три варианта:
   (a) $j = 0$. Это значит, что слева от позиции i нет элементов, меньших $k+1$. Следовательно, ответом в таком случае будет 0.

(b) $lcp(j, i) \leq SA[i] - SA[j]$. Это означает, что наибольший из возможных префиксов полностью лежит в строке $T[1 \dots k]$. Значит, ответом в этом случае будет $lcp(j, i)$.

(c) $lcp(j, i) > SA[i] - SA[j]$. Это означает, что найденный общий префикс пересекает суффикс $k + 1$, а длина ответа в таком случае равна $SA[i] - SA[j]$, то есть расстоянию между суффиксами $SA[i]$ и $SA[j]$ в строке. Значит, левее от суффикса $SA[j]$ может найтись суффикс, дающий больший ответ. Перейдем к суффиксу, расположенному в позиции $NS[j]$, и повторим рассмотрение случаев $a - c$. Эту процедуру назовем "скачком". Будем повторять "скачки" до тех пор, пока не придем в такую позицию r, что $lcp(NS[r], i) \leq SA[i] - SA[NS[r]]$. Тогда ответом будет либо $lcp(NS[r], i)$, либо $SA[i] - SA[r]$.

5. Найдем ответ для "правой" части алгоритма. Это делается абсолютно аналогично пункту 3. В данном разделе будут использоваться массивы $SARev$ и $NSRev$.

6. Из результатов "левой" и "правой" частей алгоритма выберем наибольший. Пусть он равен len. Если $len = 0$, то очередным фактором будет буква $T[k + 1]$. Иначе очередным фактором будет подстрока $T[k + 1 \dots k + len]$.

Сложность алгоритма:

Для начала оценим сложность стадии предварительной обработки.

1. Суффиксный массив можно построить за время за $O(n)$[**?**].

2. После построения суффиксного массива массив LCP можно построить за $O(n)$[**?**].

3. Для построения массива P достаточно для каждого i положить $P[SA[i]] = i$. Следовательно, время построения массива P есть $O(n)$.

4. Массив $SARev$ также строится за $O(n)$.

5. Время построения разреженной таблицы $O(n \log n)$.

6. Время построения массивов NS и $NSRev$ равно $O(n)$.

Оценим сложность основной части алгоритма.

1. Алгоритм состоит из k итераций.

2. В каждой итерации делается серия "скачков". Пусть в итерации с номером i алгоритм сделал t_i "скачков". Заметим, что после каждого "скачка" длина очередного фактора увеличивается как минимум на 1. Следовательно, $|f_i| \geq t_i$. Следовательно, $t_1 + t_2 + \cdots + t_k \leq |f_1| + |f_2| + \cdots + |f_k| = n$. Таким образом, общее время основного алгоритма есть $O(n)$.

Таким образом, алгоритм работает за время $O(n \log n)$ и использует $O(n \log n)$ дополнительной памяти.

По сравнению с алгоритмом, основанном на суффиксном массиве, данная реализация имеет одно существенное преимущество: она легко адаптируется на случай, когда оперативной памяти оказывается недостаточно. Простота достигается за счет того, что все описанные в реализации структуры данных – это обычные массивы.

4 ПП как сжатое представление строк

4.1 Обозначения и определения

Прямолинейная программа (ПП) – это последовательность правил вывода вида:

$$\mathcal{X}_1 = expr_1, \mathcal{X}_2 = expr_2, \ldots, \mathcal{X}_n = expr_n,$$

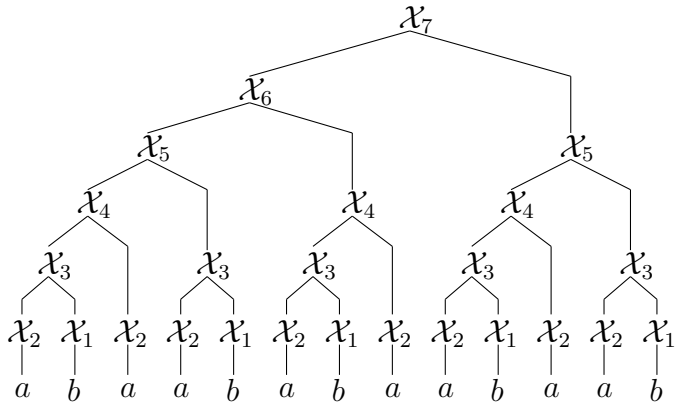
где $\mathcal{X}_i$ – это переменные, а $expr_i$ – выражения вида:

- $expr_i$ – символ из алфавита Σ (такие правила будем называть *терминальными*), или
- $expr_i = \mathcal{X}_l \cdot \mathcal{X}_r (l, r < i)$ (такие правила будем называть *нетерминальными*).

Пример: Рассмотрим ПП $\mathcal{X}$, которая порождает текст «*abaababaabaab*»:

$$\mathcal{X}_1 = b, \mathcal{X}_2 = a, \mathcal{X}_3 = \mathcal{X}_2 \cdot \mathcal{X}_1, \mathcal{X}_4 = \mathcal{X}_3 \cdot \mathcal{X}_2,$$
$$\mathcal{X}_5 = \mathcal{X}_4 \cdot \mathcal{X}_3, \mathcal{X}_6 = \mathcal{X}_5 \cdot \mathcal{X}_4, \mathcal{X}_7 = \mathcal{X}_6 \cdot \mathcal{X}_5$$

Тогда дерево вывода грамматики $\mathcal{X}$ имеет вид:



В дальнейшем дерево вывода грамматики $\mathcal{X}$ будем обозначать через $Tree(\mathcal{X})$.

В работе принято следующее соглашение: все прямолинейные программы обозначаются курсивными буквами, например, $\mathcal{X}$, а строки, которые выводятся из них, соответствующими обычными буквами, например, X. Аналогично, правило с номером i будем обозначать, через $\mathcal{X}_i$, а

соответствующую строку – X_i. *Размером* ПП $\mathcal{X}$ будем называть количество правил в ней и обозначать через $|\mathcal{X}|$. Под *высотой* дерева будем понимать наибольшую длину пути от корня до листа. *Высотой* ПП $\mathcal{X}$ будем называть высоту дерева $Tree(\mathcal{X})$ и обозначать $h(\mathcal{X})$. Аналогично, *высотой* некоторого правила $\mathcal{X}_i$ ПП $\mathcal{X}$ будем обозначать высоту соответствующего поддерева в дереве вывода и обозначать $h(\mathcal{X}_i)$.

4.2 AVL-сбалансированные ПП

Определение. Бинарное дерево называется *AVL-сбалансированным*, если для каждого его узла высоты левого и правого поддеревьев различаются не более, чем на 1.

Определение. Прямолинейную программу $\mathcal{T}$ будем называть AVL-*сбалансированной*, если дерево вывода $Tree(\mathcal{T})$ является AVL-сбалансированным.

Лемма 4 (О высоте AVL-сбалансированного дерева [?])**.**
AVL-сбалансированное дерево из n узлов имеет высоту порядка $O(\log n)$.

Следствие (О высоте AVL-сбаланированной ПП).
AVL-сбалансированная ПП $\mathcal{T}$ имеет высоту порядка $O(\log |T|)$.

В следующих двух разделам мы опишем две операции, которые поддерживают AVL-сбалансированные ПП.

4.2.1 Операция конкатенации

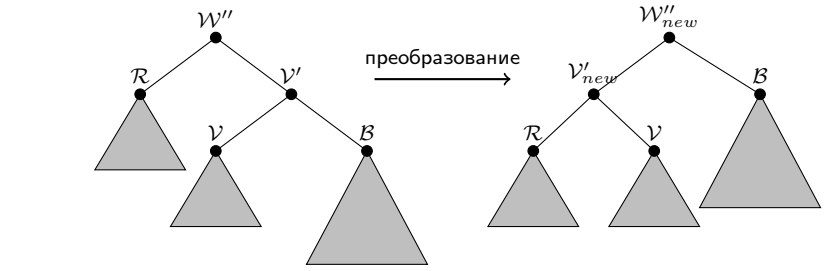
Задача: Конкатенация AVL-сбалансированных ПП.

Вход: $\mathcal{A}$, $\mathcal{B}$ – AVL-сбалансированные ПП.

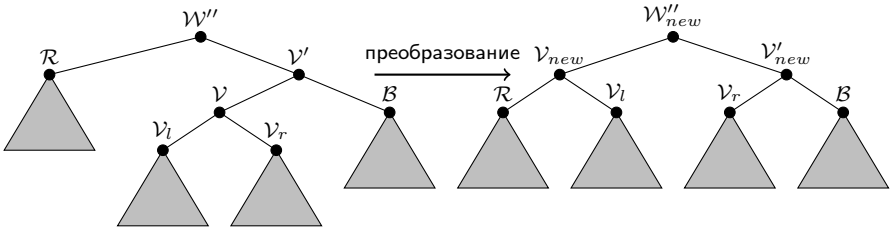
Выход: AVL-сбалансированная ПП $\mathcal{G}$, выводящая $A \cdot B$.

Алгоритм:
Предположим, что $h(\mathcal{A}) \geq h(\mathcal{B})$, другой случай аналогичен. Начиная с корня дерева $Tree(\mathcal{A})$, мы будем спускаться по самой правой ветке. Из свойства сбалансированности следует, что высота дерева сына будет отличаться от высоты дерева отца не более чем на 2. Поэтому мы сможем найти такое правило $\mathcal{V}$, что $h(\mathcal{B}) - h(\mathcal{V}) \in \{0, 1\}$. Пусть $\mathcal{W}$ – отец $\mathcal{V}$ в дереве $Tree(\mathcal{A})$, тогда добавим в ПП новое правило $\mathcal{V}'$ такое, то $\mathcal{V}$ и $\mathcal{B}$ – сыновья $\mathcal{V}'$ в дереве вывода, а $\mathcal{W}$ – отец $\mathcal{V}'$. При этом заметим, что в силу изменения правила $\mathcal{W}$ необходимо его заменить на новое правило $\mathcal{W}''$, его отца – аналогично, и т.д. до корня. Полученная ПП может оказаться несбалансированной на правой ветке. Возвращать сбалансированность начнем с правила $\mathcal{W}''$. На Рисунке 2 показаны преоб-

разования, которые возвращают свойство сбалансированности правилу $\mathcal{W}''$. В результате преобразований некоторые правила в ПП заменяются новыми правилами. В частности, как видно на рисунке, правило $\mathcal{W}''$ будет заменено на новое – $\mathcal{W}''_{new}$. Применение одного из преобразований, представленных на Рисунке 2, будем называть *операцией перебалансировки*. Заметим, что есть существенное отличие между опрерациями перебалансировки в бинарном дерева и в ПП. В дереве перебалансировка осуществляется за счет смены детей у некоторых узлов. В случае с ПП мы так поступить не можем, поскольку смена детей будет означать, что мы "испортили" какое-то правило вывода. Но это правило может присутствовать в ПП многократно, а измениться должно только в одном месте. Поэтому "портить" правила мы не можем, и необходимо вместо этого создавать новые правила вывода. После того, как к правилу $\mathcal{W}''$ применили операцию перебалансировки, несбалансированным может оказаться отец $\mathcal{W}''_{new}$ в дереве вывода. В этом случае применим операцию к нему и т.д. до корня.



$h(\mathcal{V}') = h(\mathcal{R}) + 2, h(\mathcal{V}) = h(\mathcal{B}) - 1.$



$h(\mathcal{V}') = h(\mathcal{R}) + 2, h(\mathcal{V}) = h(\mathcal{B}).$

Рисунок 2. Операция перебалансировки

Сложность алгоритма:

1. Вначале алгоритм ищет правило $\mathcal{V}$ такое, что $h(\mathcal{B}) - h(\mathcal{V}) \in \{0, 1\}$. Для этого достаточно спуститься в дереве $Tree(\mathcal{A})$ по правой ветке до правила высотой, отличающейся от $h(\mathcal{B})$ не более чем на 1. Следовательно, для этого алгоритм совершает $O(h(\mathcal{A}) - h(\mathcal{B}))$ действий.

2. После добавления правила $\mathcal{B}$ в ПП на всей правой ветке от правила $\mathcal{W}''$ до корня правила вывода заменяются новыми. Следовательно, добавляется $O(h(\mathcal{A}) - h(\mathcal{B}))$ новых правил вывода.

3. Если получанная ПП оказалась несбалансированной, алгоритм совершает серию операций перебалансировки. Каждая операция совершается за время $O(1)$ и в добавляет $O(1)$ новых правил вывода. Поскольку операции начинают выполняться от правила $\mathcal{W}''$ и идут максимум до корня, то алгоритм совершит не более $h(\mathcal{A}) - h(\mathcal{B})$ конкатенаций. Таким образом, перебалансировка ПП работает за время $O(h(\mathcal{A}) - h(\mathcal{B}))$ и в результате нее добавляется не более $O(h(\mathcal{A}) - h(\mathcal{B}))$ новых правил вывода.

Таким образом, операция конкатенации выполняется за время $O(h(\mathcal{A}) - h(\mathcal{B}))$ и порождает не более $O(h(\mathcal{A}) - h(\mathcal{B}))$ новых правил вывода.

Лемма 5 (о конкатенации [**?**])**.**
Пусть $\mathcal{A}, \mathcal{B}$ – два нетерминальных правила AVL*-сбалансированной ПП. Тогда мы можем сконструировать за $O(|h(\mathcal{A}) - h(\mathcal{B})|)$ операций перебалансировки* AVL*-сбалансированную ПП $\mathcal{G} = \mathcal{A} \cdot \mathcal{B}$, где $G = A \cdot B$, добавив не более $O(|h(\mathcal{A}) - h(\mathcal{B})|)$ новых правил вывода.*

Трудности в реализации:
Потенциально дерево может не помещаться в оперативную память. Это значит, что при реализации алгоритма необходимо самостоятельно имитировать ссылочную структуру дерева во внешней памяти. При этом, каждая операция перебалансировки может породить 3 новых правила и "убить" 3 старых правила. Значит, нужно уметь добавлять новые и удалять старые правила из грамматики. То есть, каждая операция перебалансировки может породить несколько обращений к внешней памяти. И с ростом размера входной строки стоимость этих действий будет только увеличиваться (чем больше данных хранится на дисковом пространстве,

тем более сложным является доступ к ним).
Данные наблюдения в некотором смысле меняют взгляд на оценку алгоритмов, использующих конкатенацию ПП. Сложность операции перебалансировки теперь нельзя оценивать как $O(1)$. Поэтому возникает идея, что при построении прямолинейной программы важно минимизировать количество операций перебалансировки (возможно, за счет усложнения других частей алгоритма). Реализацию этой идеи мы обсудим в следующих главах.

4.2.2 Операция взятия подстроки в AVL-сбалансированной ПП

Задача: Взятие подстроки в AVL-сбалансированной ПП.

Вход: $\mathcal{T}$ – AVL-сбалансированная ПП, и целые числа l и r, такие, что $1 \le l < r \le |T|$.

Выход: AVL-сбалансированная ПП $\mathcal{S}$, выводящая строку $T[l \dots r]$.

Алгоритм:
Будем работать с деревом вывода $Tree(\mathcal{T})$. Каждый узел $\mathcal{X}$ дерева соответствует некоторой подстроке $T[l_{\mathcal{X}} \dots r_{\mathcal{X}}]$. Будем считать, что для каждого узла $\mathcal{X}$ известны числа $l_{\mathcal{X}}$ и $r_{\mathcal{X}}$. Отрезок $[l_{\mathcal{X}} \dots r_{\mathcal{X}}]$ будем называть *интервалом узла* $\mathcal{X}$ и обозначать $Range(\mathcal{X})$. Левого сына узла $\mathcal{X}$ будем обозначать $\mathcal{X}_l$, правого – $\mathcal{X}_r$. Будем говорить, что набор узлов $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_t$ *покрывает* некоторый отрезок $[l \dots r]$, если $Range(\mathcal{S}_1) \cup Range(\mathcal{S}_2) \cup \dots \cup Range(\mathcal{S}_t) = [l \dots r]$.

Ниже приведено описание алгоритма взятия подстроки.

1. Найдем минимальный по высоте узел $\mathcal{U}$, такой, что $Range(\mathcal{U}) \supseteq [l \dots r]$.

   Для этого будем спускаться по дереву $Tree(\mathcal{T})$, начиная с корня. Пусть $\mathcal{V}$ – текущий узел. Если $Range(\mathcal{V}_l) \supseteq [l \dots r]$, то спустимся к левому сыну и продолжим процедуру, считая уже $\mathcal{V}_l$ текущим узлом. Если $Range(\mathcal{V}_r) \supseteq [l \dots r]$, то спустимся к правому сыну и продолжим процедуру, считая уже $\mathcal{V}_r$ текущим узлом. Если оба этих условия не выполняются, то текущий узел $\mathcal{V}$ является искомым.

2. Если $Range(\mathcal{U}) = [l \dots r]$, то искомый набор состоит из одного узла, и ответ найден.

3. В противном случае сначала перейдем к левому сыну узла $\mathcal{U}$ – $\mathcal{U}_l$. Будем спускаться от него по левой ветке до тех пор, пока интервал левого сына текущего узла пересекается с отрезком $[l \dots r]$. То есть, мы должны спуститься до такого узла $\mathcal{U}'$, что $Range(\mathcal{U}'_l) \cap [l \dots r] = \oslash$. Если такого узла нет, то ответ для левой части состоит из единственного узла $\mathcal{U}_l$. Если же узел есть, то повторим рекурсивно данный пункт, только в качестве узла $\mathcal{U}_l$ будет выступать узел $\mathcal{U}'_r$. По-

сле этого добавим к ответу правых детей всех узлов,которые мы прошли по пути от $\mathcal{U}_l$ к $\mathcal{U}'$, кроме самих $\mathcal{U}_l$ и $\mathcal{U}'$ (в порядке, противоположном порядку обхода). В результате мы построим набор узлов $\mathcal{S}_1, \ldots, \mathcal{S}_{t_1}$, покрывающих отрезок $[l \ldots c]$ для некоторого $c < r$.

Заметим, что полученный набор узлов возрастает по высоте, то есть $h(\mathcal{S}_1) > h(\mathcal{S}_2) > \cdots > h(\mathcal{S}_{t_1})$.

4. Проделаем аналогичную процедуру для правого сына узла $\mathcal{U} - \mathcal{U}_r$. Только в этом случае спуск будет осуществляться по правой ветке, к ответу будут добавляться левые дети пройденных узлов и в порядке обхода, а рекурсивное выполнение процедуры будет выполняться после пополнения ответа новыми узлами. В результате мы построим набор узлов $\mathcal{S}_1, \ldots, \mathcal{S}_{t_2}$, покрывающих отрезок $[c+1 \ldots l]$.

   Заметим, что полученный набор узлов убывает по высоте, то есть $h(\mathcal{S}_1) < h(\mathcal{S}_2) < \cdots < h(\mathcal{S}_{t_2})$.

5. Таким образом, мы построим набор узлов $\mathcal{S}_1, \ldots, \mathcal{S}_t$, покрывающих отрезок $[l \ldots r]$.

6. С помощью операции конкатенации построим ПП $\mathcal{S}$, выводящую $S_1 \cdot S_2 \cdot \ldots \cdot S_t$.

Сложность алгоритма:

1. В пункте 1 алгоритм совершает спуск по дереву до тех пор, пока не найдет узел с нужным свойством. Поскольку ПП $\mathcal{T}$ AVL-сбалансирована, то ее высота есть $O(\log |T|)$, следовательно, в данной части алгоритм выполнит не более $O(\log |T|)$ действий.

2. В пунктах 3 и 4 алгоритм также совершает спуски по дереву, следовательно, выполняет не более $O(\log |T|)$.

3. Заметим, что получившееся количество узлов есть $\Theta(\log |T|)$, поскольку во время спусков алгоритм на каждом уровне добавляет к ответу не более 1 узла.

4. В пункте 6 производится серия операций конкатенации. Сначала сконкатенируем правила $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{t_1}$ в заданном порядке. Результатом конкатенации будет ПП $\mathcal{S}_l$. Сложность конкатенации и число новых правил вывода будут пропорциональны $h(\mathcal{S}_2) - h(\mathcal{S}_1) +$

$h(\mathcal{S}_3) - h(\mathcal{S}_2) + \cdots + h(\mathcal{S}_{t_1}) - h(\mathcal{S}_{t_1-1}) = h(\mathcal{S}_{t_1}) - h(\mathcal{S}_1) = O(\log |T|)$. Аналогично поступим с правой частью, только там правила будем конкатенировать в порядке $\mathcal{S}_t, \mathcal{S}_{t-1}, \ldots, \mathcal{S}_{t_1+1}$ и в результате получим ПП $\mathcal{S}_r$. Сконкатенируем $\mathcal{S}_l$ и $\mathcal{S}_r$, это также можно сделать за время $O(\log |T|)$, породив при этом $O(\log |T|)$ новых правил. Таким образом, пункт 6 выполняется за время $O(\log |T|)$, порождая при этом $O(\log |T|)$ новых правил вывода.

Описанные в этом пункте рассуждения о сложности нестрогие. Строгое доказательство можно найти в работе Риттера [**?**].

Таким образом, операция взятия подстроки выполняется за время $O(\log |T|)$, при этом порождает не более $O(\log |T|)$ новых правил вывода.

4.3 Алгоритм Риттера построения ПП по некоторой факторизации строки

В этой главе мы обсудим алгоритм, который позволяет построить ПП по строке, если известна некоторая ее факторизация. Однако часто в контексте данного алгоритма под факторизацией понимается $LZ77$-факторизация. Данная ассоциация обусловлена "минимальностью" данной факторизации, которая была доказана в Лемме 1. Следствием этого свойства является отношение между количеством факторов в $LZ77$-факторизации некоторой строки T и размером произвольной ПП $\mathcal{T}$, выводящей T. Для более формального описания этого отношения, сформулируем следующую лемму.

Лемма 6 (о связи между размером ПП и числом $LZ77$-факторов [**?**])**.** *Для любой строки T и произвольной ПП $\mathcal{T}$, выводящей T, верно, что $|\mathcal{T}| \geq |LZ77(T)|$.*

Данная лемма показывает, что в минимальной ПП, выводящей строку T, не меньше $|LZ77(T)|$ правил. Это значит, что для сравнения некоторой ПП с минимальной достаточно сравнить количество правил в ней с числом факторов в $LZ77$-факторизации.

Далее мы опишем результат Риттера – алгоритм построения ПП $\mathcal{T}$, размер которой в $\log |T|$ раз больше $|LZ77(T)|$.

Задача: Построение ПП $\mathcal{T}$, выводящую заданную строку T.

Вход: Строка T длины n, факторизация $F(T) : T = f_1 \cdot f_2 \cdot \ldots f_k$. Также для каждого фактора f_i, если он не однобуквенный, известна позиция его вхождения в строку $T_{i-1} = f_1 \cdot f_2 \cdot \ \ldots \ \cdot f_{i-1}$, то есть, известны числа l_i и t_i такие, что $f_i = T[l_i \ldots r_i]$.

Выход: ПП $\mathcal{T}$, выводящая T.

Агоритм:
Будем строить прямолинейную программу итеративно.

База: Построим тривиальную AVL-сбалансированную ПП $\mathcal{T}_1$, выводящую однобуквенный фактор f_1.

Шаг:
Пусть мы уже построили AVL-сбалансированную ПП $\mathcal{T}_i$ для строки $T_i = f_1 \cdot f_2 \cdot \ldots f_i$. Покажем, как построить $\mathcal{T}_{i+1}$ для строки $T_{i+1} = f_1 \cdot f_2 \cdot \ldots \cdot f_{i+1}$.

1. Возьмем фактор f_{i+1}. Если это не буква, то известна его позиция вхождения в строку T_i. Пусть $f_{i+1} = T_i[l_i \ldots r_i]$. Если это буква, то построим из нее тривиальную ПП $\mathcal{S}$ и перейдем к пункту 3.

2. С помощью операции взятия подстроки построим ПП $\mathcal{S}$, выводящую строку $T_i[l_i \ldots r_i]$.

3. С помощью алгоритма конкатенации построим ПП $\mathcal{T}_{i+1} = \mathcal{T}_i \cdot \mathcal{S}$.

Сложность алгоритма:

1. По условию для каждого фактора известны позиции l_i и r_i его вхождения в строку. Это не является серьезным допущением, поскольку во время построения факторизации можно для каждого фактора запомнить его позицию вхождения в строку. Поэтому в ходе построения ПП вычисление значений l_i и r_i не требуется.

2. Операция взятия подстроки работает за время $O(\log |T|)$ и порождает не более $O(\log |T|)$ новых правил вывода.

3. Конкатенация ПП $\mathcal{T}_i$ и $\mathcal{S}$ также требует $O(\log |T|)$ времени и порождает $O(\log |T|)$ новых правил.

Таким образом, можно сформулировать следующую теорему:

Теорема 1 ([?])**.**
Пусть задана строка T и некоторая ее факторизация $f_1 \cdot f_2 \cdot \ldots f_k$. Тогда алгоритм Риттера за время $O(k \cdot \log |T|)$ построит ПП $\mathcal{T}$,выводящую T, размера $O(k \cdot \log |T|)$.

У описанного алгоритма есть узкие места. Одно из них проиллюстрируем примером.

Пример. Пусть $n > 0$. Рассмотрим строку $S = ba^{2^n} ba^{2^{n-1}} \ldots ba$. $LZ77$-факторизация этой строки имеет вид: $b \cdot a \cdot a \cdot a^2 \cdot a^4 \cdot \ldots \cdot a^{2^{n-1}-1} \cdot$

$ba^{2^{n-1}} \cdot ba^{2^{n-2}} \cdot \dots \cdot ba$. Пусть $\mathcal{S}$ – прямолинейная программа, построенная для строки $S_1 = ba^{2^n}$. Оценим, сколько операций перебалансировки может произвести алгоритм Риттера в ходе построения прямолинейной программы для всей строки S. При обработке фактора $S_2 = ba^{2^{n-1}}$ алгоритм может произвести $h(\mathcal{S}) - h(\mathcal{S}_2) \approx n - (n-1) = 1$ операций перебалансировки, при этом после конкатенации высота $\mathcal{S}$ увеличится на 1. Тогда при обработке строки $S_3 = ba^{2^{n-2}}$ может быть произведено $h(\mathcal{S}) - h(\mathcal{S}_3) \approx n + 1 - (n-2) = 3$ перебалансировки. Таким образом, для общей оценки количества требуется посчитать сумму $\sum_{i=0}^{n-1}(n + (n-1-i) - (i)) = \sum_{i=0}^{n-1}(2n - 2i - 1) = n^2$. Заметим, что факторы $S_2, S_3, \dots, S_{n+1}$ независимы. То есть, все они входят в строку S_1. Это значит, что мы их могли обработать группой, то есть сконкатенировать, и только затем добавить к S_1. При этом мы могли конкатенировать данные правила с конца. В этом случае общее количество операций перебалансировки было бы не больше n. Это наблюдение лежит в основе идеи улучшения алгоритма Риттера с точки зрения количества перебалансировок.

4.4 Модернизированный алгоритм Риттера

Алгоритм основан на простой идее: в ходе построения ПП можно обрабатывать не по одному фактору, а по нескольку факторов подряд, если это позволит уменьшить количество перебалансировок.

Рассмотрим набор правил $\mathcal{F}_1, \mathcal{F}_2, \ldots \mathcal{F}_k$. Будем считать, что для любых двух правил $\mathcal{A}$ и $\mathcal{B}$ некоторой ПП время конкатенации равно $|h(\mathcal{A}) - h(\mathcal{B})|$. В данном допущении мы отбросили константу в оценке времени алгоритма конкатенации и считаем, что всегда реализуется его худший случай. Будем также считать, что для любой строки S высота правила, выводящего S, есть $\log|S|$. Здесь тоже кроется некоторое допущение, однако оно незначительно, поскольку высота AVL-дерева близка к двоичному логарифму от числа листьев. Мы хотим понять, в какой последовательности необходимо конкатенировать правила, чтобы общее количество перебалансировок было минимальным (в рамках описанных выше допущений).Рассмотрим функцию φ, где $\varphi(i,j)$ равно минимальному количеству перебалансировок при конкатенации правил $\mathcal{F}_i, \mathcal{F}_{i+1}, \ldots, \mathcal{F}_j$. Выпишем формулу для функции φ.

$$\varphi(i,j) = \begin{cases} 0, \text{если } i = j \\ \min_{r=i}^{j}(\varphi(i,r) + \varphi(r+1,j) + |\log(|F_i| + \cdots + |F_r|) - \\ - \log(|F_{r+1}| + \cdots + |F_j|)|), \text{иначе.} \end{cases}$$

Данную функцию можно рассчитать с помощью динамического программирования. Псевдокод для вычисления таблицы $\varphi(i,j)$ приведен ниже:

Построение таблицы $\varphi(i, j)$

```
 1  for i ← 1 to k
 2        do φ(i, i) ← 0
 3
 4  for length ← 1 to k − 1
 5        do
 6           for i ← 1 to k − length
 7                 do
 8                    j ← i + length
 9                    result ← ∞
10                    l ← 0
11                    r ← |F_i| + |F_{i+1}| + ··· + |F_j|
12                    for t ← i to j
13                          do
14                             l ← l + |F_t|
15                             r ← r − |F_t|
16                             tmp ← φ(i, t) + φ(t + 1, j) + log l − log r
17                             result ← min(result, tmp)
18
19                    φ(i, j) ← result
```

Время работы вычисления таблицы $\varphi(i, j)$, очевидно, $O(k^3)$. Опишем моднизированный алгоритм Риттера формально.

Вход: Строка T длины n, факторизация $F(T) : T = f_1 \cdot f_2 \cdot \ldots f_k$. Также для каждого фактора f_i, если он не однобуквенный, известна позиция его вхождения в строку $T_{i-1} = f_1 \cdot f_2 \cdot \ \ldots \ \cdot f_{i-1}$, то есть, известны числа l_i и t_i такие, что $f_i = T[l_i \ldots r_i]$.

Выход: ПП $\mathcal{T}$, выводящая T.

Алгоритм:

База: Построим тривиальную AVL-сбалансированную ПП $\mathcal{T}_1$ для однобуквенного фактора f_1.

Шаг:
Пусть мы построили AVL-сбалансированную ПП $\mathcal{T}_i$ для строки $T_i =$

$f_1 \cdot f_2 \cdot \ldots \cdot f_i$.

1. Возьмем факторы $f_{i+1}, f_{i+1}, \ldots, f_{i+r}$, такие что, каждый из них входит как подстрока в T_i, а f_{i+r+1} уже не входит.

2. Для каждого фактора f_{i+s} с помощью операции взятия подстроки построим ПП $\mathcal{F}_s$, выводящую f_{i+s}.

3. Для полученного набора правил вычислим таблицу $\varphi(i, j)$.

4. Применим к набору правил алгоритм конкатенации в порядке, который диктует функция $\varphi(i, j)$, то есть, другими словами, в порядке, дающем минимальное число перебалансировок. Таким образом, построим ПП $\mathcal{S}$, выводящую $F_1 \cdot F_2 \cdot \ldots F_r$

5. С помощью алгоритма конкатенации построим ПП $\mathcal{T}_{i+r} = \mathcal{T}_i \cdot \mathcal{S}$.

Примечание.
Может получиться, что в ходе алгоритма размер очередной группы будет неприемлемым для вычисления функции φ (например, сравнимым с количеством факторов в факторизации). Чтобы этого избежать, можно ограничить размер обрабатываемой группы некоторой константой. В ходе наших исследований мы использовали константу 128.

Алгоритм отличается от классического алгоритма Риттера только тем, что обрабатывает, где это возможно, несколько факторов подряд, и вычисляет порядок конкатенаций с целью уменьшения числа перебалансировок. При этом с точки зрения размера результирующей ПП данный алгоритм имеет ту же асимптотику, что и классический. Для подтверждения этого сформулируем и докажем следующую теорему.

Теорема 2.
Пусть задана строка T и некоторая ее факторизация $f_1 \cdot f_2 \cdot \ldots f_k$. Тогда модернизированный алгоритм Риттера построит ПП $\mathcal{T}$, выводящую T, размером $O(k \cdot \log |T|)$.

Доказательство: Для доказательства достаточно заметить, что при построении правила, выводящего один фактор, по-прежнему, как и в алгоритме Риттера, порождается не более $O(\log |T|)$ новых правил.

При конкатенации двух факторов порождается также не более $\log |T|$ новых правил. Всего факторов k, конкатенаций между факторами не более $k - 1$. Шаг 5 алгоритма также выполняется не более k раз. Таким образом, количество новых правил есть $O(k \cdot \log |T|)$. $\square$

5 Практические результаты

Очевидно, что природа текстов влияет на степень и время сжатия. В данной работе рассматриваются следующие типы текстов:

- **ДНК** – строки, свойства которых интересны на практике и активно изучаются;
- **случайные строки над алфавитом из 4 букв** – предположительно худший вход для алгоритмов сжатия;
- **строки Фибоначчи** – предположительно один из лучших входов алгоритмов сжатия.

В данном разделе мы приведем результаты тестирования следующих алгоритмов:

1. Алгоритм Лемпеля-Зива преобразования строки в последовательность факторов (см. раздел 3.2). Данный алгоритм мы будем обозначать **lz77**.
2. Алгоритм Риттера преобразования последовательности факторов в прямолинейную программу (см. раздел 4.4). Обозначение – **SLPClassic**.
3. Модернизированный алгоритм Риттера преобразования последовательности факторов в прямолинейную программу (см. раздел 4.5). Обозначение – **SLPnew**.

Для изображения результатов работы алгоритмов на графиках мы будем использовать следующие обозначения:

■ – lz77
○ – SLPclassic
● – SLPnew

Под производительностью алгоритма мы будем понимать два параметра: время выполнения и *относительный размер*. Для алгоритмов построения ПП мы будем также считать количество перебалансировок. *Относительный размер* рассчитывается как отношение размера сжатого представления к длине исходного текста. Для алгоритма Лемпеля-Зива под размером сжатого представления будем понимать количество факторов в результате, для алгоритмов ПП – размер получившейся прямолинейной программы. Наше понимание относительного размера может

показаться непривычным, поскольку оно не учитывает, что для хранения как фактора, так и правила, требуется значительно больше памяти, чем для хранения одного символа строки. Но в данном случае это не важно, поскольку нас интересует зависимость между длиной входной строки и размерами сжатых представлений.

5.1 Сравнение алгоритмов построения ПП

В данной главе мы приведем подробное сравнение двух алгоритмов построения прямолинейной программы.

Оба алгоритма, как и ожидалось, работают очень быстро на строках Фибоначчи. Например, на 36 строке Фибоначчи длиной около $36.9 \cdot 10^6$ оба алгоритма укладываются в 1 миллисекунду и строят ПП размером 100. Данный тест доказывает, что существуют строки, для которых модель сжатия с помощью прямолинейных программ является очень эффективной.

На следующем графике для обоих алгоритмов представлена зависимость количества перебалансировок от размера исходного текста.

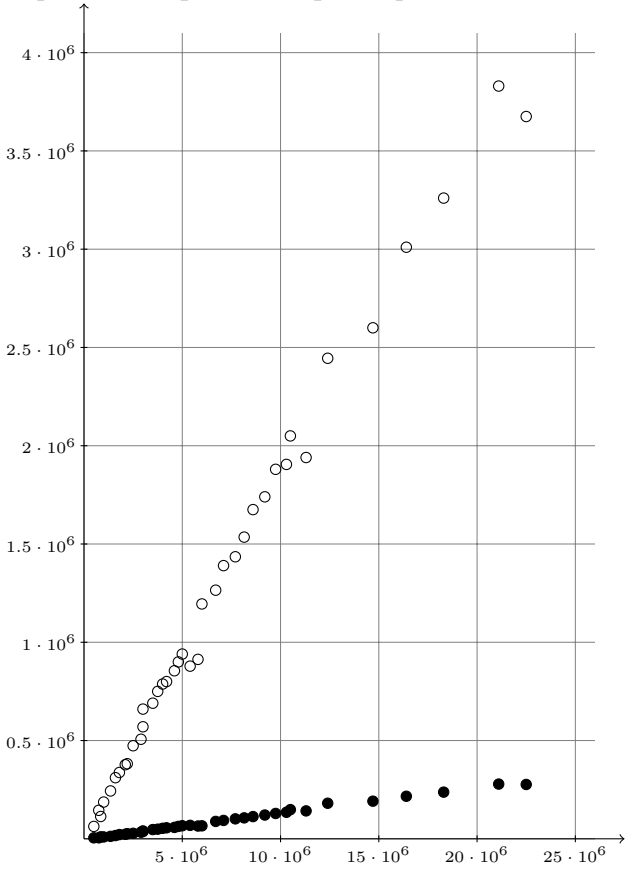


Рисунок 3. Количество операций перебалансировки в процессе построения ПП для строк ДНК.

По горизонтали отмечены длины входных строк, по вертикали – количество операций перебалансировки.

Для оценки того, насколько уменьшение количества перебалансировок влияет на реальное время выполнения алгоритма, мы представим два теста. В первом тесте AVL-дерево хранится в оперативной памяти. Во втором тесте AVL-дерево хранится во внешней памяти. То есть, каждая операция перебалансировки порождает обращение к жесткому диску. Ниже представлены графики зависимости времени выполнения алгоритма от размера исходного текста.

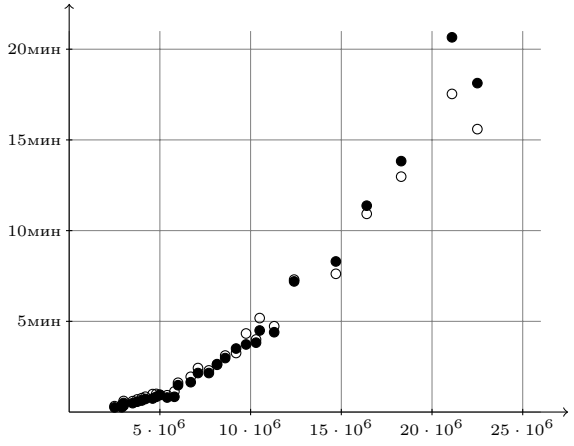


Рисунок 4. Время выполнения алгоритма на строках ДНК (AVL-дерево хранится в оперативной памяти).

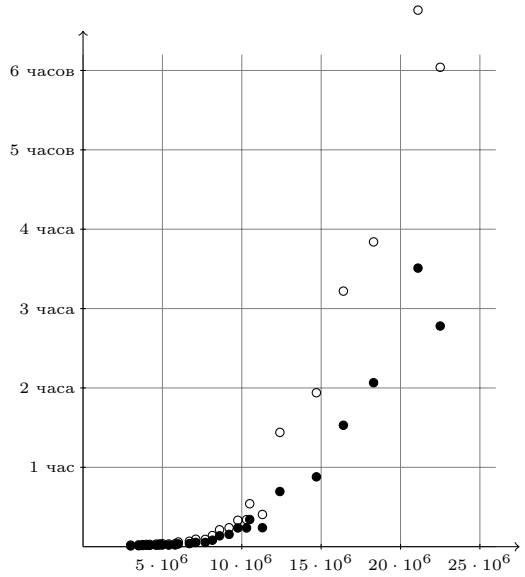


Рисунок 5. Время выполнения алгоритма на строках ДНК (AVL-дерево хранится во внешней памяти).

На следующих трех графиках представлено сравнения алгоритмов построения ПП для случайных строк.

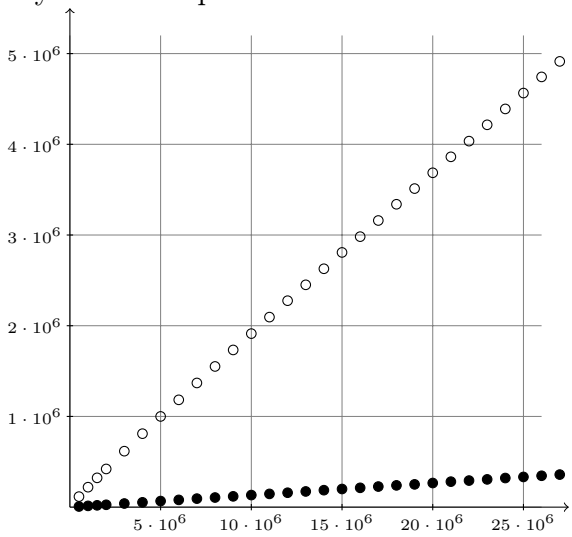


Рисунок 6. Количество операций перебалансировки в процессе построения ПП для случайных строк.
По горизонтали отмечены длины входных строк, по вертикали – количество операций перебалансировки.

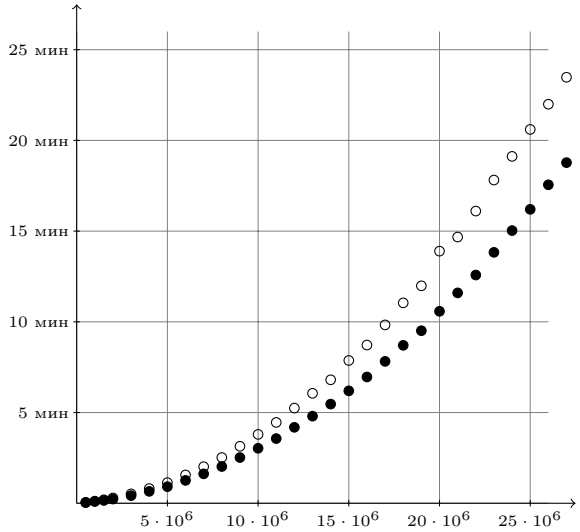


Рисунок 7. Время выполнения алгоритма на случайных строках (AVL-дерево хранится в оперативной памяти).

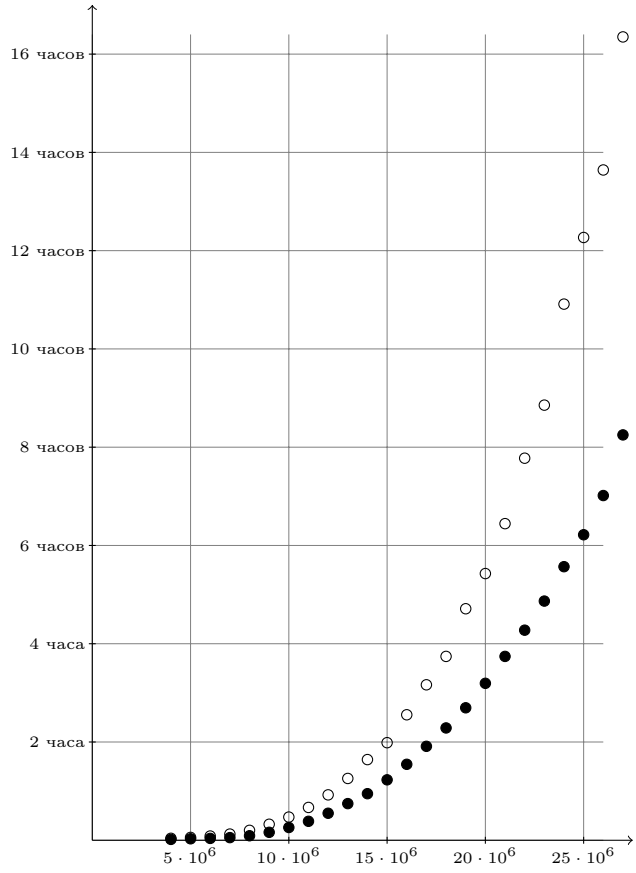


Рисунок 8. Время выполнения алгоритма на случайных строках (AVL-дерево хранится во внешней памяти).

Приведенные графики показывают, что модернизированный алгоритм Риттера имеет ту же производительность, что и классический, если дерево хранится в оперативной памяти. Когда же дерево хранится во внешней памяти, то новый алгоритм становится в среднем в 2 раза быстрее.

5.2 Результаты по относительному размеру

Ниже приведены два графика, сравнивающие относительные размеры.

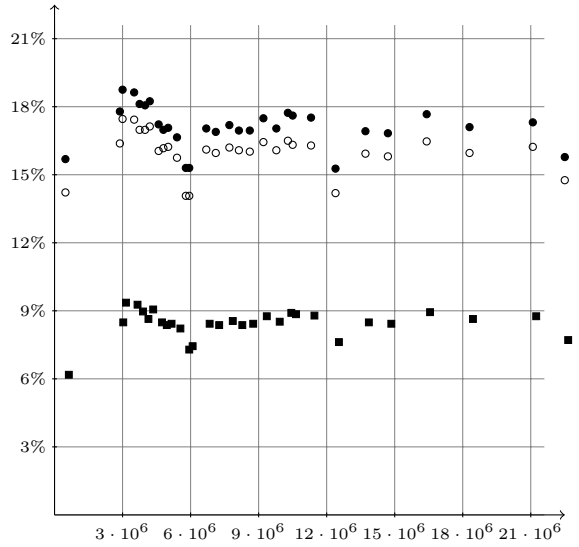


Рисунок 9. Результаты по относительному размеру для строк ДНК.

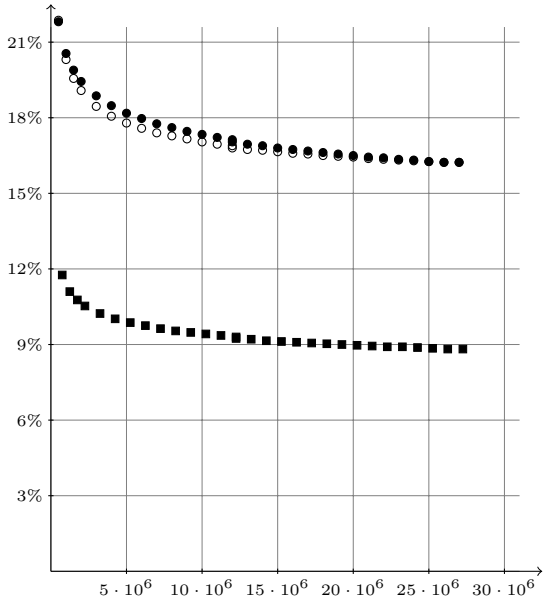


Рисунок 10. Результаты по относительному размеру для случайных строк.

Из графиков видно, что относительные размеры классического и модернизированного алгоритмов Риттера практически одинаковы. Для строк

небольшой длины новый алгоритм может сгенерировать несколько большую ПП. Это объясняется тем, что вначале новый алгоритм многократно конкатенирует небольшие по размеру ПП, порождая новые правила вывода. Также, анализируя графики, можно сделать интересное наблюдение: отношение размеров ПП к количеству факторов, генерируемых алгоритмом Лемпеля-Зива, ведет себя как константа, близкая к значению 2. Мы пока не можем дать объяснения этому наблюдению.

6 Заключение

В данной работе мы представили практическую реализацию алгоритма Лемпеля-Зива, построенную на структуре данных суффиксный массив и более удобную для использования на практике в случае, когда не хватает оперативной памяти.

Также мы представили алгоритм построения прямолинейных программ. Алгоритм построен на тех же идеях, что и классический алгоритм Риттера, но является более эффективным на практике с точки зрения производительности. При этом в новом алгоритме остается ряд открытых вопросов. В частности, не дана его теоретическая оценка сложности. Новый алгоритм совершает гораздо меньшее количество операций перебалансировки, однако это количество не оптимальное.

В последней главы мы сравнили классический и модернизированный алгоритмы Риттера. Сравнение показало, что ПП, генерируемые данными алгоритмами, имеют практически одинаковые размеры. Также наши исследования показали, что отношение размеров ПП к размеру факторизации не меняется с увеличением длины исходной строки. Возможно, теоретические оценки размера ПП, генерируемой данными алгоритмами, являются оценками сверху и могут быть улучшены.

Список литературы

[1] *P. Cegielski, I. Guessarian, Y. Lifshits and Y. Matiyasevich* Window sunsequence problems for compressed texts. In Proceedings of 1st International Symposium Computer Science in Russia (CSR 2006), pp. 127-136. Springer-Verlag, 2006

[2] *Maxime Crochemore and Wojciech Rytter.* Text Algorithms. Oxford University Press, New York, 1994.

[3] *Martin Farach and Mikkel Thorup* String matching in Lempel-Ziv compressed strings. In Proceedings of the 27-th Annual ACM Symposium on Theory of Computing (STOC 1995), pp. 703-712, ACM Press, 1995.

[4] *Michael Garey and David Johnson* Computers and Intractability: a Guide of the Theory of NP-completness. Freeman, 1979.

[5] *L. Gasieniec, M. Karpinski, W. Plandowski and W.Rytter* Efficient algorithms for Lempel-Ziv encoding (extended abstract). In Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996), pp. 392-403. Springer-Verlag, 1996.

[6] *Dan Gusfield* Algorithms on Strings, Trees and Sequences. Cambridge University Press, 1997.

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

[8] *Y. Ishida, S. Inenaga, A. Shinohara, M. Takeda* Full incremental LCS computation. In Proceedings of Fundamentals of Computation Theory (FCT 2005), pp. 563-574, Springer-Verlag 2005.

[9] *Juha Karkkainen, Peter Sanders, Stefan Burkhardt* Linear Work Suffix Array Construction. Journal of the ACM (JACM), Volume 53 Issue 6, November 2006.

[10] *Marek Karpinski*, *Wojciech Rytter*, *Ayumi Shinohara* An efficient pattern-matching algorithm for strings with short descriptions. Nordic Journal of Computing, pp. 172-186, 1997.

[11] *Toru Kasai*, *Gunho Lee*, *Hiroki Arimura*, *Setsuo Arikawa*, *Kunsoo Park* Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. CPM '01 Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, pp. 181-192. Springer-Verlag, 2001.

[12] *Lesha Khvorost* Seraching All Pure Squares. In Proc. Plenary Conference AutoMathA, 2009.

[13] *T. Kida*, *T. Matsumoto*, *Y. Shibata*, *M. Takeda*, *A. Shinohara*, *S. Arikawa* Collage system: a unifying framework for compressed pattern matching. Theoretical Computer Science Journal, pp. 253-272, 2003.

[14] *Donald Knuth* The Art of Computing Vol.II: Seminumerical Algorithms. Second Edition. Addison-Wesley 1981.

[15] *Yury Lifshits and Markus Lohrey* Quering and embedding compressed texts. In Proceeding of International Symposium on Mathematical Foundations of Computer Science (MFCS 2006), pp. 681-692. Springer-Verlag, 2006.

[16] *Yury Lifshits*. Processing Compressed Texts: A Tractability Border, In Proceedings of Symposium on Combinatorial Pattern Matching (CPM 2007), pp. 228-240. Springer 2007.

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

[18] *Markus Lohrey* Word problems and membership problems on compressed words. Society of Industrial and Applied Mathematics Journal (SIAM), pp. 1210-1240, 2006.

[19] *W. Matsubara*, *S. Inenega*, *A. Ishino*, *A. Shinohara*, *T. Nakamura*, *K. Hashimoto* Computing Longest Common Substring and All Palindromes from Compressed Strings. In Proceeding of Software Seminar (SOFSEM 2008) 2008.

[20] *Masamichi Miyazaki*, *Ayumi Shinohara*, *and Masayuki Takeda* An improved pattern matching algorithm for strings in terms of straight line programs. In Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997), pp. 1-11. Springer-Verlag, 1997.

[21] *Wojciech Plandowski* Testing equivalence of morphisms on context-free languages. In Proceedings of Second Annual European Symposium on Algorithms (ESA 1994), pp. 460-470. Springer-Verlag, 1994.

[22] *Wojciech Rytter* Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science Journal, pp. 211-222, 2003.

[23] *Y. Shibata*, *M. Takeda*, *A. Shinohara*, *S. Arikawa* Pattern matching in texts compressed by using antidictionaries. In Proceeding of Symposium on Combinatorial Pattern Matching (CPM 1999), pp. 37-49. Springer-Verlag, 1999.

[24] *Jacob Ziv and Abraham Lempel* A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, pp. 337-343, 1977.