Содержание

1 Введение

Стремительный рост объемов доступной информации существенно увеличивает сложность поисковых задач. Для того чтобы преодолеть эту трудность можно по-новому взглянуть на способы хранения и представления данных. С одной стороны, классические поисковые алгоритмы ориентированы на явное представление данных и поэтому слабо устойчивы к росту входных данных. С другой стороны, алгоритмы сжатия данных (например, алгоритмы семейства Лемпеля-Зива) не поддерживают поисковые запросы без предварительной распаковки.

В течение последних 15 лет стала развиваться идея сжатых представлений, поддерживающих поисковые запросы. Одним из таких представлений являются прямолинейные программы (ПП). Для данного сжатого представления были найдены полиномиальные алгоритмы решающие некоторые классические поисковые задачи: на вход такой алгоритм принимает сжатые представления и, соответственно, сложность алгоритма оценивается в зависимости от размера этих представлений. Для следующих задач существует полиномиальный алгоритм на ПП: поиск сжатого шаблона в сжатом текста[1], поиск всех палиндромов и набольшей общей подстроки[15].

В работе рассмотривается задача о поиске сжатого шаблона в сжатом тексте. На вход алгоритму подается ПП размера n, построенная для текста, и ПП размера m, построенная для шаблона, на выходе – список вхождений шаблона в текст. Первый алгоритм, решающий данную задачу, был предложен в работе [1]. Чуть позже в работе [2] был предложен более эффективный алгоритм работающий за $O((n+m)^4 \cdot log(n+m))$ использующий $O((n+m)^3)$ памяти. Однако данные алгоритмы нашли лишь некоторые, но не все вхождения шаблона в текст. В работе [3] был предложен алгоритм находящий все вхождения шаблона в текст, работающий за $O(n^2 \cdot m^2)$ и использующий $O(nm)$ памяти. Последнее известное улучшение было представленно в работе [4] Юрия Лифшица – асимптотика времени работы была уменьшена до $O(n^2 \cdot m)$.

1.1 Решаемые задачи

Исследование вопроса об эффективности способа обработки данных с помощью ПП представляет особый интерес. В работе центральное внимание уделено практической применимости алгоритма Лифшица. Работа отвечает на вопрос, может ли алгоритм Лифшица быть эффективнее классических алгоритмов поиска шаблона в тексте и где находится эта граница. В рамках данной работы был реализован алгоритм Лифшица и выявлены узкие места с практической точки зрения. Также предложена эвристика позволяющая запускать алгоритм во многопоточном режиме. В итоге был проведен сравнительных анализ параллельной версии алгоритма Лифшица с алгоритмом Кнута-Морриса-Пратта.

1.2 Структура работы

В работе производится обзор алгоритма Лифшица, полное его описание можно найти в статье [4]. Далее представлена базовая реализация алгоритма и ряд улучшений, которые позволяют уменьшить размер потребляемой алгоритмом памяти и ускорить скорость его работы, предложена параллельная версия алгоритма. Также в работе приводится сравнение базовой и улучшенной реализации алгоритма на практике. Для построения ПП используется улучшенная версия алгоритмов Риттера, представленная в работе [13]. Сравнение проводится на ПП, которые были получены из следующий типов строк:

- ДНК
- Случайные строки с различной мощностью алфавита

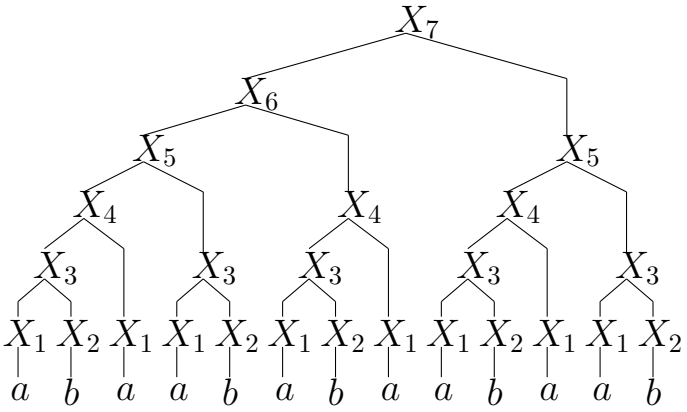
2 Основные определения

Прямолинейная программа (ПП) — это контекстно-свободная грамматика, которая состоит из правил вида: $X_i \to a$, где a – терминал, или $X_i \to X_j X_k$, где $j, k < i$.

Пример: Рассмотрим ПП для текста «*abaababaabaab*»:

$$X_1 \to a, X_2 \to b, X_3 \to X_1 \cdot X_2, X_4 \to X_3 \cdot X_1,$$
$$X_5 \to X_4 \cdot X_3, X_6 \to X_5 \cdot X_4, X_7 \to X_6 \cdot X_5$$

Восстановление этого текста из ПП можно проиллюстрировать в виде дерева на рис. 2



Для удобства изложения используется обозначение X_k как для символа грамматики, так и для соответствующего текста, полученного при распаковке X_k. Таким образом, можно считать, что $T = X_m$. *Позицией* в строке называется место между любыми двумя соседними символами, а также перед первым и за последним символами строки. Таким образом, в строке из n символов есть позиции $0, 1, \ldots, n$. Будем говорить, что подстрока *касается* некой позиции в тексте, если эта позиция лежит внутри или на границе подстроки. Термин *вхождение* используется для обозначения начальной позиции подстроки в тексте. Для нетерминалов разрезом называется позиция на стыке правил, а для нетерминалов – начальная позиция, то есть 0.

Проиллюстрируем на примере рис. 2 данные определения. Разрез строки «*abaab*», выводящийся из нетерминала X_5, находится на третьей позиции —

«$aba|ba$». Есть два вхождения строки «ab» в строку «$ababa$» — на позициях 1 и 3, причем второе касается разреза, а первое – нет. Строка «ba» также входит в «$ababa$» дважды, причем оба вхождения касаются разреза.

3 Обзор алгоритма

Приведем описание алгоритма Лифшица, останавливаясь подробно на деталях, важных для анализа алгоритма. Полное рписание алгоритма можно найти в работе [4]. Алгоритм получает на вход две прямолинейные программы T и P. Результатом работы является список всех вхождений образца в текст. Заметим, что хранить позиции вхождений P в T в явном виде не представляется возможным: при поиске строки вида a^k в строке вида a^p число вхождений будет экспоненциально относительно размера сжатого текста. Авторы работ [4] и [6] используют для хранения информации о позициях вхождений арифметические прогрессии, основываясь при этом на следующем теоретическом факте:

Лемма 3.1. *Вхождения образца P в текст T, касающиеся некоторой фиксированной позиции, образуют одну арифметическую прогрессию.*

Алгоритм Лифшица основан на построении таблицы арифметических прогрессий, которая определяется следующим образом:

- Для всех пар чисел $1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$ значение $A[i, j]$ задает арифметическую прогрессию вхождений P_i в T_j, касающихся разреза T_j.

Заметим, что хранить арифметическую прогрессию можно в виде трех чисел – начального элемента, разности и количества элементов в прогрессии.

В работе [4] сформулированы и доказаны следующие утверждения:

Утверждение 3.1. *Используя таблицу арифметических прогрессий можно решить задачу поиска подстроки в строке за $O(n)$.*

Утверждение 3.2. *Вычислить таблицу арифметических прогрессий можно за $O(n^2 m)$ с помощью метода динамического программирования.*

Напомним, что здесь n и m — размеры прямолинейных программ, порождающих текст и шаблон соответственно.

Построение таблицы арифметических прогрессий состоит из трех этапов.

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 $A[i, j]$:
   (a) Находим вхождения большей компоненты шаблона P_i вокруг разреза текста T_j
   (b) Находим вхождения меньшей компоненты шаблона P_i, которые стыкуются с уже найденными вхождениями большей половины
   (c) Пересекаем вхождения большей и меньшей компонент, приводим ответ к одной прогрессии

3.1 Вычисление (i, j) ячейки

Пусть $P_i = P_r P_s$. Обозначим за γ разрез T_j. Для вычисления $A[i, j]$ мы используются уже вычисленные значения $A[r, 1], \ldots, A[r, j]$ и $A[s, 1], \ldots, A[s, j]$. Другими словами, требуется информация о вхождении правой и левой части шаблона в текущий текст и все предшествующие тексты. Будем для удобства считать, что длина строки P_f не меньше, чем длина строки P_s.

1. Найдем все вхождения P_f в T_j, которые могут быть началом интересующего нас вхождения.
2. Найдем вхождения P_s в T_j, которые начинаются в позициях концов вхождений P_f.
3. Пересечем полученные множества вхождений и запишем их в виде одной арифметической прогрессии вхождений P_i в T_j.

Для поиска вхождений P_f в T_j и P_s в T_j будем использовать процедуру LocalSearch. Процедура $LocalSearch(i, j, [a, b])$ возвращает вхождения P_i в T_j , лежащие целиком внутри интервала $[a, b]$. Важные характеристики алгоритма локального поиска:

1. Он корректно работает лишь при условии $|b-a| <= 3|P_i|$.
2. Локальный поиск использует значения $A[i,k]$ для $1 <= k <= j$.
3. Его трудоемкость O(j) шагов
4. На выходе локальный поиск выдает пару арифметических прогрессий, причем внутри каждой прогрессии все вхождения имеют общую точку, и все вхождения из первой прогрессии находятся левее всех вхождений из второй.

Для вычисления $A[i,j]$ нужно 5 запусков локального поиска.

1. С помощью одного вызова локального поиска мы найдем все вхождения P_f в интервале $[\gamma - |P_i|, \gamma + |P_f|]$, длина которого $|P_i| + |P_f| <= 3|P_f|$. В ответе получаются две арифметические прогрессии, представляющие все потенциальные стартовые позиции для вхождений P_i, касающихся разреза. Далее подлежат поиску только те вхождения P_s, которые начинаются с позиций из двух арифметических прогрессий окончаний вхождений P_f.
2. Рассмотрим каждую прогрессию по отдельности. Назовем точку окончания вхождения P_f континентальной, если она находится на расстоянии хотя бы $|P_s|$ от последнего окончания в прогрессии. Все остальные окончания будем называть приморскими. Для поиска вхождений в приморском районе применяется локальный поиск P_s в $|Ps|$–окрестности последнего окончания и пересечем ответ с прогрессией приморских окончаний вхождений P_r. Для проверки первого континентального окончания применятся локальный поиск к $|P_s|$–строке, начинающейся с этого окончания.
3. В зависимости от результатов проверки, возьмем или все континентальные окончания, или ни одно из них. Возьмем подпрогрессию приморских окончаний (тех, которые соответствуют началам вхождений P_s). Также возьмем аналогичные множества для второй прогрессии вхождений P_r. Эти четыре части могут быть упрощены упростить до одной

прогрессии по лемме 3.1, и эта прогрессия будет результатом вычисления (i, j) ячейки.

3.2 Процедура LocalSearch

Локальный поиск находит вхождения P_i в подстроке $T_j[a, b]$. Он состоит из двух этапов: процедуры обхода и процедуры упрощения.

`Процедура обхода`. На входных данных $(i, j, [a, b])$ производятся следующие действия.

1. Из таблицы прогрессий берутся вхождения P_i в T_j, которые касаются разреза.
2. Обрезается прогрессию этих вхождений, остаются лишь находящиеся целиком внутри $[a, b]$.
3. Усеченная прогрессия записывается в список ответов.
4. Проверяется, имеет ли пересечение $[a, b]$ с левой/правой частью T_j длину хотя бы $|P_i|$.
5. Если да, то делается рекурсивный вызов процедуры обхода с тем же i, индексом левой/правой части T_j, и интервалом пересечения.

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

`Процедура упрощения`. В процедуре склеивается полученный список не более, чем в две прогрессии. Данные факт доказан в работе [4].

	T_1 «a»	T_2 «b»	T_3 «ba»	T_4 «aba»	T_5 «$ababa$»
P_1 «a»	(1, 1, 1)	-	(2, 1, 1)	(1, 1, 1)	(3, 1, 1)
P_2 «b»	-	(1, 1, 1)	(1, 1, 1)	(2, 1, 1)	(4, 1, 1)
P_3 «ab»	-	-			
P_4 «aba»	-	-			

Рис. 1: Вычислены значения ячеек, соответствующих однобуквенным текстам

Из-за рекурсивности процедуры LocalSearch оценка глубины рекурсии так же представляет интерес. Не трудно заметить, что глубина рекурсии зависит от сбалансированности ПП T_j и ее можно оценить высотой ПП. Соответственно, в случае сбалансированной T_j оценка высоты – $O(log|T_j|)$, иначе – $O(|T_j|)$. В случае несбалансированной ПП использование рекурсивного локального поиска на практике не представляется возможным и возникает потребность итеративного алгоритма.

3.3 Пример работы

Рассмотрим работу алгоритма на примере. Пусть текст «$ababa$» представлен следующей ПП:

$$T_1 \to a, T_2 \to b, T_3 \to T_2 \cdot T_1, T_4 \to T_1 \cdot T_3, T_5 \to T_4 \cdot T_3$$

Будем искать в этом тексте образец «aba»

$$P_1 \to a, P_2 \to b, P_3 \to P_1 \cdot P_2, P_4 \to P_3 \cdot P_1$$

Тройка (f, d, s) в ячейке таблицы означает арифметическую прогрессию с началом в позиции f, разностью d и состоящую из s элементов. Для простоты будем считать, что у одноэлементных арифметических прогрессий $d = 1$. Символ «$-$» в ячейке $APTable[i, j]$ означает отсутствие вхождений P_i в T_j, касающихся разреза.

После вычисления ячеек таблицы, соответствующих однобуквенным текстам, таблица арифметических прогрессий выглядит следующим образом:

	T_1 «a»	T_2 «b»	T_3 «$b\|a$»	T_4 «$a\|ba$»	T_5 «$aba\|ba$»
P_1 «a»	(1, 1, 1)	-	(2, 1, 1)	(1, 1, 1)	(3, 1, 1)
P_2 «b»	-	(1, 1, 1)	(1, 1, 1)	(2, 1, 1)	(4, 1, 1)
P_3 «ab»	-	-	-	(1, 1, 1)	(3, 1, 1)
P_4 «aba»	-	-	-	(1, 1, 1)	

Рис. 2: Вид таблицы перед вычислением $APTable[4, 5]$

На следующем этапе построения таблицы заполняются остальные ячейки. Подробнее рассмотрим процесс вычисление значения в ячейке $APTable[4, 5]$. Она соответствует вхождениям образца «aba» в текст «$ababa$». Перед началом вычисления значения ячейки таблица выглядит так:

Образец «aba» получен из правила $P_4 \to P_3 \cdot P_1$, где из P_3 выводится строка «ab», а из P_1 – строка «a». Так как $|P_3| > |P_1|$ сначала будем искать в тексте вхождения P_3. Для этого запустим процедуру `LocalSearch`$(3, 5, [1, 5])$. Рассмотрим процесс выполнения этой процедуры.

Сначала получим вхождения P_3, которые касаются разреза T_5. Чтобы получить арифметическую прогрессию, соответствующую этим вхождениям, нужно просто обратиться к ячейке $APTable[3, 5]$, которая была вычислена ранее. Таким образом, получаем прогрессию $(3, 1, 1)$. Далее из полученной прогрессии удаляем элементы, соответствующие вхождениям P_3, которые не содержатся в отрезке $[1, 5]$. В данном случае таких элементов нет, и прогрессия остается неизменной.

Далее следует поиск вхождения P_3 в T_5 на отрезке $[1, 5]$, не касающиеся разреза. Для этого рекурсивно запускается процедура `LocalSearch` для образца P_3 и каждого из текстов T_4 и T_3, так как T_5 получено из правила $T_5 \to T_4 \cdot T_3$. При этом границы отрезка поиска также будут меняться.

Рекурсивные вызовы выглядят следующим образом: `LocalSearch`$(3, 4, [1, 3])$ и `LocalSearch`$(3, 3, [1, 2])$.

Первый вызов ищет арифметическую прогрессию $(1, 1, 1)$ из ячейки $APTable[3,$ В результате второго вызова новых вхождений не найдено. Дальнейших рекурсивных вызовов также нет из-за того, что отрезки поиска стали короче

образца, который мы ищем.

Таким образом, найдены две арифметические прогрессии вхождений P_3 в T_5, которые объединяются в одну $(1, 2, 2)$. Для каждого из вхождений теперь нужно проверить, можно ли дополнить его до вхождения P_4 в T_5, то есть существует ли вхождение P_1, начинающееся в позиции следующей за концом вхождения P_3. Эти проверки так же производятся с помощью процедуры `LocalSearch`. Вызовы выглядят так: `LocalSearch`$(1, 5, [3, 3])$ и `LocalSearch`$(1, 5, [5, 5])$

В итоге получаем $APTable[4, 5] = (1, 2, 2)$

4 Анализ алгоритма

В разделе детально рассматривается алгоритм Лифшица. Особое внимание уделяется скорости работы и использованию оперативной памяти. Структура раздела такова, что сначала рассматривается найденная проблемная часть, предлагается улучшение и проводится сравнение до и после для того, чтобы увидеть влияние улучшения на скорость работы и потребление памяти алгоритма. Для реализации алгоритма Лифшица был использован язык программирования Java. Все результаты получены на компьютере с четырех ядерным процессором Intel i5(учитывая Hyper-Threading), 16 гигабайтами оперативной памяти, Windows 7 x64 и JDK 7. Для построения прямолинейных программ использовался улучшенный алгоритм Риттера [13]. Исходный код реализации алгоритма Лифшица, построения ПП и сравнений, можно найти в репозитории http://overclocking.googlecode.com. Сравнения проводились на ДНК из открытого банка Японии(http://www.ddbj.nig.ac.jp/) и случайных текстах с алфавитом различной мощности. Согласно использованной методологии сравнения, запуск алгоритма происходит в несколько этапов:

- Построение ПП по тексту.
- Выбор случайной подстроки зафиксированного размера из текста под шаблон и построение ПП по нему.
- Запуск алгоритма на подготовленных ПП текста и шаблона. Именно это время считается за время работы алгоритма, исключая время на построение ПП. Так же запуск производится десять раз, среди всех значений отбирается медиана. Это необходимо для устранения выбросов по времени, которые могут происходить из-за различных факторов. Формат описания таблиц с результатами фиксируется: количество правил в ПП для текста T обозначается как |PP(T)|, а количество символов в тексте T – |T|. Также приведенные в качестве T названия являются названиями файлов из банка ДНК Японии. Приведем использованные файлы и их размеры: AACR – 55 кб, AAEZ – 77 кб, AAES – 68 кб, AAZV – 258 кб.

|PP(T)|	|PP(P)|	Размер таблицы(КБ)
10^3	10	280
10^3	10^2	2.800
10^4	10^2	28.000
10^4	10^3	280.000
10^5	10^3	2.800.000
10^6	10^3	28.800.000

Рис. 3: Зависимость размера таблица от размера грамматики шаблона и текста

4.1 Таблица прогрессий

4.1.1 Первый подход

Базовой структурой данных для алгоритма является таблица прогрессий, размер которой – N * M, где N и M количество строк и столбцов соответственно. С точки зрения автора работы, наиболее естественное представление таблицы – двумерный массив. Достоинствами этого представления являются простота и доступ за O(1) к (i, j) ячейки. К недостаткам можно отнести размер потребляемой памяти – N * M. Напомним, что количество строк и столбцов в таблице прогрессий определяется количеством правил в грамматики шаблона и текста соответственно. Проиллюстрируем количество потребляемой памяти для таблицы прогрессий. Напомним, что прогрессия описывается тремя числами. Для ее хранения используется класс с 3-мя int-ми, размер каждого равен 4 байтам, однако суммарный размер объекта данного класса больше. Для вычисления размера объекта арифметический прогрессий была использована статья с openjdk.java.net(http://openjdk.java.net/projects/code-tools/jol/), получен размер прогрессии 24 байта. Размер одной ссылки на прогрессию – 4 байта.

Используя эти размеры, были получены следующие результаты для размера таблицы прогрессий:

Анализ полученных результатов показывает, что наблюдаемый быстрый рост требуемого объема памяти согласуется с теоретической оценкой размера таблицы и влиянием константы равной размеру прогрессии. Более того,

|PP(T)|	|PP(P)|	Разреженность
10^3	10	80.38
10^3	10^2	95.30
10^4	10^2	94.99
10^4	10^3	98.53
10^5	10^3	98.52
10^6	10^3	98.44

Рис. 4: Процент пустоты

из-за быстрого роста потребляемой памяти, алгоритм становится малопригодным к использованию на ПП больше 10^6 правил(для случайного текста с алфавитом мощности 4 – это строка около 5мб). Стоит отметит, что при росте размера прямолинейных программ вполне возможна ситуация, когда для описания прогресии необходимо использование числа большего размера (применительно к Java – числа типа long), что увеличивает размер прогрессии и таблицы в целом. Можно считать, что размер прогрессии увеличится вдвое (данный факт остается вне проводимого исследования).

4.1.2 Разреженность

Однако, запустив реализацию алгоритма на текстах приведенного выше размера, было обнаружено, что памяти потребляется меньше. Данный факт сигнализировал о неравномерном заполнении ячеек таблицы. Был проверен процент пустоты таблицы на практике, используя упомянутые размеры шаблонов и текстов. Тесты были проведены на случайных текстах с 4х буквенным алфавитом.

Оказалось, что большая часть ячеек таблицы – пустые. Также были выявлены следующие закономерности:

- При увеличении размера грамматики шаблона при фиксированном размере грамматики текста процент пустоты увеличивается.
- При увеличении размера алфавита при зафиксированном размере грамматики шаблона и текста процент пустоты увеличивается.

T	|P|	Разреженность
AACR	1/100	95.04
AACR	1/200	93.06
AACR	1/500	89.19
AACR	1/1000	85.37
AAEZ	1/100	95.46
AAEZ	1/200	93.28
AAEZ	1/500	89.36
AAEZ	1/1000	85.84
AAES	1/100	96.72
AAES	1/200	95.48
AAES	1/500	93.04
AAES	1/1000	90.28
AAZV	1/100	97.60
AAZV	1/200	96.67
AAZV	1/500	94.63
AAZV	1/1000	92.40

Рис. 5: Процент пустоты при фиксированном размере грамматики текста и меняющемся размере грамматики шаблона

|PP(T)|	|PP(P)|	Мощность алфавита	Разреженность
1000	10	2	60.47
1000	10	4	76.06
1000	10	8	88.02
1000	10	16	94.20
10000	100	2	86.29
10000	100	4	94.13
10000	100	8	96.01
10000	100	16	98.29
100000	1000	2	95.74
100000	1000	4	98.67
100000	1000	8	99.37
100000	1000	16	99.55

Рис. 6: Процент пустоты при фиксированном размере грамматики текста и шаблона при увеличеннии алфавита

|PP(T)|	|PP(P)|	Размер таблицы(КБ)
10^3	10	8.8
10^3	10^2	52
10^4	10^2	4.100
10^4	10^3	43.600
10^5	10^3	436.000
10^6	10^3	4.360.000

Рис. 7: Зависимость размера таблица от размера грамматики шаблона и текста

Основываясь на факте разреженности данных в таблице и полученном проценте пустоты в каждом из случаев, можно высчитать новые оценки по размеру таблицы прогрессий:

Один из других способов представить таблицу прогрессий – с помощью ассоциативного массива, где ключ является парой (i, j). Доступ к ячейке таблицы осуществляется за O(1) в среднем. Представление с помощью ассоциативного массива позволяет уменьшить объем потребляемой памяти в случае, если таблица разрежена. Для оценки размера, занимающего ассоциативный массив, необходимо рассмотреть детали его реализации. В стандартной библиотеке Java 7 есть несколько реализаций ассоциативного массива, в работе рассматривается HashMap. Реализация основана на хеш-таблице с методом цепочек в качестве разрешения коллизий(с ней можно ознакомиться http://en.wikipedia.org/wiki/Hash_table). Каждый узел такой хеш-таблицы можно описать следующим образом: ссылка на ключ, ссылка на значение, ссылка на следующий узел (используется для разрешения коллизий), где ключ – класс, представляющий пару (i, j), а значение – прогрессия. Используя предложенный выше способ измерения размера объекта, получаем, что размер узла хеш-таблицы равен 32 байт. Оцениваем размер ассоциативного массива с N элементами по формуле: $(32 * nodeSize + 24 * progressionSize * N + power) * referenceSize$, где nodeSize – размер узла, progressionSize – размер прогресси, power – наименьшая привосходящяя N степень двойки, referenceSize – размер ссылки. Наименьшая привосходящяя N степень двойки была выбрана для оценки размера массива внутри

|PP(T)|	|PP(P)|	N и выбранная степень 2	Размер таблицы(КБ)
10^3	10	2000|2048	120
10^3	10^2	5000|8192	313
10^4	10^2	15000|32768	971
10^4	10^3	150000|262144	9.448
10^5	10^3	1500000|2097152	92.388
10^6	10^3	15000000|33554432	974.217

Рис. 8: Зависимость размера таблица от размера грамматики шаблона и текста

хеш-таблицы.

Использование таблицы прогрессий, представленной с помощью ассоциативного массива, позволяет сэкономить память до 5 раз и использовать расматриваемый алгоритм на ПП большего размера (по крайней мере до 10^6 правил) грамматиках без опасений о недостатке памяти. Стоит отметить, что HashMap реализован для абстрактных ключей, поэтому необходим объект для доступа к (i, j) ячейке, который приходится создавать для каждой операции записи-чтения, что вносит дополнительные накладные разходы. Оптимизация хеш-таблицы под ключ (i, j) с помощью внесения их непосредственно в ее узел для уменьшения его суммарного размера позволяет ускорить алгоритм и уменьшить размер таблицы.

4.2 Локальный поиск

Вторая базовая часть алгоритма -– локальный поиск, который используется для вычисления каждой из (i, j) ячейки таблицы прогрессий. Более того, в процессе вычисления ячейки локальный поиск может быть вызван до пяти раз. Именно этот факт говорит о необходимости рассмотрения данной части алгоритма.

4.2.1 Упрощение первоначальной версии локального поиска

При реализации локального поиска оказалось, что небольшое изменение алгоритма упрощает его реализацию. В описанном выше локальном поиске

предлагается хранить полученные прогрессии в двунаправленном списке и сдвиг относительно позиции в этом списке. Размер полученного списка прогрессий будет пропорционален размеру поддерева грамматики текста, на котором вызывается локальный поиск. Для слияния списка прогрессий необходимо выполнить количество операций пропорционально длине списка. Оказалось, что хранить список не обязательно, достаточно лишь двух прогрессий (далее результат локального поиска). Механика работы с результатом локального поиска следующая:

- При добавлении прогрессий мы пытаемся соединять их сначала с первой прогрессией результата.
- После того, как мы обнаружили факт того, что прогрессии соединить нельзя, начинаем соединять со второй прогрессией результата.

Списка и указателя на текущий элемент теперь нет, но необходимо сохранить порядок обхода. Для этого немного изменяется порядок рекурсивных вызовов. Окончательный вариант выглядит следующим образом:

- Берем (из таблицы прогрессий) вхождения P_i в T_j, которые касаются разреза
- Проверяем, имеет ли пересечение [a, b] с левой частью T_j длину хотя бы $|P_i|$.
- Если да, то делаем рекурсивный вызов процедуры обхода с тем же i, индексом левой части T_j , интервалом пересечения и смещением.
- Обрезаем прогрессию этих вхождений, оставляя лишь находящиеся целиком внутри [a, b].
- Записываем эту усеченную прогрессию в результат локального поиска.
- Проверяем, имеет ли пересечение [a, b] с правой частью T_j длину хотя бы $|P_i|$.
- Если да, то делаем рекурсивный вызов процедуры обхода с тем же i, индексом правой части T_j , интервалом пересечения и смещением.

T	|P|	Начальная	Упрощенная	Ускорение
AACR	1/100	752 мс	475 мс	1.5
AACR	1/200	338 мс	267 мс	1.2
AACR	1/500	109 мс	110 мс	0.99
AACR	1/1000	64 мс	48 мс	1.3
AAEZ	1/100	957 мс	768 мс	1.2
AAEZ	1/200	490 мс	461 мс	1.06
AAEZ	1/500	228 мс	179 мс	1.27
AAEZ	1/1000	127 мс	101 мс	1.25
AAES	1/100	3223 мс	2749 мс	1.17
AAES	1/200	1583 мс	1508 мс	1.04
AAES	1/500	791 мс	804 мс	0.98
AAES	1/1000	523 мс	553 мс	0.94
AAZV	1/100	12390 мс	10687 мс	1.151
AAZV	1/200	5860 мс	5167 мс	1.13
AAZV	1/500	2514 мс	2202 мс	1.14
AAZV	1/1000	1666 мс	1334 мс	1.24

Рис. 9: Сравнение времени выполнение начальной и упрощенной версии локального поиска (ускорение в среднем 15 процентов)

Данное упрощение ползволяет выиграть в скорости выполнения локального поиска в среднем 15 процентов и уменьшить объем требуемой памяти для локального поиска до O(1).

4.2.2 Рекурсия

Поскольку локальный поиск является рекурсивным, необходимо ответить на вопрос о глубине рекурсии. Ответ в описании алгоритма: глубина рекурсии пропорциональна глубине поддерева, в котором происходит поиск. Из этого следует, что рекурсию можно использовать только тогда, когда прямолинейная программа сбалансирована и ее высота O(log(n)), где n – количество элементов в ней. Вопрос о построении сбалансированных прямолинейных программ лежит вне приводимого исследования, ознакомиться подробнее с ним можно в работах [8] и [13]. Весь процесс анализа алгорит-

ма и его улучшения в исследовании рассматривался только на сбалансированных прямолинейных программах, но с авторской точки зрения, это не имеет особого значения, за исключением лишь вопроса о глубине рекурсии локального поиска.

В случае несбалансированности прямолинейных программ и, следовательно, большей глубины рекурсии, стэк вызовов больше и в худшем случае может достигать количества узлов в поддереве (крайний случай, когда дерево вырождается в список). Один из способов избавления от рекурсии состоит в конструировании стэка вызовов вручную. Изменение локального поиска исходя из предложенного способа:

1. Будем хранить порядок обхода ПП в стэке. Начальным элементов в нем будет кортэж <T_j, интервал пересечения, смещение>.
2. Пока стэк не пуст:
3. Смотрим на кортэж с вершины стэка.
4. Если его еще не посещали, кладем в стэк кортэж <индекс левой части T_j , интервал пересечения и смещение> и возвращаемся к 2.
5. Иначе, проверяем, имеет ли пересечение [a, b] с левой частью T_j длину хотя бы $|P_i|$.
6. Обрезаем прогрессию этих вхождений, оставляя лишь находящиеся целиком внутри [a, b].
7. Записываем эту усеченную прогрессию в результат локального поиска.
8. Убираем кортэж с вершины стэка.
9. Проверяем, имеет ли пересечение [a, b] с правой частью T_j длину хотя бы $|P_i|$.
10. Если да, то кладем в стэк кортэж <индекс правой части T_j , интервал пересечения и смещение> и возвращаемся к 2.

Однако является неясной скорость работы данной модификации локального поиска. Проводится сравнительное тестирование итеративной версии локального поиска с рекурсивной.

T	|P|	Без рекурсии	С рекурсией
AACR	1/100	773 мс	475 мс
AACR	1/200	330 мс	267 мс
AACR	1/500	105 мс	110 мс
AACR	1/1000	61 мс	48 мс
AAEZ	1/100	1005 мс	768 мс
AAEZ	1/200	530 мс	461 мс
AAEZ	1/500	257 мс	179 мс
AAEZ	1/1000	110 мс	101 мс
AAES	1/100	3384 мс	2749 мс
AAES	1/200	1630 мс	1508 мс
AAES	1/500	1195 мс	804 мс
AAES	1/1000	620 мс	553 мс
AAZV	1/100	12380 мс	10687 мс
AAZV	1/200	5693 мс	5167 мс
AAZV	1/500	2441 мс	2202 мс
AAZV	1/1000	1423 мс	1334 мс

Рис. 10: Сравнение времени выполнение упрощенной версии локального поиска с рекурсией и без

Анализ полученных результатов показывает, что на сбалансированных прямолинейных программах итеративный вариант сопоставим с рекурсивным по скорости, но все-таки немного уступает ему. Однако, несмотря на небольшое снижение скорости, данная модификация позволяет использовать алгоритм на несбалансированных ПП.

4.3 Параллелизм

Опираясь на улучшения из предыдущей главы удалось реализовать алгоритм, использующий меньший объем памяти и работающий быстрее первоначальной версии. Тем не менее, фактическая скорость работы алгоритма до сих пор оставляет желать лучшего. Текущая реализация алгоритма является последовательной, устранены все известные узкие места. Однако, последовательная реализация не позволяет использовать имеющиеся компьютерные ресурсы (в основном, многопроцессорность) для ускорения алгоритма. Решением является параллельная версия алгоритма. При разработке и реализации алгоритма необходимо ответить на следующие вопросы:

- Какова стратегия распараллеливания?
- Как обеспечивать равномерную загрузку потоков?
- Как добиться эффективной координации между потоками?

4.3.1 Стратегия распараллеливания

Одна из самых распространенных практик распараллеливания алгоритмов, вычисляющих ячейки таблицы последовательно заключается в том, чтобы считать строку или столбец используя заполненные до этого строки и столбцы, а после этого – вычисление строки и стобца разбить на блоки и вычислять их параллельно. В случае рассматриваемого алгоритма для вычисления i-ой строки используются ячейки только из предыдущих строк. Данный факт позволяет сконструировать скелет параллельного алгоритма:

1. Препроцессинг и вычисления однобуквенных текстов не изменяются
2. Для каждого P_i в порядке возрастания i

(a) Разбить T на блоки

(b) Блоки вычислить параллельно используя N потоков

Интерес представляют два параметра: количество потоков и способ разбиения T на блоки.

Ответ на вопрос о количестве потоков достаточно прост: для вычисления CPU-intensive задач количество потоков, как правило, берут равным количеству процессоров (в случае с hyper-threading количество необходимо увеличить на два): при уменьшении количества потоков нельзя загрузить доступные процессоры, а при увеличении возникают накладные расходы на переключение между потоками, что замедляет алгоритм в целом.

В качестве стратегий разбиения T рассматриваются:

- Разделение T на блоки размером равным количеству потоков. Назовем данную стратегию "интервальная".
- Разделение T на блоки по следующему правилу: каждому потоку сопоставим индекс k, число потоков обозначим за K. Блок для потока k определим как последовательность $T_k, T_{k+K}, T_{k+2K}, \ldots$. Назовем данные стратегию "порядковая".

С теоретической точки зрения, второй способ должен оптимальнее распределять нагрузку между потоками. Причина в том, что чем дальше правила расположены в T, тем дольше выполняется процедура локального поиска. Это связано с его работой за O(j) шагов, и чем больше j, тем большее количество операций необходимо выполнить. Тем не менее, было решено проверить обе стратегии на практике. Для этого было замерено время выполнения каждого из потоков для соответствующей стратегии.

Предположение о неоптимальности первой стратегии было подтверждено на практике: каждый следующий поток работает дольше предыдущего, что является следствием неравномерной загрузки потоков. Использование первой стратегии привело к ситуации, когда процент загрузки процессоров был не больше 75 процентов, что указывало на проблемы в реализация. Попытка "дозагрузить"процессоры с помощью добавления потоков ситуацию не улучшила: при добавлении очередного потока вместе с увеличением

T	|P|	1 поток	2 поток	3 поток	4 поток
AACR	1/100	1	1.34	1.35	1.32
AACR	1/200	1	1.30	1.6	2.51
AACR	1/500	1	1.35	1.92	3
AACR	1/1000	1	1.4	2.6	3
AAEZ	1/100	1	2.25	4	6
AAEZ	1/200	1	1.5	2.4	3.77
AAEZ	1/500	1	1.34	1.76	2.76
AAEZ	1/1000	1	1.6	1.9	3.3
AAES	1/100	1	1.89	3.02	5.92
AAES	1/200	1	1.80	2.8	5.72
AAES	1/500	1	1.6	2.57	4.8
AAES	1/1000	1	1.51	2.08	3.64
AAZV	1/100	1	1.92	3.75	8
AAZV	1/200	1	1.95	3.78	8.5
AAZV	1/500	1	1.73	2.96	6.29
AAZV	1/1000	1	1.60	2.6	5.2

Рис. 11: Относительное время работы потоков при первой стратегии(в разах)

T	|P|	1 поток	2 поток	3 поток	4 поток
AACR	1/100	1	1	1.1	1.05
AACR	1/200	1	1.1	1.08	1.08
AACR	1/500	1	1.03	1.14	1.04
AACR	1/1000	1	1.06	1	1.06
AAEZ	1/100	1.01	1.02	1	1.02
AAEZ	1/200	1.03	1.07	1.05	1
AAEZ	1/500	1.27	1.9	1.59	1
AAEZ	1/1000	1	1.03	1	1
AAES	1/100	1	1	1	1
AAES	1/200	1	1	1.01	1
AAES	1/500	1.02	1.03	1.07	1
AAES	1/1000	1	1	1	1.01
AAZV	1/100	1	1	1.01	1
AAZV	1/200	1.01	1	1.01	1
AAZV	1/500	1.03	1.03	1.03	1
AAZV	1/1000	1	1	1.01	1

Рис. 12: Относительное время работы потоков при второй стратегии(в разах)

T	|P|	1 стратегия	2 стратегия	Ускорение
AACR	1/100	216 мс	166 мс	1.3
AACR	1/200	110 мс	88 мс	1.24
AACR	1/500	45 мс	39 мс	1.15
AACR	1/1000	28 мс	28 мс	1
AAEZ	1/100	301 мс	233 мс	1.29
AAEZ	1/200	179 мс	115 мс	1.55
AAEZ	1/500	75 мс	49 мс	1.5
AAEZ	1/1000	62 мс	33 мс	1.87
AAES	1/100	1876 мс	1130 мс	1.66
AAES	1/200	895 мс	541 мс	1.65
AAES	1/500	387 мс	277 мс	1.39
AAES	1/1000	166 мс	122 мс	1.36
AAZV	1/100	4649 мс	3461 мс	1.34
AAZV	1/200	2769 мс	1991 мс	1.39
AAZV	1/500	1382 мс	810 мс	1.7
AAZV	1/1000	716 мс	515 мс	1.39

Рис. 13: Разница во времени работы при различных стратегиях

загруженности процесоров на несколько процентов увеличивалось и время работы.

Предположение о равномерной загрузке потоков при использовании второй стратегии было подтверждено. Вследствие лучшей загрузки потоков время работы алгоритма уменьшилось. Однако при использовании второй стратегии не удалось загрузить процессоры на 100 процентов, хотя в данном случае этот показатель выше – около 90 процентов. Улучшение процента загрузки процессоров остается вопросом для исследования.

4.3.2 Координация потоков

Решение проблемы эффективной координации потоков применительно к рассматриваемому алгоритму состоит в ответе на следующие два вопроса:

- Как узнать, что все потоки уже вычислили свои блоки?
- Как обеспечить потокобезопасное сохранение результатов параллельных вычислений блоков?

Ответом на первый вопрос является использование барьеров. Данная абстракция позволяет останавливать выполнение потока, пока не выполнено определенное условие. В рассматриваемом случае барьер используется в главном потоке, и условием является окончание расчета блоков соответствующими потоками.

Ответить на второй вопрос можно, используя одну из следующих стратегий:

- Использование потокобезопасной реализацию таблицы прогрессий и сохранение результатов в таблицу прогрессий по мере их вычисления потоками. Достоинство данного способа в отсутствии необходимости в дополнительной памяти, недостатком являются издержки на синхронизацию таблицы прогрессий. Назовем данную стратегию "с общей памятью".
- Позволить потокам иметь собственную память, в которой они будут хранить результаты вычислений соответствующих блоков, а сохранять

T	|P|	1 стратегия	2 стратегия	Ускорение
AACR	1/100	340 мс	228 мс	1.49
AACR	1/200	208 мс	118 мс	1.76
AACR	1/500	90 мс	41 мс	2.19
AACR	1/1000	39 мс	22 мс	1.77
AAEZ	1/100	591 мс	383 мс	1.54
AAEZ	1/200	295 мс	154 мс	1.91
AAEZ	1/500	112 мс	84 мс	1.33
AAEZ	1/1000	61 мс	38 мс	1.60
AAES	1/100	2547 мс	1563 мс	1.62
AAES	1/200	1465 мс	745 мс	1.96
AAES	1/500	506 мс	349 мс	1.44
AAES	1/1000	246 мс	163 мс	1.50
AAZV	1/100	5574 мс	5262 мс	1.05
AAZV	1/200	3413 мс	2512 мс	1.35
AAZV	1/500	1561 мс	965 мс	1.61
AAZV	1/1000	886 мс	618 мс	1.43

Рис. 14: Сравнение различных стратегий сохранения результатов

результаты в потоконебезопасную таблицу прогрессий будет главный поток. Данный способ, напротив, не требует синхронизации таблицы прогрессий, но требует дополнительной памяти пропорционально длине ПП текста. Назовем данную стратегию "агрегированной".

Каждая из стратегий имеет свои недостатки и достоинства, но в общем случае нельзя сказать, какая из них является эффективнее, поэтому были измерены скорости выполнения обеих стратегий.

График демонстрирует, что вторая стратегия оказалась выигрышнее. Несмотря на то, что потокобезопасная таблица прогрессий основана на эффективной реализации, которая присутствует в стандартной библиотеки Java 7 – ConcurrentHashMap, издержки на сихронизацию таблицы прогрессий оказывают существенное влияние на скорость выполнения алгоритма.

T	|P|	Последовательно	Параллельно	Ускорение
AACR	1/100	875 мс	153 мс	5.79
AACR	1/200	459 мс	101 мс	4.54
AACR	1/500	132 мс	39 мс	3.38
AACR	1/1000	74 мс	19 мс	3.89
AAEZ	1/100	1423 мс	223 мс	6.43
AAEZ	1/200	797 мс	125 мс	6.36
AAEZ	1/500	303 мс	60 мс	5.05
AAEZ	1/1000	110 мс	36 мс	3.05
AAES	1/100	4393 мс	1260 мс	3.46
AAES	1/200	2704 мс	565 мс	4.7
AAES	1/500	1439 мс	244 мс	5.89
AAES	1/1000	699 мс	121 мс	5.77
AAZV	1/100	16506 мс	4187 мс	3.94
AAZV	1/200	7374 мс	2050 мс	3.59
AAZV	1/500	3115 мс	844 мс	3.75
AAZV	1/1000	2132 мс	475 мс	4.48

Рис. 15: Сравнение последовательной и параллельной версии

4.4 Результаты

Теоретическая оценка требуемой памяти, приведенная автором алгоритма, не изменилась – $O(nm)$. Обнаружение факта разреженности таблицы и выбор оптимальной структуры данных позволили уменьшить объем требуемой памяти на практике, но пока не позволяют говорить о лучшей асимптотической оценке. Теоретическая оценка времени работы алгоритма также не изменилась – $O(n^2 \cdot m)$. Однако оптимизация локального поиска и параллелизация алгоритма позволили увеличить скорость работы на практике. В работе было проведено сравнение базовой последовательной (неоптимизированная таблица на ассоциативном массиве, неоптимизированная версия локального поиска) и оптимизированной параллельной версии(с оптимизированной под ключ таблицей на ассоциативном массиве, оптимизированная версия локального поиска, "порядковая"и "агреггированная"стратегия):

Ускорение более чем в 6 раз связано не только с параллелизацией, но

и с выбором подходящей структуры для таблицы прогрессий и улучшением локального поиска. Однако, в некоторых замерах видно, что ускорение оказывается менее чем в 4 раза. Данный факт можно объясним с помощью закона Амдала: в алгоритме до сих пор остались последовательные части (например, агрегация результатов, полученных из нескольких потоков).

Остается открытым вопрос о возможности дальнейшей оптимизации алгоритма, которая существенно бы уменьшила скорость его работы. Стоит рассмотреть следующие возможные точки улучшения:

- Исследование более эффективных стратегий распараллеливания.
- Использование распределенных вычислений. В рамках проведенной работы удалось распараллелить алгоритм и упереться в производительность одного компьютера. Одним из вариантов может быть использование OpenMPI(http://en.wikipedia.org/wiki/Open_MPI).
- Исследовать разреженность таблицы с теоретической стороны, что возможно позволило бы уменьшить теоретическую оценку алгоритма по требуемой памяти.
- Исследовать границу, когда таблица прогрессий перестанет умещаться в оперативной памяти. В данном случае понадобится другая техника работы с таблицей прогрессий, с хранением ее на жестком диске и эффективным доступом к ней.
- Возможность более плотной упаковки арифметических прогрессий, например, при использовании кодирование с переменной длиной (http://en.wikip
length_quantity).

4.4.1 Сравнение с алгоритмами на строках

Одним из значимых направлений исследования является сравнение алгоритма Лифшица с классическими алгоритма поиска. Кроме того, представляет интерес проверка существования границы, когда алгоритм Лифшица может оказаться эффективнее. Для сравнения используется однопоточная реализация алгоритма Кнута-Морриса-Пратта (время работы $O(m + n)$).

T	|P|	Алгоритм Кнута– Морриса – Пратта	Алгоритм Лифшица
AACR	1/100	1 мс <	153 мс
AACR	1/200	1 мс <	101 мс
AACR	1/500	1 мс <	39 мс
AACR	1/1000	1 мс <	19 мс
AAEZ	1/100	1 мс <	223 мс
AAEZ	1/200	1 мс <	125 мс
AAEZ	1/500	1 мс <	60 мс
AAEZ	1/1000	1 мс <	36 мс
AAES	1/100	1 мс <	1260 мс
AAES	1/200	1 мс <	565 мс
AAES	1/500	1 мс <	244 мс
AAES	1/1000	1 мс <	121 мс
AAZV	1/100	1 мс <	4187 мс
AAZV	1/200	1 мс <	2050 мс
AAZV	1/500	1 мс <	844 мс
AAZV	1/1000	1 мс <	475 мс

Рис. 16: Сравнение алгоритма Кнута-Морриса-Пратта и алгоритма Лифшица

Хотя сравнивать однопоточную реалиацию алгоритма с многопоточной не совсем правильно методически, данное упрощение позволит построить примерное сравнение времени их работы.

Анализ полученных результатов показывает, что алгоритм Кнута-Морриса-Пратта на тестовых выборках работает практически мгновенно (менее 1 мс), а время работы предложенной реализации алгоритма Лифшица больше на два-три порядка. Поскольку алгоритм Лифшица работает над ПП текста и шаблона, время его работы напрямую зависит от размера ПП и сжимаемости исходного текста и шаблона. Однако вопрос сжимаемости текстов на практике остается вне текущего исследования. На данный момент обозначить границы, когда алгоритм Лифшица будет работать быстрее, не представляется возможным.

5 Заключение

В работе была представлена реализация алгоритма Лифшица и его улучшения. Таблица прогрессий оказалась разреженной и для нее была выбрана оптимальная структура данных, которая уменьшает потребляемую память и позволяет использовать алгоритм на бóльших объемах данных. Была упрощена процедура локального поиска и исследован вопрос ее применимости при различных ПП. Кроме того, была представлена параллельная реализация алгоритма Лифшица, которая основывается на тех же идеях, что и первоначальный алгоритм. Данные улучшения позволили ускорить алгоритм на приведенных тестах с трех до шести раз.

Было произведено сравнение классических алгоритмом поиска шаблона в тексте с алгоритмом Лифшица для нахождения границы применимости последнего. На данный момент существование такой границы остается под вопросом, так как на текстах размером в сотни килобайт время работы алгоритма Лифшица на порядки больше времени работы классических алгоритмов. Исходя из того, что для исследования был выбран алгоритм на ПП с самой лучшей асимптотикой, ситуация с остальными алгоритмами должна быть похожей. Это позволяет сделать вывод о том, что на данный момент эффективное использование на практике ПП и алгоритмов на них не представляется возможным.

Список литературы

[1] *M. Karpinski, W. Rytter, and A. Shinohara*, Pattern-matching for strings with short descriptions. In Proc. Combinatorial Pattern Matching, volume 637 of Lecture Notes in Computer Science, pages 205-214. Springer-Verlag, 199

[2] *M. Karpinski, W. Rytter, and A. Shinohara*, An efficient pattern-matching algorithm for strings with short descriptions. Nordic Journal of Computing, 1997.

[3] *Masamichi Miyazaki, Ayumi Shinohara, Masayuki Takeda*, An improved pattern matching algorithm for strings in terms of straight-line programs, Combinatorial Pattern Matching Lecture Notes in Computer Science Volume 1264, 1997, pp 1-11

[4] *Yu. Lifshits*, Processing Compressed Texts: A Tractability Border, Combinatorial Pattern Matching Lecture Notes in Computer Science Volume 4580, 2007, pp 228-240

[5] *W. Plandowski*, Testing equivalence of morphisms on context-free languages. In ESA'94, LNCS 855, pages 460470. Springer-Verlag, 1994.

[6] *M. Miyazaki, A. Shinohara, and M. Takeda*, An improved pattern matching algorithm for strings in terms of straight line programs. In CPM'97, LNCS 1264, pages 111. Springer-Verlag, 1997.

[7] *Бурмистров И. С., Козлова А. В., Курпилянский Е. Б., Хворост А. А.* Эффективное сжатие данных с помощью прямолинейных программ, Записки научных семинаров ПОМИ, RuFiDiM (2012), 45-68.

[8] *W. Rytter*, Application of Lempel-Ziv factorization to the approximation of grammar-based compression, Theor. Comput. Sci., 302 (2003), 211–222.

[9] *D. Huffman*, A Method for the Construction of Minimum-Redundancy Codes Proceedings of the IRE, Vol. 40, No. 9. (1952), pp. 1098-1101

[10] *Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa*, Pattern matching in text compressed by using antidictionaries, Lect. Notes Comput. Sci., 1645 (1999), 37–49.

[11] *T. Welch*, A technique for high-performance data compression, IEEE Computer, 17 (1984), 8–19.

[12] *J. Ziv, A. Lempel*, A universal algorithm for sequential data compression, IEEE Trans. Information Theory, 23 (1977), 337–343.

[13] *I. Burmistrov, L. Khvorost*, Straight-line programs: a practical test, Proc. Int. Conf. Data Compression, Commun., Process., CCP (2011), 76–81.

[14] *I. Burmistrov, A. Kozlova, E. Kurpilyansky, A. Khvorost*, Straight-line programs: a practical test, Journal of Mathematical Sciences 192.3 (2013): 282-294.

[15] *W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto.* Computing longest common substring and all palindromes from compressed strings. In SOFSEM, volume 4910 of Lecture Notes in Computer Science, pages 364-375. Springer, 2008.