Содержание

1 Введение

Во многих современных задачах возникает потребность в обработке больших объемов данных. Это связано с быстро растущим объемом доступной информации (например в сети Интернет). Среди существующего многообразия алгоритмов поиска и обработки данных часто оказывается сложно найти метод, подходящий для практического применения. Это связано с тем, что каждый из существующих алгоритмов накладывает некие ограничения на данные, с которыми он может работать. Поэтому не всегда возможно найти готовый алгоритм, который будет подходить для решения новой задачи.

Из-за потребности в решении задач со специфическими ограничениями, стали появляться классы алгоритмов с характерными особенностями в работе. Например, существует класс алгоритмов, которые эффективно работают с кэшем процессора и таким образом дают хорошую скорость работы алгоритма (класс cache-oblivious). Также можно вспомнить о классе IO-efficient алгоритмов — они призваны обрабатывать объемы данных, которые невозможно сохранить в оперативной памяти, и поэтому сильно зависят от способа работы с жестким диском.

В конце XX века стала развиваться менее очевидная идея обработки больших объемов данных, которая заключается в том, что данные можно хранить в сжатом виде. При этом, если способ сжатия имеет хорошую структуру, это позволит обрабатывать текст без предварительной распаковки. Этот подход позволяет сэкономить место для хранения данных и уменьшить объемы входных и выходных данных для решаемой задачи. Такие алгоритмы называются алгоритмами над сжатыми представлениями.

Существуют разные методы сжатия данных: контекстное сжатие [9] [8], статистическое сжатие [6], сжатие с помощью антисловарей [7] и т. д. Понятно, что не каждый способ сжатия данных порождает структуру, поддерживающую возможность выполнять поисковые запросы. В работе рассматривается один из самых популярных сегодня методов — метод сжатия с помощью прямолинейных программ (straight line programs, [5]). Прямолинейные программы являются удобной абстракцией, позволяющей описывать различные способы сжатия данных. В частности, известно, что сжатие с помощью алгоритмов типа Лемпеля–Зива почти эквивалентно прямолинейной программе. Более точно — существует эффективный алгоритм, позволяющий по данному кодированию типа Лемпеля–Зива построить прямолинейную программу кодирующую тот же текст, причем длина кода увеличится не более чем в $O(log|T|)$ раз, где $|T|$ — длина закодированной строки [5]. Это позволяет рассматривать только алгоритмы над прямолинейными программами, что гораздо удобнее из-за простой структуры ПП.

Прямолинейная программа (ПП) — это контекстно-свободная грамматика, порождающая ровно одну строку. Правила этой грамматики могут иметь один из двух видов $X_i \to a$ (терминальные символы) и $X_i \to X_j X_k$, где $i > j, k$.

Мы подробнее рассмотрим понятие прямолинейной программы в одной из следующих глав.

Одной из самых распространенных задач, связанных с получением информации о тексте — поиск образца в тексте. Эта задача — одна из простейших, а потому встречается

на практике крайне часто (например, в работе поисковых машин, сборе статистики и т. д.).

Задача проверки равенства двух сжатых строк была решена в 1994 году в работе [2] за время $O(n^4)$, в 1997 году появился алгоритм поиска образца в строке, имеющий сложность $O(n^2m^2)$, где n и m — размеры прямолинейных программ, порождающих образец и текст соответственно [3]. В нашей работе рассматривается алгоритм, работающий за время $O(n^2m)$, предложенный Ю. Лифшицем в работе [1].

В статье [1] доказывается теоретическая асимптотическая оценка времени работы алгоритма. Однако, на практике скорость работы алгоритма может зависеть от деталей реализации. Также, при практическом применении важна не только асимптотика, но и фактическая скорость работы алгоритма на реальных данных. Кроме того, различные алгоритмы могут показывать различный результат в зависимости от особенностей входных данных (например, для текстов с маленьким алфавитом или текстов имеющих специальную структуру).

В этой работе мы рассматриваем следующие вопросы:

- Насколько эффективен алгоритм Лифшица на практике? Удается ли достичь теоретически предсказываемой эффективности?
- Как можно увеличить производительность алгоритма на практике?
- Как меняется эффективность алгоритма в зависимости от исходного текста, на котором была построена ПП?
- В какой степени алгоритм Лифшица допускает распараллеливание?

Структура работы такова: во второй главе вводятся основные определения, касающиеся прямолинейных программ и алгоритма Лифшица; третья глава посвящена краткому изложению алгоритма; в четвертой главе приводятся результаты практических тестов алгоритма и описывается ход исследования; в пятой главе описаны основные выводы.

2 Основные определения

Будем рассматривать тексты над конечным алфавитом Σ.

Прямолинейная программа (ПП) — это контекстно-свободная грамматика, порождающая ровно одну строку. Правила этой грамматики могут иметь один из двух видов $X_i \to a$ (терминальные символы) и $X_i \to X_jX_k$, где $i > j, k$.

Строка, которую выводит прямолинейная программа, относится к последнему нетерминалу X_n.

Пример: Рассмотрим ПП, которая порождает текст «*abaababaabaab*»:

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

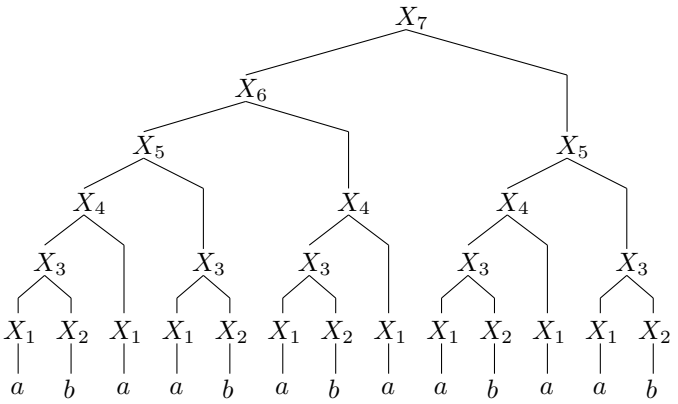

Рис. 1: Дерево вывода слова «abaababaabaab»

Дерево вывода этой грамматики изображено на рис. 1

Каждый нетерминал грамматики порождает некоторую подстроку исходной строки. Например, из нетерминала X_5, в примере приведенном выше, выводится подстрока *abaab*.

Для удобства изложения мы будем обозначать символом X как прямолинейную программу, так и порождаемую ею строку. Символы X_i обозначают отдельные нетерминалы контекстно-свободной грамматики X (или строки, соответствующие этим нетерминалам).

Символ $|X|$ означает количество символов в строке, порождаемой прямолинейной программой X.

Для изложения алгоритма Лифшица нам потребуется ввести некоторые термины.

- *Позицией* в строке называется место между любыми двумя соседними символами, а так же 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* будет использоваться для обозначения начальной позиции подстроки в тексте.
- Пусть строка порождается грамматикой с нетерминалами $X_1, \ldots, X_n$. Будем называть *позицией разреза* (или просто *разрезом*) начальные позиции для терминальных строк и позиции на стыке двух правил.

В примере из рис. 1 разрез строки «*abaab*», выводящейся 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*» так же входит в «*ababa*» дважды, причем оба вхождения касаются разреза.

3 Алгоритм Лифшица

3.1 Общая схема

Мы приведем краткое описание алгоритма Лифшица, останавливаясь подробно лишь на деталях, важных для нашего анализа алгоритма. Более полное описание алгоритма можно найти в работе [1].

Алгоритм получает на вход две прямолинейные программы T и P, выводящие текст и образец, который нужно найти в этом тексте. Результатом работы является список всех вхождений образца в текст.

Заметим, что хранить позиции вхождений P в T в явном виде не представляется возможным, поскольку при достаточно большом тексте их может оказаться слишком много. Например, если искать строку вида a^k в строке вида a^p, число вхождений будет экспоненциально относительно размеров сжатого текста. Поэтому, хранить эту информацию следует в сжатом виде. Авторы работ [1] и [3] используют для хранения информации о позициях вхождений арифметические прогрессии, основываясь при этом на следующем теоретическом факте:

Лемма 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 элеменов в прогрессии.

В работе [1] 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 $O(n^2m)$ с помощью метода динамического программирования.*

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

Построение таблицы арифметических прогрессий состоит из трех этапов.

1. Сбор вспомогательной информации: длин и разрезов строк, выводимых из всех правил грамматики.

2. Вычисление значений ячеек таблицы арифметических прогрессий, которые соответствуют однобуквенным текстам.

3. Вычисление оставшихся ячеек таблицы арифметических прогрессий в порядке от меньшего(по длине) образца к большему и от меньшего текста к большему.

Шаги 1 и 2 достаточно просты. Остановимся подробнее на описании третьего шага.

Вычисление значения в ячейке $A[i, j]$ для нетерминальных строк происходит следующим образом. Пусть ПП P содержит правило $P_i \to P_f P_s$. Будем для удобства считать, что длина строки P_f не меньше, чем длина строки P_s.

(a) Найдем все вхождения P_f в T_j, которые могут быть началом интересующего нас вхождения.

(b) Найдем вхождения P_s в T_j, которые начинаются в позициях концов вхождений P_f.

(c) Пересечем полученные множества вхождений и запишем их в виде одной арифметической прогрессии вхождений P_i в T_j.

Для поиска P_f в T_j будем использовать вспомогательную процедуру `LocalSearch(f, j,` α`,` β`)`, работа которой будет описана ниже. Эта процедура находит все вхождения P_f в T_j на отрезке $[\alpha, \beta]$. Можно показать, что эти вхождения записываются в виде не более чем двух арифметических прогрессий. Для того, чтобы процедура работала корректно требуется чтобы выполнялось неравенство

$$\beta - \alpha < 3|P_f|. \tag{1}$$

Чтобы найти все вхождения P_f которые потенциально могут быть началом вхождения P_i, касающегося разреза, запустим процедуру поиска `LocalSearch(`f`, `j`, `r` - |`P_i`|, `r` + |`P_f`|)`, где r — позиция разреза в T_j. Заметим, что неравенство (1) выполняется, поскольку мы рассматриваем случай $|P_f| \geqslant |P_s|$. Обратный случай обрабатывается аналогичным образом.

После того как интересующие нас вхождения P_f в T_j найдены, нужно для каждого из них проверить, верно ли что существует вхождение P_s, начинающееся в позиции конца данного вхождения. Эта проверка также осуществляется с помощью вспомогательной процедуры `LocalSearch`.

Отметим, что алгоритм Лифшица позволяет проверить все вхождения P_f, образующие арифметическую прогрессию, не рассматривая каждое вхождение отдельно. Это позволяет существенно уменьшить количество обращений к функции `LocalSearch` на этапе (b).

Позиции вхождений P_f, для которых проверка прошла успешно, образуют интересующее нас множество вхождений P_i в T_j, касающихся разреза. По лемме 1 эти позиции будут образовывать одну арифметическую прогрессию.

3.2 Процедура Localsearch

`LocalSearch(`f`, `j`, `α`, `β`)` работает следующим образом Вхождения P_f в T_j, которые касаются разреза, уже найдены и записаны в ячейке $A[f, j]$ таблицы арифметических прогрессий, нужно лишь выбрать те из них, что находятся внутри отрезка $[\alpha, \beta]$.

После того как такие вхождения найдены остается найти вхождения P_f в T_j, которые не касаются разреза. Таким вхождениям соответствуют вхождения P_f в строки T_{j_1} и T_{j_2} (мы считаем, что правило для T_j имеет вид $T_j \to T_{j_1} \cdot T_{j_2}$). Для того чтобы найти их запустим процедуру поиска рекурсивно для каждой из строк T_{j_1} и T_{j_2} на нужных подотрезках.

В результате получим множество позиций, которые образуют не более чем две арифметические прогрессии [1].

Процедура `LocalSearch` работает за время $O(j)$.

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 будет выглядеть следующим образом:

	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. 2: Вычислены значения ячеек, соответствующих однобуквеннм текстам

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

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

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 этой 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 найти вхождения P_3 в T_5 на отрезке $[1, 5]$, не касающиеся разреза. Для этого нужно рекурсивно запустить процедуру `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 будут выглядеть следующим образом: `LocalSearch`$(3, 4, [1, 3])$ и `LocalSearch`$(3, 3, [1, 2])$.

Первый вызов найдет арифметическую прогрессию $(1, 1, 1)$ из ячейки $APTable[3, 4]$. В результате второго вызова новых вхождений не найдется. Дальнейших рекурсивных вызовов также не последует из-за того, что отрезки поиска станут 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 Практические исследования

4.1 Особенности реализации

Все практические исследования выполнены с помощью программ на языке Java, которые находятся в свободном доступе по ссылке `https://code.google.com/p/overclocking`.

Прямолинейные программы, порождающие текст и образец строились с помощью алгоритма, описанного в статье [4], который является модификацией алгоритма Риттера [5]. Этот алгоритм строит ПП, деревья выводов которых являются сбалансированными. За счет этого достигается лучшая степень сжатия.

Первая проблема, с которой мы столкнулись при попытке запуска алгоритма Лифшица на практике, заключается в том, что для его работы требуется заполнять таблицу арифметических прогрессий, размер которой $n \times m$, где n — размер прямолинейной программы P (то есть количество нетерминальных символов в этой параллельной программе), а m — размер параллельной программы T. В каждой ячейке таблицы хранится три целых числа, занимающие 12 байт без учета накладных расходов на создание объекта. Если T содержит 10^5 правил (этому соответствует ДНК размером около 1,5 мегабайт), поиск образца размером 10^4 требует около 12 гигабайт оперативной памяти только для хранения таблицы арифметических прогрессий. То есть даже при относительно небольших размерах сжатых текста и образца хранить таблицу в памяти становится проблематично.

Однако, хранить все ячейки таблицы в течении всего времени работы алгоритма необязательно. Чтобы получить информацию о вхождении образца в текст, достаточно последней строки таблицы. Остальные строки нужны лишь для промежуточных вычислений.

Так, для вычисления ячейки $APTable[i, j]$ нужно чтобы уже были вычислены ячейки $APTable[f, 1..j]$, $APTable[s, 1..j]$. Здесь мы считаем, что правило P_i имеет вид $P_i \to P_f P_s$. Это позволяет построить граф зависимостей строк таблицы и хранить очередную строку только до тех пор, пока она нужна для вычислений.

Также, была выдвинута гипотеза о том, что чем больше размер используемого алфавита, тем меньше соответствия будет между текстом и образцом и, соответственно, больше пустых ячеек в таблице арифметических прогрессий. Для подтверждения этой гипотезы была посчитана степень разреженности таблицы для текстов над алфавитом размера 4 и 26.

	Total	Empty	Percentage
$TSize = 14463$ $PSize = 14463$	209178369	208745588	99.79%
$TSize = 14463$ $PSize = 2283$	48928329	48644839	99.42%
$TSize = 14463$ $PSize = 1841$	26626383	26383393	99.09%
$TSize = 14463$ $PSize = 1019$	14737797	14536063	98.63%
$TSize = 14463$ $PSize = 465$	6725295	6572425	97.72%
$TSize = 14463$ $PSize = 266$	3847158	3721341	96.72%

Рис. 4: Разреженность таблицы для четырехбуквенного алфавита

Из приведенных данных видно, что гипотеза подтверждается на практике. Это позволяет вообще отказаться от хранения таблицы как таковой и хранить лишь ячейки, содержащие непустые арифметические прогрессии, например, с помощью хеш-таблиц или списков.

	Total	Empty	Percentage
$TSize = 35133$ $PSize = 35133$	1234327689	1234013855	99.97%
$TSize = 35253$ $PSize = 7694$	271236582	271040921	99.92%
$TSize = 35225$ $PSize = 4147$	146078075	145899926	99.88%
$TSize = 35243$ $PSize = 2307$	81305601	81145099	99.80%
$TSize = 35176$ $PSize = 1060$	37286560	37161795	99.67%
$TSize = 35178$ $PSize = 580$	20403240	20299706	99.49%

Рис. 5: Разреженность таблицы для 26-буквенного алфавита

4.2 Исследование эффективности алгоритма

Была выдвинута гипотеза о том, что на текстах над маленьким алфавитом алгоритм Лифшица может показать более эффективную работу, за счет того что структуры прямолинейных программ, выводящих текст и образец могут оказаться более похожими, чем для больших алфавитов. Поэтому для тестирования алгоритма был выбран класс текстов ДНК. Помимо того, что такие тексты отличаются большим объемом и строятся над алфавитом всего из четырех букв, эффективные методы работы с ними могут оказаться полезны в различных работах из области биоинформатики. Используемые нами ДНК были взяты из открытого банка Японии (`http://www.ddbj.nig.ac.jp/`).

В результате была собрана статистика по следующим текстам

- Случайные тексты над 26-буквенным алфавитом
- Случайные тексты над 4-буквенным алфавитом
- Последовательности ДНК

Ниже приведены графики зависимости времени работы алгоритма от длины образца для трех упомянутых типов текстов. Длина текста внутри одного графика фиксирована.

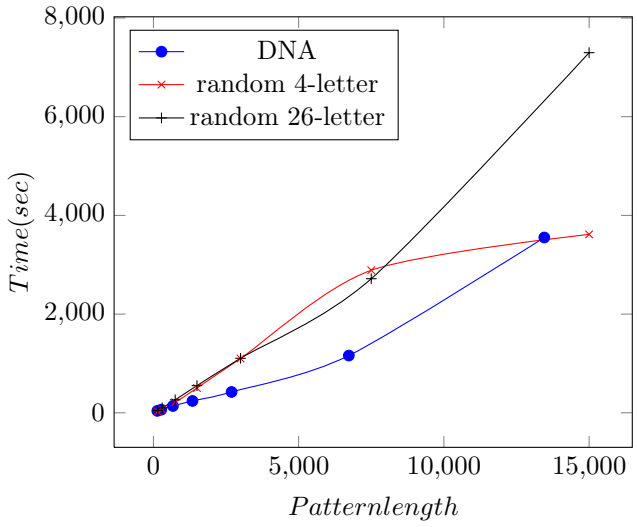

Рис. 6: Длина текста около 1500000 символов

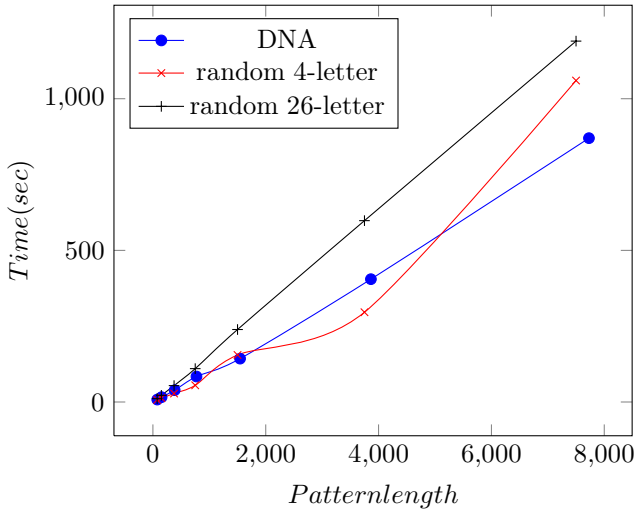

Рис. 7: Длина текста около 750000 символов

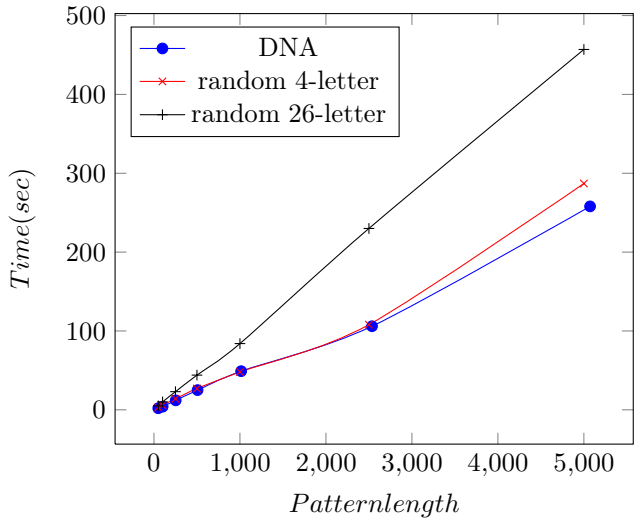

Рис. 8: Длина текста около 500000 символов

Из графиков видно, что алгоритм действительно показывает большую эффективность для текстов над алфавитом из четырех букв. На текстах 26-буквенного алфавита алгоритм работает почти в два раза медленнее, чем на строках ДНК такой же длины. При этом существенной разницы между временем работы на случайных текстах над четырехбуквенным алфавитом и на строках ДНК как правило не наблюдается. Из этого можно сделать вывод о том, что на фрагментах ДНК алгоритм работает быстрее именно за счет небольшого размера алфавита, и специальная структура ДНК не имеет существенного влияния на скорость работы алгоритма.

На рис. 9 показана зависимость между реальным временем работы алгоритма в секундах и функцией n^2m из теоретической оценки времени работы алгоритма. Видно,

что зависимость близка к линейной, то есть сравниваемые величины пропорциональны. Это позволяет утверждать что теоретическая оценка времени работы алгоритма подтверждается на практике.

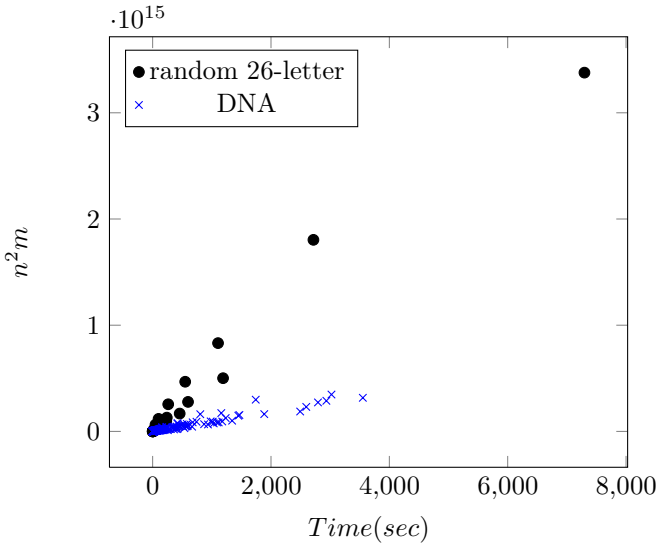

Рис. 9: Сравнение асимптотической оценки и реального времени работы алгоритма

Отметим, что если выбирать в качестве параметра не длину текста, а длину ПП, порождающей этот текст, исследуемый алгоритм показывает лучший результат для текстов над 26-буквенным алфавитом. Это связано с тем, что тексты над четырехбуквенным алфавитом имеют большую степень сжатия и, как следствие, большую плотность непустых ячеек в таблице арифметических прогрессий. Это приводит к тому, что глубина рекурсии в процедуре LocalSearch для текстов с четырех буквенным алфавитом больше и соответственно больше константа в асимптотике времени работы.

В процессе реализации было выделено несколько моментов, которые потенциально могут вызывать чрезмерные затраты по времени работы алгоритма.

Основной момент, дающий существенное замедление — рекурсивная процедура LocalSearch. Суммарное время выполнения вызовов этой процедуры составляет порядка 70% времени работы алгоритма, а общее количество вызовов достигает порядка 10^{15}. Исходя из того что рекурсивная реализация как правило уступает в эффективности нерекурсивному аналогу, можно предположить, что развертывание рекурсии окажется существенной оптимизацией.

Помимо этого, вспомним, что для вычисления значения в ячейке таблицы арифметических прогрессий $A[i, j]$ требуется чтобы были уже вычислены значения $A[f, 1] \dots A[f, j]$, $A[s, 1] \dots A[s, j]$, где $P_i \to P_f P_s$. То есть, для вычисления очередной ячейки достаточно знать значения в двух строках таблицы, соответствующих правилам, образующим P_i (а точнее префиксов этих строк).

Такая ситуация приводит к мысли о том, что некоторые ячейки таблицы арифметических прогрессий можно вычислять параллельно.

Рассмотрим граф зависимостей ячеек таблицы. Каждому значению $A[i][j]$ соответствует вершина графа. Направленное ребро из вершины $A[i][j]$ в вершину $A[k][l]$ означает, что для вычисления значения в вершине $A[i][j]$ используется значение в вершине $A[k][l]$. Структура графа определяется видом ПП P и T, поступающих на вход алгоритму. С точки зрения распараллеливания вычислений это означает, что значения в двух ячейках могут вычисляться параллельно тогда и только тогда, когда не существует направленного пути из одной вершины в другую.

Разумеется, эффективность распараллеливания существенно зависит от реализации, но анализ графа зависимостей позволяет дать разумную оценку эффективности. Мы будем оценивать две характеристики этого графа: максимальную длину пути и количество вершин, находящихся на расстоянии s от листов дерева для различных s. Первая характеристика соответствует распараллеливанию с "неограниченным" числом потоков: если считать, что мы можем одновременно вычислять все значения, которые доступны в данный момент именно длина максимального пути соответствует времени работы алгоритма. Вторая характеристика в разумном смысле показывает, сколько потоков могут эффективно работать одновременно: количество потоков не должно превосходить количества вершин на большей части уровней. Строго говоря, такая оценка не вполне точна: вершины, находящиеся на более высоком уровне не требуют для своей обработки готовности *всех* вершин нижнего уровня, но нам представляется, что подобная оценка будет достаточно разумной.

Были измерены указанные характеристики, ниже приведен результат для одного из тестов, в остальных принципиальных отличий не наблюдается.

Расстояние до листов	$MaxPathLength = 12$ $TextSlpSize = 14480$ $PatternSlpSize = 416$
1	59568
2	231616
3	1621312
4	1795024
5	1056748
6	579040
7	303996
8	188188
9	86856
10	57904
11	28952

Рис. 10: Количество вершин на фиксированном расстоянии от листов

Из результатов видно, что количество вершин на одном уровне значительно превышает количество потоков, которое разумно использовать при распараллеливании вычислений. Это означает, что без учета обмена данными между потоками, эффективность распараллеливания будет пропорциональна количеству потоков.

Также, можно использовать подход, в котором ребра, входящие в вершину, удаляются

после того, как значение в вершине было посчитано и стало доступно для использования. Вершина становится доступной для вычисления, если из нее не выходит ни одного ребра.

В качестве первого шага была реализована версия алгоритма с параллельными вычислениями, основанная на том, что ячейки таблицы, находящиеся в одной строке не зависят друг от друга, и, таким образом могут вычисляться параллельно (при условии что все предыдущие строки таблицы уже посчитаны).

Это простое распаралелливание дало на восьмиядерном процессоре ускорение в четыре раза. Ясно, что этот результат можно существенно улучшить за счет выбора более удачной стратегии распаралелливания.

5 Заключение

Не смотря на то, что теоретическая асимптотическая оценка времени работы алгоритма верна, реальное время работы оказывается довольно большим. Это связано с затратами на выполнение процедуры LocalSearch а так же на операции хеш-таблицы, в которой хранится таблица арифметических прогрессий. Возможно выбор других инструментов при реализации алгоритма позволит ускорить его работу.

В ходе реализации выяснилось, что алгоритм требует существенной оптимизации по памяти из-за огромных размеров таблицы арифметических прогрессий. Предложено решение этой проблемы, основанное на наблюдении о сильной разреженности таблицы. Практические тесты показывают высокую эффективность этого решения.

Также, было предложено несколько возможных оптимизаций алгоритма по времени. Самая существенная связана со вспомогательной рекурсивной процедурой LocalSearch, время исполнения которой занимает большую часть времени работы алгоритма. Было сделано предположение о том, что если прибегнуть к стандартной для таких случаев оптимизации — развертыванию рекурсии, это позволит значительно уменьшить время работы алгоритма.

Была продемонстрирована зависимость эффективности алгоритма от размера алфавита, с которым он работает.

Было выдвинуто и подтверждено предположение о том, что алгоритм Лифшица допускает распараллеливание, которое позволит значительно ускорить работу алгоритма.

Список литературы

[1] *Yu. Lifshits*, Processing Compressed Texts: A Tractability Border.

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

[3] *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.

[4] *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.

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

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

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

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

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