ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Федоркина Мария Олеговна

РЕАЛИЗАЦИЯ АЛГЕБРАИЧЕСКИХ АЛГОРИТМОВ ПРИБЛИЖЕННОГО ПОИСКА ПО ОБРАЗЦУ В СЖАТЫХ ДАННЫХ

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
канд.физ.-мат.наук
Д.А. Березун

Руководитель
д-р физ.-мат. наук
Б.А. Новиков

Консультант
д-р физ.-мат. наук, DPhil
А.В. Тискин

Санкт-Петербург 2021

Оглавление

Введение **4**

1. Обзор литературы **10**

1.1. Задача приближенного поиска по образцу 10
1.2. Строковые алгоритмы для сжатых данных 12
1.3. Алгебраические строковые алгоритмы 13
1.4. Выводы 14

2. Предлагаемый подход **15**

2.1. Необходимые алгебраические конструкции 15
2.2. Реализация умножения липких кос 20
2.3. Реализация рекурсивного подсчета наибольшей общей подпоследовательности 26
2.4. Наибольшая общая подпоследовательность на сжатых строках 27
2.5. Выводы 28

3. Адаптация алгоритма для форматов сжатия **29**

3.1. Сжатие LZ78 и LZW 29
3.2. Сжатие UNIX-compress 30
3.3. Выводы 31

4. Оценка алгоритма **32**

4.1. Эксперименты на искусственных строках 32
4.2. Эксперименты на реальных строках 34
4.3. Эксперименты на UNIX-compress 36
4.4. Выводы 39

Заключение **40**

Список литературы **42**

Приближенный поиск по образцу — хорошо изученная проблема для строк: по тексту t и образцу p необходимо найти все подстроки в t, похожие по какому-то критерию на p. Мера сходства, которую мы используем в этой работе, — это длина наибольшей общей подпоследовательности (НОП) двух строк. Мы реализуем алгоритм, который решает задачу приближенного поиска по образцу для образца p длины m и текста t длины n за время $O(mn)$, неявно сохраняя все значения НОП в алгебраической структуре размера $O(m + n)$.

Мы дополнительно обобщаем алгоритм для вычисления НОП несжатой строки-образца длины m и строки-текста длины n, сжатой контекстно-свободной грамматикой длины $\bar{n}$ за время $O(m\bar{n} \log m)$.

Используя то, что временная сложность этого алгоритма не зависит от длины несжатого текста, мы показываем, что алгоритм достигнет существенного ускорения для определенных типов сжатых текстов, даже с увеличением константы времени работы в результате сложных операций с алгебраическими структурами. Мы сравниваем время работы нашего алгоритма со стандартным алгоритмом динамического программирования и утилитой для приближенного поиска по образцу agrep и демонстрируем улучшение для некоторых специальных видов текстов в обоих случаях, хотя на случайных текстах наш алгоритм стандартным решениям проигрывает.

Ключевые слова: строковые алгоритмы, приближенный поиск по образцу, наибольшая общая подпоследовательность, контекстно-свободные грамматики.

Approximate pattern matching is a well-studied problem on strings: given a text t and a pattern p, find all substrings of t that are similar to p. The measure of similarity that we use in this work is the length of the longest common subsequence (LCS) of two strings. We implement an algorithm that solves the approximate pattern matching problem for a pattern p of length m and a text t of length n in time $O(mn)$ implicitly storing all the LCS values in an algebraic structure of size $O(m+n)$.

We additionally generalize the algorithm to calculate the longest common subsequence of a plain (uncompressed) pattern string of length m and a text string of length n, compressed by a context-free grammar of length $\bar{n}$ in time $O(m\bar{n}\log m)$.

Using the fact that this algorithm's time complexity does not depend on the length of the uncompressed text, we show that the algorithm will achieve a substantial speedup on specific types of compressed texts, even with the increase in the running time constant resulting from complex operations with algebraic structures. We compare the running time of our algorithm to the standard dynamic programming algorithm for approximate pattern matching and to the approximate pattern matching utility agrep, and we demonstrate an improvement for some specific types of text in both cases, even though our algorithm is less optimal than the standard solutions on random texts.

Keywords: string algorithms, approximate pattern matching, longest common subsequence, context-free grammar.

Введение

Актуальность и релевантные работы

Приближенный поиск по образцу — классическая строковая задача, у которой есть широкий ряд применений в различных областях. Например, приближенный поиск используется в проверке правописания и определении спама, а также во многих алгоритмов биоинформатики, таких как сравнение последовательностей нуклеотидов [8]. Поэтому эта задача широко изучается и существует как и множество теоретических работ [24, 16, 19], так и ряд программных реализаций [18, 14, 17] для приближенного поиска по образцу с различными метриками сходства строк.

Рассматриваемая в этой работе метрика сходства — длина НОП двух строк. Связанная с ней метрика, расстояние НОП — количество операций удаления и вставки символа которое требуется, чтобы преобразовать одну строку в другую. Можно заметить, что при таком преобразовании не будут модифицироваться элементы какой-то НОП, поэтому метрика в каком-то смысле обратна длине НОП. Расстояние НОП отличается от более стандартной метрики, называемой расстоянием Левенштейна [10], тем, что помимо операция вставки и удаления допустима также операция изменения символа. Метрика, связанная с НОП имеет то преимущество, что для нее предложен алгоритм приближенного поиска по образцу в текстах, сжатых КСГ [21], но в действительности простым преобразованием строк можно перевести одну метрику в другую и переделать алгоритм, работающий с одной из метрик, на другую [22].

Так как в современном мире обрабатываемые объемы данных становятся все больше и больше, файлы достаточно большого размера часто хранятся в сжатом виде. Рассматриваемое в этой работе сжатие — сжатие контекстно-свободной грамматикой, или КСГ. Эта достаточно простая математическая конструкция покрывает класс использующихся на практике алгоритмов сжатия, таких как LZ78 [28] и LZW [25], а

также UNIX-compress. Соответственно, обработка таких файлов и решение некоторых задачах на них без распаковки может потенциально оптимизировать время работы со сжатыми файлами. На файлах, сжатых подобным образом, существует ряд алгоритмов для решения различных строковых задач: точного поиска [15], поиска по регулярному выражению [29], приближенного поиска [6, 21]. Для некоторых из этих алгоритмов существуют также и практические реализации [15, 29], но для задачи приближенного поиска по образцу такой реализации нет ни для одного алгоритма сжатия КСГ.

Определения ключевых терминов

В данной работе рассматривается задача приближенного поиска по образцу (в оригинале — Approximate Pattern Matching).

Определение 0.1 (Задача приближенного поиска по образцу)**.** *Задача приближенного поиска по образцу требует для строки-образца p длины m и строки-текста t длины $n \geq m$ найти все подстроки текста, похожие на образец.*

В различных вариациях задачи приближенного поиска по образцу может требоваться вернуть либо все подстроки, отличающиеся от образца не более чем на какой-то порог, либо найти подстроку текста, максимально похожую на образец. В данной работе рассматривается вторая версия этой задачи. В данной работе в качестве меры сходства мы рассматриваем длину наибольшей общей подпоследовательности, или НОП, строки-образца и подстроки строки-текста.

Определение 0.2 (Наибольшая общая подпоследовательность)**.** *Задача о наибольшей общей подпоследовательности (longest common subsequence) требует для двух строк a, b найти наибольшую длину строки, являющейся подпоследовательностью как a, так и b. Эту длину мы будем в дальнейшем обозначать $lcs(a, b)$.*

Важная алгебраическая структура для данной работы — 0-моноид Гекке симметрической группы, или моноид липких кос.

Определение 0.3 (Моноид липких кос)**.** *Моноид липких кос порядка n, он же 0-моноид Гекке (0-Hecke Monoid) симметрической группы порядка n — моноид, порождаемый единичным элементом ι и $n-1$ образующими $g_1, \ldots, g_{n-1}$, задаваемых соотношениями*

- $g_i^2 = g_i \ \forall i \in [1 : n-1]$
- $g_i g_j = g_j g_i \ \forall i, j \in [1 : n-1],\ j - i \geq 2$
- $g_i g_j g_i = g_j g_i g_j \ \forall i, j \in [1 : n-1],\ j - i = 1$

Для краткости изложения, в данном документе в дальнейшем вместо термина "элемент 0-моноид Гекке симметрической группы" используются термины "коса" и "липкая коса", вместо термина "0-моноид Гекке симметрической группы" используется термин "моноид липких кос", а операция умножения в моноиде липких кос называется "липким умножением".

В данной работе мы будем рассматривать данные, сжатые с помощью контекстно-свободной грамматики.

Определение 0.4 (Контекстно-свободная грамматика)**.** *Контекстно-свободная грамматика, или КСГ, размера $\bar{n}$ – набор правил, где каждое имеет вид либо $t_k = \alpha$, где α — алфавитный символ, либо $t_k = t_i t_j$, для некоторых i, j, $1 \leq i, j < k$. Для удобства мы также разрешаем правило $t_k = \epsilon$, где ϵ — пустая строка.*

Мы говорим, что строка s сжата КСГ размера $\bar{n}$, если последнее правило этой грамматики $t_{\bar{n}}$ разжимается в строку s.

В дальнейшем в работе мы будем придерживаться тех же обозначений: m — длина строки-образца, n — длина строки-текста, $\bar{n}$ — размер КСГ, которой сжата строка-текст.

Цель и задачи

Данная работа ставит целью реализацию алгоритма [21] для приближенного поиска по образцу в данных, сжатых КСГ со временем работы

$O(m\bar{n}\log m)$, использующий как метрику сходства длину НОП и находящий подстроку текста, максимально похожую на образец. Также в цель данной работы входит проверка того, действительно ли данный алгоритм работает быстрее, чем стандартный алгоритм со временем работы $O(mn)$ и стандартные утилиты для приближенного поиска по образцу.
Для этого ставятся следующие задачи:

- Реализовать рекурсивный алгоритм решения задачи о НОП, на котором основывается рассматриваемый алгоритм приближенного поиска по образцу.
- Расширить этот алгоритм для эффективного приближенного поиска по образцу в сжатых данных.
- Реализовать декомпрессоры, переводящие несколько форматов сжатых данных в формат КСГ, использующийся алгоритмом. В данной работе рассматриваются данные, сжатые LZ78, LZW и UNIX-compress.
- Сравнить время работы алгоритма со стандартным алгоритмом динамического программирования для приближенного поиска по образцу в сжатых данных на строках.
- Сравнить время работы алгоритма с существующими утилитами для приближенного поиска на файлах, используя для этого файлы, сжатые использующимися на практике архиваторами.

Достигнутые результаты

В рамках данной работы был реализован алгоритм приближенного поиска по образцу для образца и текста, сжатого КСГ с асимптотикой $O(m\bar{n}\log m)$. Такая асимптотика достигается за счет реализованного алгоритма конкатенации алгебраических структур размера m, хранящих внутри себя информацию о НОП двух строк, за время $O(m\log m)$. Это дает возможность подсчета НОП с образцом в сжатом тексте с

помощью рекурсии по структуре КСГ. Информация о структуре КСГ — единственное знание о формате сжатого текста или файла, которое требуется алгоритму для его обработки. Таким образом, поддержка любого формата сжатия, задающегося КСГ заключается в предоставлении декомпрессора, переводящего сжатые данные в набор правил КСГ, задающие эти данные.

Также была реализована генерация различных строк и файлов, достигающих квадратичное сжатие рядом различных алгоритмов сжатия КСГ. На ее основе и на основе реальных текстов было проведено исследование времени работы реализованного алгоритма. Исходный код всех реализаций и некоторых дополнительных материалов для запуска исследований доступен на GitHub [1]. В качестве алгоритмов сжатия для исследования были использованы алгоритмы LZ78 и LZW, а также реализованная на основе LZW UNIX-утилита compress. Сравнение времени работы проводилось со стандартным динамическим программированием и с UNIX-утилитой agrep для приближенного поиска по образцу.

Структура работы

В главе 1 представлен обзор существующих теоретических работ и программных реализаций в данной области, история исследований задачи о приближенном поиске и подходов к специальной обработке сжатых данных без предварительной распаковки.

В главе 2 подробно описан рассматриваемый алгоритм приближенного поиска по образцу в сжатом тексте, а также наиболее важные детали его реализации.

В главе 3 описаны технические подробности генерации тестовых файлов для анализа времени работы алгоритма и построения КСГ по анализируемым форматам сжатия.

В главе 4 представлены результаты исследования времени работы реализованного алгоритма, сравнение времени его работы со стандарт-

[1] Реализация: https://github.com/crossopt/Recursive-LCS

ным динамическим программированием для приближенного поиска по образцу и с UNIX-утилитой agrep.

В заключении представлены анализ проделанной работы и возможные направления дальнейшего развития и продолжения работы.

1. Обзор литературы

Приближенный поиск по образцу — широко изучаемая задача теоретической информатики с рядом практических реализаций алгоритмов, ее решающих. Далее рассматривается история изучения этой задачи и различные предложенные подходы, а также имеющиеся практические реализации алгоритмов приближенного поиска.

Дополнительно рассматриваются различные алгоритмы для решения строковых задач на сжатых данных, в особенности задача приближенного поиска, и представляются существующие утилиты и реализации различных алгоритмов на сжатых данных.

1.1. Задача приближенного поиска по образцу

Термин “приближенный поиск по образцу” встречается в работе Ukkonen [23], где он анализирует и улучшает существующий алгоритм для вычисления редакционного расстояния между двумя строками. Тем не менее, хотя он использует термин “приближенный поиск по образцу”, Ukkonen не пытается явно посчитать сходство между образцом и всеми подстроками строки.

Более подробный подход к решению задачи приближенного поиска в том виде, как она рассматривается в данной работе, дал Sellers в своей работе [19]. Метод, который предложил Sellers основан на динамическом программировании, как и метод Ukkonen. Тем не менее, его результат — список пар подстрок входных двух строк вместо единственного значения. В этом списке в каждой паре присутствует по подстроке из изначальных двух строк, и каждая пара достигает максимальное локально возможное значение сходства, то есть Sellers ищет в явном виде сходства между подстроками двух строк. У данного подхода итоговое время работы $O(mn)$.

Конкретная метрика сходства, рассматриваемая в этой работе — длина НОП двух строк. Для решения задачи о НОП существует стандартный алгоритм динамического программирования, который был открыт независимо Wagner and Fischer [24], а также Needleman and Wunsch

[16]. Интересно, что в этих работах задача НОП рассматривается в контексте двух ее различных применений: первая работа рассматривает редакционное расстояние между двумя строками, в то время как вторая изучает последовательности аминокислот. Предложенный алгоритм за время $O(mn)$ считает длину НОП для двух строк и прост в реализации. В дальнейшем появился ряд работ, несущественно улучшающих асимптотику этого алгоритма с помощью различных оптимизаций. Например, Masek and Paterson [13] предложили решение задачи о НОП за $O(\frac{mn}{\log n})$. Более того, доказано [1, 4], что при верности гипотезы SETH не существует алгоритма для подсчета НОП двух строк длины n со временем $O(n^{2-\epsilon})$.

Существует ряд утилит и различного программного обеспечения, решающего задачу приближенного поиска по образцу на практике. Среди них есть решения, реализующие стандартный алгоритм динамического программирования. Пример такого решения – утилита SSEARCH, часть биологического пакета ПО FASTA [17], предназначенного для обработки последовательностей нуклеотидов. Также существуют и решения, достигающие выигрыш в производительности за счет реализации не стандартного динамического программирования, а алгоритма для специального вида задачи приближенного поиска. Например, это утилита agrep [26] и библиотека TRE [11], которые решают слегка более общую задачу, где рассматриваются различные метрики сходства, но решают ее только для ограниченной длины образца $m < 32$.

Более того, на практике часто используются различные эвристические решения для достижения уменьшения среднего времени работы за счет отказа от оптимальности найденного приближенного совпадения с образцом. Особенно это актуально для биологических применений задачи приближенного поиска по образцу, где зачастую встречаются большие длины текстов и образцов. Биологические пакеты ПО BLAST [14] и FASTA [17] предоставляют ряд эвристических решений, также пакет difflib [18] языка Python решает приближенную задачу эвристически, используя решения для точного поиска чтобы аппроксимировать ответ для приближенного. Более подробный обзор эвристических алгоритмов

для приближенного поиска дан Bucak and Uslan [5]. Тем не менее, в пределах данной работы эвристические алгоритмы для приближенного поиска по образцу не рассматриваются и задача приближенного поиска решается исключительно точно.

1.2. Строковые алгоритмы для сжатых данных

Модель сжатия, рассматриваемая в данной работе — сжатие КСГ, в котором по сжимаемой строке строится КСГ, ее задающая. Эта модель покрывает несколько различных методов сжатия, использующихся на практике. Два таких способа, LZ78 и LZW, были предложены Ziv and Lempel [28] и Welch [25]. Дополнительный пример сжатия КСГ — алгоритм, использующийся UNIX-утилитой compress, являющийся модификацией алгоритма LZW.

Так как большое количество сжатых данных используется как в науке, так и в промышленных применениях, вопрос о том можно ли эффективно обрабатывать сжатые данные без их предварительной распаковки был обширно изучен. Ранние примеры таких алгоритмов были предложены Amir et al. для сжатия UNIX-утилитой compress [3].

Несколько существующих работ рассматривают решение задачи о НОП на сжатых данных. В частности, для LZW-сжатия, являющимся частным случаем сжатия КСГ, Crochemore et al. [6] предлагают алгоритм для подсчета НОП в двух сжатых строках за время $O(\bar{n}m + \bar{m}n)$. Решение похожей задачи сравнения на окне (Window Subsequence) было предложено Cégielski et al. [27] за $O(\bar{n}m^2 \log m)$. В дальнейшем более быстрый алгоритм был предложен Тискиным [21] за $O(m\bar{n} \log m)$, далее улучшенный Gawrychowski [7] до $O(m\bar{n}\sqrt{\log \frac{\bar{n}+m}{\bar{n}}})$ оптимизациями существенно сложнее предыдущих алгоритмов.

Более подробный обзор существующих алгоритмов для строк, сжатых КСГ был дан Lohrey [12].

Для реализации приближенного поиска по образцу на сжатых данных без распаковки в текущий момент не существует реализованных программных утилит, тем не менее, существует ряд утилит для реше-

ния смежных строковых задач без распаковки на различных классах КСГ. Например, для строк, сжатых LZW существуют решения для точного поиска [15], поиска по регулярному выражению [29].

1.3. Алгебраические строковые алгоритмы

Альтернативное решение для задачи о НОП было предложено Тискиным [20]. Этот подход основан на сходстве между умножением в моноиде липких кос и поведением длины НОП при конкатенации строк. В этом подходе липкая коса, соответствующая паре строк строится итеративно за время $O(mn)$.

Алгоритм реализации умножения в моноиде липких кос n с асимптотикой $O(n \log n)$, также предложенный в [20], может быть применен для того чтобы использовать такое умножение как основу для рекурсивного подсчета НОП двух строк за время $O(mn)$ с помощью метода “разделяй-и-властвуй”. Этот метод также описан в [20], вместе с несколькими дополнительными теоретическими применениями рекурсивного подсчета длины НОП, таких как подсчет НОП несжатой строки со строкой, сжатой КСГ.

Предложенный алгоритм работает для сравнения строки-образца и строки-текста, сжатой КСГ, за время $O(m\bar{n} \log m)$, так как каждое правило КСГ обрабатывается с помощью алгоритма умножения липких кос за время $O(m \log m)$.

Важно заметить, что алгоритмическая сложность стандартных алгоритмов подсчета НОП зависит от длины несжатого текста, которая может теоретически расти экспоненциально относительно размера сжатого текста. Хотя в используемых на практике алгоритмах экспоненциальное сжатие не достигается, для алгоритмов LZ78, LZW и UNIX-compress существуют КСГ, сжимающие строки квадратичной длины, что тоже позволяет надеяться на существенное ускорение производительности при применении данного алгоритма.

1.4. Выводы

По результатам изучения работ в области приближенного поиска по образцу и в области работы на сжатых данных были сделаны следующие выводы:

- Среди существующих подходов к приближенному поиску по образцу можно выделить эвристические подходы, подходы решающие задачу в частных случаях и подходы, решающие задачу приближенного поиска точно.
- Большинство теоретических алгоритмов для приближенного поиска по образцу основаны на динамическом программировании, и ряд этих подходов реализованы и находятся в открытом доступе в качестве утилит и библиотек.
- Теоретически описано точное решение задачи приближенного поиска по образцу, работающее в данных, сжатых КСГ, без предварительной распаковки, за время $O(m\bar{n}\log m)$, но программной реализации алгоритма приближенного поиска по образцу на сжатых данных не существует.

2. Предлагаемый подход

В этой главе детально описывается рассматриваемый алгоритм для решения задачи приближенного поиска по образцу в сжатых данных [21], а также рекурсивный алгоритм для поиска НОП [20], на котором он основывается. Исходный код реализаций рассматриваемых алгоритмов выложен в открытый доступ на GitHub.

2.1. Необходимые алгебраические конструкции

Ранее в данной работе было введено определение липких кос. Рассмотрим этот объект поподробнее: липкие косы порядка n можно представить геометрически так: нарисуем две параллельные горизонтальные линии в евклидовой плоскости и выберем на каждой линии набор из n точек, называемых узлами. Два набора узлов могут быть сопоставлены попарным соединением непрерывными монотонными кривыми, называемыми нитями. Пересечения соседних нитей в таком представлении соответствуют образующим элементам группы.

Определение 2.1 (Причесывание липкой косы)**.** *У каждой липкой косы есть эквивалентная ей приведенная липкая коса: коса, где каждая пара нитей пересекается не более одного раза. Под эквивалентностью двух кос понимается их равенство как элементов моноида, то есть две косы эквивалентны, если одну из них можно привести к другой вышеопределенными соотношениями.*

Для того, чтобы "причесать", или привести косу, можно пройти по всем пересечениям нитей сверху вниз. Если пара нитей уже пересекалась, можно убрать пересечение и заменить его на два непересекающихся куска. Это может быть доказано применением соотношений, задающих моноид.

Липкой косе можно поставить в соответствие перестановочную матрицу порядка n, переводящую верхние узлы косы в соответствующие им нижние. При этом одной перестановочной матрице могут соответ-

ствовать несколько различных кос, но все приведенные косы, соответствующие одной перестановочной матрице эквивалентны.

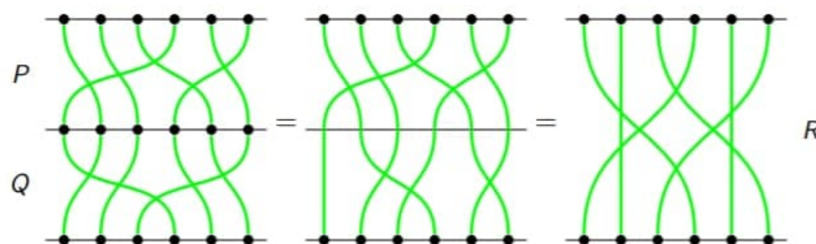


Рис. 1: Умножение липких кос

Геометрически операция умножения двух липких кос порядка n представлена на рисунке 1: первая умножаемая коса рисуется над второй, затем мы объединяем каждую пару нитей, которые стали инцидентными, и получается одна коса-произведение той же ширины. Она не обязательно будет приведенной, даже если оба множителя таковыми являлись, но можно после умножения косу дополнительно причесать и получить приведенное произведение двух липких кос. Важность данной структуры для этой работы состоит в том, что существует [20] алгоритм умножения липких кос порядка n с приведением за $O(n \log n)$.

С помощью этой структуры мы будем рассматривать не собственно задачу НОП, а чуть более общую задачу о полулокальной НОП.

Определение 2.2 (Задача о полулокальной НОП)**.** *Задача о полулокальной НОП (semi-local LCS problem) требует для двух строк a, b уметь отвечать на запросы:*

- *НОП строки a со всеми подстроками b*
- *НОП каждого префикса a с каждым суффиксом b*
- *НОП каждого суффикса a с каждым префиксом b*
- *НОП всех подстрок a со строкой b*

Определение 2.3 (Сетка НОП)**.** *Граф НОП-сетка для строк a, b (LCS grid) $G_{a,b}$ — это взвешенный граф, вершины которого являются элементами $\{0 \ldots m\} \times \{0 \ldots n\}$, где $m = |a|$, $n = |b|$. Ребра графа ведут*

из $v_1 = (x, y)$ *в* $v_2 = (x, y+1)$ *и* $v_3 = (x+1, y)$ *с весом* 0. *Также, если символы* a_x *и* b_y *совпадают, то из* v_1 *в* $v_4 = (x+1, y+1)$ *ведет ребро веса* 1.

Геометрически граф $G_{a,b}$ можно представить в виде вершин — узлов сетки клетчатой бумаги и ребер — сторон клеток. Координаты введем таким образом, чтобы $v_0 = (0, 0)$ лежало в левом верхнем углу, а первая координата росла вниз, вторая — вправо. Таким образом на графе будут горизонтальные и вертикальные ребра веса 0, а также некоторые диагональные ребра веса 1.

Определение 2.4 (Ячейки НОП-сетки)**.** *Будем называть ячейкой квадратную ограниченную грань или две треугольные ограниченные грани геометрического вложения графа-сетки, соответствующие одной клетке клетчатой бумаги. При этом ячейка* (x, y) *— ячейка, соответствующая клетке клетчатой бумаги с правой нижней координатой в* (x, y).

Определение 2.5 (Границы и граничные вершины)**.** *Будем называть вершину* (x, y) *вершиной на верхней границе или вершиной верхней границы, если у нее* $x = 0$. *Аналогичный критерий для нижней границы* $x = m$, *для левой границы* $y = 0$ *и правой* $y = n$.

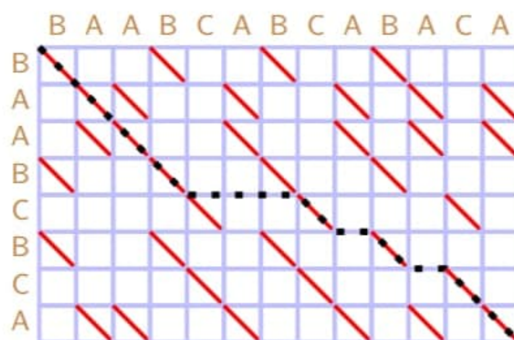


Рис. 2: Пример НОП-сетки

Можно заметить, что $lcs(a, b)$ будет естественным образом соответствовать максимальный путь из вершины $(0, 0)$ в (m, n). НОП-сетка для строк $BAABCABCABACA$ и $BAABCBCA$ вместе с путем, соответствующим их НОП представлена на рисунке 2.

Более того, эта конструкция позволяет проще представить решение задачи полулокального НОП: все общие подпоследовательности соответствуют некоторому пути по этому графу от одной границы до другой.

В частности:

- общая подпоследовательность строки a с подстрокой строки b будет соответствовать пути от верхней границы до нижней
- общая подпоследовательность префикса a с суффиксом b будет соответствовать пути от верхней границы до правой
- общая подпоследовательность суффикса a с префиксом b будет соответствовать пути от левой границы до нижней
- общая подпоследовательность подстроки строки a со строкой b будет соответствовать пути от левой границы до правой

Анализ путей между границами НОП-сетки можно упростить, добавив в начало и конец строки b по m символов-джокеров '?', при сравнении совпадающих с любым символом. Мы получим строку b_{pad} длины $2m + n$. Тогда интересующие нас подпоследовательности совпадают с путями от верхней границы до нижней в НОП-сетке $G_{a,b_{pad}}$.

Решить задачу полулокальной НОП для пары строк a, b можно с помощью так называемой матрицы НОП $H_{a,b}$.

Определение 2.6 (Матрица НОП)**.** *Матрица НОП $H_{a,b}$ (LCS matrix) для строк a, b — матрица размера $(m + n) \times (m + n)$, где по первой координате индексы принимают значения от $-m$ до n, а по второй — от 0 до $m + n$.*

$H[i, j]$ определяется как длина максимального пути $lcs(a, b_{pad}[i : j])$ в НОП-сетке $G_{a,b_{pad}}$ от вершины $(0, i)$ до вершины (m, j). При этом, если $i = j$, то $H[i, j] = 0$, а если $j < i$, то $H[i, j] = j - i$.

Соответственно, все значения, требуемые для решения задачи полулокальной НОП, можно получить из элементов матрицы НОП.

Определение 2.7 (Ядро НОП)**.** *Матрица перекрестных разностей для матрицы A размера $(m+1) \times (n+1)$ — матрица размера $m \times n$, у которой в элементе (i, j) записано значение $A(i, j+1) + A(i+1, j) - A(i, j) - A(i+1, j+1)$.*

Перестановочная матрица $P_{a,b}$, являющаяся матрицей перекрестных разностей матрицы $-H_{a,b}$, называется ядром НОП (LCS kernel).

Этой же перестановочной матрице соответствует липкая коса порядка $m + n$. Геометрически это соответствие можно представить так: изобразим нити косы, идущие слева направо сверху вниз по ячейкам графа-сетки таким образом, что каждая нить начинается из одного из ребер верхней или левой границы и заканчивается в одном из ребер нижней или правой границы. Аналогично конструкции НОП-сетки, если символы a_x и b_y не совпадают, то нитки в клетке (x, y) будут пересекаться, а если символы совпадают, то нитки пересекаться не будут. В результате этих операций мы получим неприведенную косу, которую можно в дальнейшем причесать и получить приведенную липкую косу.

Мы еще будем в дальнейшем пользоваться соответствием между перестановочными матрицами и липкими косами. Для дополнительного прояснения конструкции, можно заметить что в таком представлении нити липкой косы соответствуют путям в двойственном графе.

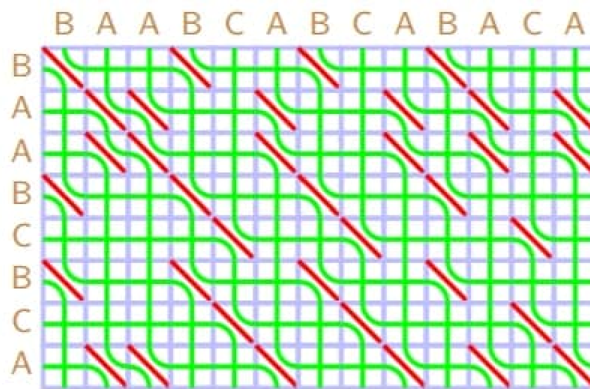


Рис. 3: Липкая коса на НОП-сетке

Рассмотрим приведенную липкую косу, идущую по ячейкам графа-сетки, как изображено на рисунке 3. В каждую ячейку НОП-сетки приходят две нити, слева и сверху, и продолжаются вниз и вправо. Ячейки,

где нити пересекаются, то есть одна идет сверху вниз, а другая слева направо, назовем ячейками пересечения. Ячейки, где нити не пересекаются, то есть одна идет сверху направо, а другая слева вниз, назовем ячейками непересечения.

Таким образом, задача полулокального НОП сводится к подсчету ядра НОП: по ядру НОП можно посчитать матрицу НОП, из элементов которой можно получить все значения, требуемые для решения задачи полулокального НОП.

Один из возможных способов посчитать ядро НОП — рекурсивным алгоритмом привести вышеописанную косу, и посчитать соответствующую полученной приведенной косе перестановочную матрицу, которая и будет являться искомым ядром.

2.2. Реализация умножения липких кос

В данной работе реализован алгоритм [20], позволяющий умножать косы порядка n за время $O(n \log n)$. Здесь будут описаны только некоторые детали реализации без необходимых доказательств. Липкое перемножение двух перестановочных матриц, соответствующих липким косам, производится с помощью метода “разделяй-и-властвуй”:

Липкое перемножение двух матриц порядка 1 тривиально. Для липкого перемножения двух перестановочных матриц большего порядка n P и Q требуется перемножаемые матрицы разделить на два блока каждую: P по вертикали, Q по горизонтали.

Для корректности дальнейшей работы алгоритма требуется, чтобы ширина блоков P соответствовала высоте блоков Q. Таким образом, матрица P будет разделена на блоки размера $n \times k$ и $n \times (n - k)$, а матрица Q будет разделена на блоки размера $k \times n$ и $(n-k) \times n$. При этом для достижения желаемой асимптотики времени липкого умножения $O(n \log n)$ необходимо брать k примерно равным $\frac{n}{2}$.

Получившиеся блоки не будут перестановочными матрицами, так как в них есть нулевые строки или стобцы. Тем не менее, можно заметить, что при перемножении двух блоков нулевая строка в любом из

блоков P будет соответствовать нулевой строке в произведении, и аналогично нулевой столбец в любом из блоков Q в перемножаемом блоке будут соответствовать нулевому столбцу в произведении. Таким образом, можно сжать координаты, удалив из блоков P нулевые строки, а из блоков Q нулевые столбцы. Получившиеся после сжатия координат блоки окажутся перестановочными матрицами, которые можно перемножить двумя рекурсивными вызовами. После перемножения блоков для восстановления координат требуется в получившиеся матрицы вернуть удаленные строки и столбцы. Для удобства назовем две матрицы, получившиеся в результате этих операций R_{lo} и R_{hi}.

Основные требования на данном этапе алгоритма к структуре данных, хранящей перестановочную матрицу — возможность за время $O(n)$ делить ее на две части как по первому индексу, так и по второму, а также возможность за время $O(n)$ реализовывать сжатие координат, приводя полученные половины матриц к квадратной форме $k \times k$ или $(n-k) \times (n-k)$.

В исходной статье сжатие кусков перестановочных матриц предлагается делать явно, с помощью сжатия координат, тем не менее, в имеющейся реализации это сделано неявно. Основная мотивация при сжатии координат это необходимость сделать из блоков размера $n \times k$ и $k \times n$ блоки размера $k \times k$, а из блоков размера $n \times (n-k)$ и $(n-k) \times n$ блоки размера $(n-k) \times (n-k)$. Это требуется для того, чтобы суммарное время вызовов на следующем уровне рекурсии не увеличивалось.

Назовем строковым индексом единичного элемента перестановочной матрицы индекс строки, в котором он находится. Соответственно, следующая по строковому индексу единица — единичный элемент в следующей строке. Аналогично введем столбцовый индекс и следующую по столбцовому индексу единицу. Традиционный способ хранения перестановочных матриц — массив чисел, в элементе i которого хранится столбцовый индекс единицы в строке i. Как альтернатива текущая реализация хранит массив пар чисел, где каждая пара (i, j) — пара индексов единичных элементов перестановочной матрицы. Тогда при разделении матрицы на блоки никаких явных операций для сжатия

координат не требуется, так как в каждой половине уже хранятся требуемые k и $n-k$ пар индексов соответственно. Тем не менее, необходимо при дальнейшей реализации алгоритма помнить, что формально объект, с которым алгоритм работает не перестановочная матрица порядка k, а разреженная подматрица перестановочной матрицы, в которой не более k единиц.

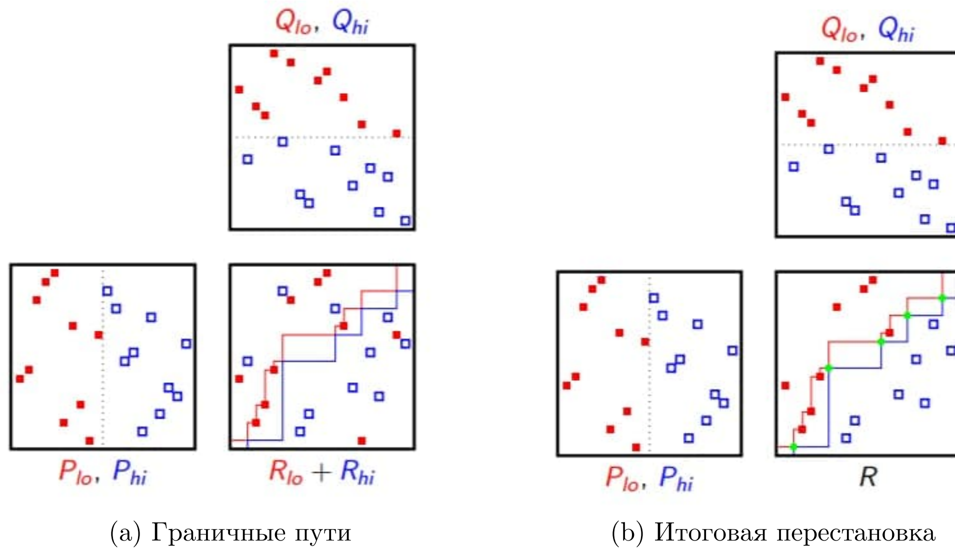
Такой способ хранения был выбран из соображений эффективности и простоты реализации: сжатие координат требовало бы перестроения перестановочной матрицы на каждом шаге, а также возвращение исходных координат в конце алгоритма, тогда как хранение исходных координат позволяет подобные манипуляции избежать.

В фазе "властвования" решение исходной задачи получается из решений подзадач R_{lo}, R_{hi} так: определим $\Delta[i,j]$ как разность между количеством единичных элементов R_{hi} строго левее и нестрого выше (i,j) и количеством единичных элементов R_{lo} нестрого правее и строго ниже (i,j).

Тогда существует монотонный путь из левого нижнего элемента в правый верхний с шагами, параллельными осям координат по целым координатам такой, что все целые точки, где Δ отрицательно лежат строго левее и нестрого выше этого пути, а все целые точки, где Δ неотрицательно лежат нестрого правее и строго ниже этого пути. Также существует путь с шагами, параллельными осям координат по целым координатам такой, что все целые точки, где Δ неположительно лежат строго левее и нестрого выше этого пути, а все целые точки, где Δ положительно лежат нестрого правее и строго ниже этого пути. Назовем эти пути левым граничным путем и правым граничным путем соответственно.

Левый граничный путь и правый граничный путь не могут пересекаться (это несложным образом следует из их определения). Тем не менее, они могут в некоторых точках касаться. Эти точки касания нас и будут интересовать.

Для получения итогового решения требуется к перестановочной матрице $R_{lo} + R_{hi}$ добавить элементы, лежащие на точках касания левого



(a) Граничные пути

(b) Итоговая перестановка

Рис. 4: Липкое умножение перестановочных матриц

и правого граничных путей, а также убрать элементы R_{lo}, лежащие выше левого граничного пути и элементы R_{hi}, лежащие ниже правого граничного пути. Данные операции представлены на рисунке 4.

Неформально подобные операции могут быть описаны следующим образом:

Муравей может стоять в произвольной целой точке в квадрате $n \times n$ и умеет двигаться на одну клетку вправо или вверх. Изначально он стоит в левом нижнем углу. У муравья есть два глаза: один глаз наблюдает верхний левый квадрант матрицы R_{hi}, симметрично другой глаз наблюдает нижний правый квадрант матрицы R_{lo} (оба квадранта — с основанием в текущей позиции муравья).

Назовем те единицы, которые видит муравей плохими, и поставим дополнительное условие на движение муравья: количество плохих единиц должно быть равным для обоих глаз в любой момент времени. Соответственно, единицы, которые муравей ни в какой момент времени не увидел назовем хорошими.

В моменты, когда муравей не может подвинуться ни вверх, ни вправо, требуется добавить новую единицу в итоговую матрицу, и подвинуть муравья по диагонали. То есть в итоговой перестановочной матрице будут хорошие единицы $R_{lo} + R_{hi}$, а также единицы, добавленные при диагональных шагах муравья.

Для того, чтобы понять, какие единицы из исходных двух перестановочных матриц войдут в итоговую перестановочную матрицу, требуется эмулировать движение “муравья” по перестановочной матрице, т.е. уметь в любой момент за $O(1)$ времени находить следующие по столбцовому и строковому индексу единицы перестановочной матрицы.

Несложно найти следующую по строковому индексу единицу, если хранить перестановочную матрицу как набор из пар строковых и столбцовых индексов единиц, отсортированных по строковым индексам, и несложно найти следующую по столбцовому индексу единицу, если хранить перестановочную матрицу как набор из пар столбцовых и строковых индексов единиц, отсортированных по столбцовым индексам. В таком случае, движение “муравья” сводится к одновременному проходу указателями по двум наборам, задающих перестановочную матрицу.

Отдельно можно отметить, что построение итоговой перестановочной матрицы необязательно делать inplace. Это связано с тем, что при таком проходе по единицам перестановочных матриц R_{lo} и R_{hi} как те единицы из R_{lo} и R_{hi}, которые будут единицами итоговой перестановочной матрицы, так и новые единицы, которые добавятся при диагональных переходах муравья будут обходиться ровно в том порядке, в котором они должны содержаться в отсортированном представлении итоговой матрицы.

Таким образом, в реализации алгоритма движение муравья будут эмулировать два указателя: один будет двигаться по строковым индексам единиц из R_{lo} и R_{hi} в порядке их убывания, другой будет двигаться по столбцовым индексам из R_{lo} и R_{hi} в порядке их возрастания. Поскольку в представлении обеих матриц хранятся списки пар индексов отсортированных как по строковым индексам, так и по столбцовым, переход к следующему интересующему нас индексу осуществляется выбором одного из двух следующих индексов.

В момент, когда указатель муравья переходит от единицы к следующему значению как по строковому индексу, так и по столбцовому, нам известны его текущие строковые и столбцовые индексы. Соответствен-

но, для этой единицы мы можем проверить, является ли она хорошей, то есть необходимо ли добавить ее в итоговый ответ. Также при увеличении столбцового индекса и уменьшении строкового индекса при диагональном переходе мы гарантированно знаем, что текущую позицию муравья необходимо добавить в итоговую перестановочную матрицу. В итоге в новую перестановочную матрицу пары индексов, отсортированные по строковым индексам будут добавляться в порядке прохода по строковым индексам, а пары индексов, отсортированные по столбцовым индексам будут добавляться в порядке прохода по столбцовым индексам. Это значит, что после добавления все интересующие нас значения уже записаны в отсортированном порядке, и нигде в алгоритме явной сортировки нет.

Для наглядности приведем пример хранения перестановочной матрицы в алгоритме липкого умножения. Рассмотрим перестановочную матрицу порядка 5

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Эта матрица хранит перестановку, которая переводит 1 в 2, 2 в 1, 3 в 4, 4 в 5, 5 в 3. Соответственно, для хранения матрицы A будут храниться два вектора. В одном будут храниться пары из индексов строк и столбцов единичных элементов A, отсортированных по индексу строки в порядке прохода по ним муравья, то есть по убыванию. В другом будут храниться пары из индексов столбцов и строк единичных элементов A, отсортированных по индексу столбца в порядке прохода по ним муравья, то есть по возрастанию. Таким образом, будут храниться вектора:

$$A.rows = \{\{5,3\},\{4,5\},\{3,4\},\{2,1\},\{1,2\}\}$$
$$A.cols = \{\{1,2\},\{2,1\},\{3,5\},\{4,3\},\{5,4\}\}$$

На этом примере может не быть понятно, зачем хранить пары из индексов. Тем не менее, можно заметить, что при дальнейших операциях с этой матрицей координаты не обязательно будут оставаться последовательными. Так, при разделении A с $k = 3$ по вертикали мы получим две разреженные матрицы A_{lo} и A_{hi}:

$$A_{lo}.rows = \{\{5,3\},\{2,1\},\{1,2\}\},\ A_{hi}.rows = \{\{4,5\},\{3,4\}\}$$
$$A_{lo}.cols = \{\{1,2\},\{2,1\},\{3,5\}\},\ A_{hi}.cols = \{\{4,3\},\{5,4\}\}$$

При таком формате хранения перестановочной матрицы итерация по строкам и столбцам в процессе алгоритма будет выглядеть практически идентично.

2.3. Реализация рекурсивного подсчета наибольшей общей подпоследовательности

Ядро НОП можно считать с помощью рекурсивного липкого умножения кос: разбить сетку на две примерно равные части, посчитать для частей ядро рекурсивно, и перемножить две получившиеся косы.

Так как дерево рекурсии доминируется нижним уровнем, на котором происходит $O(mn)$ умножений матриц порядка $O(1)$, суммарное время работы рекурсивного алгоритма $O(mn)$.

Липкое умножение — наиболее ресурсоемкая часть рекурсивного алгоритма подсчета НОП. Для реализации липкого умножения, необходимо выполнять ряд нетривиальных операций с перестановочными матрицами, из-за чего вычисление НОП с помощью липкого умножения имеет существенно большую константу чем стандартный алгоритм динамического программирования, хоть и имеет хоть и имеет асимпотику схожую с ним.

2.4. Наибольшая общая подпоследовательность на сжатых строках

Рекурсивный алгоритм вычисления НОП полезен в нашей задаче, когда одна из строк задана не явно, а с помощью ряда рекурсивных правил.

Подсчет НОП образца p и текста t, заданного КСГ ведется образом, похожим на обычный рекурсивный алгоритм подсчета НОП. Основное идейное различие будет в том, что мы будем считать не всю липкую косу — ядро НОП для p и t — а только те ее нити, которые ведут от левой границы или приходят в правую. Обозначим этот кусок ядра $P_{p,t}^{sub}$. Поскольку нитей, ведущих от левой границы не более m и нитей, приходящих в правую границу не более m, всего суммарно нас будут интересовать не более $2m$ нитей. Это значит, что порядок $P_{p,t}^{sub}$ не более $2m$, и хранение только этого куска косы является существенной оптимизацией по сравнении с максимальным возможным порядком всего ядра НОП $m+n$, так как n может быть существенно больше m.

Рекурсия алгоритма будет не по обеим строкам, а только по t. В случае, когда $\bar{n}=1$ сетка НОП состоит из одного столбца высоты m, а $P_{p,t}^{sub}$ может быть посчитано итеративно за время $O(m)$.

В случае, когда $\bar{n}>1$, правило задающее t имеет вид $t=t't''$. Посчитаем рекурсивно куски $P_{p,t'}^{sub}$ и $P_{p,t''}^{sub}$. Нити из $P_{p,t'}^{sub}$, которые ведут от левой границы в нижнюю, а также нити из $P_{p,t''}^{sub}$, которые ведут от верхней границы в правую, так и будут в $P_{p,t}^{sub}$ вести от левой границы в нижнюю и от верхней границы в правую соответственно, и следовательно могут быть опущены из вызова алгоритма липкого умножения. Чтобы восстановить оставшиеся нити $P_{p,t}^{sub}$, липко перемножим описанным ранее алгоритмом куски кос, содержащие нити, соприкасающиеся с общей границей двух имеющихся НОП-сеток: правой для $P_{p,t'}^{sub}$ и левой для $P_{p,t''}^{sub}$.

При этом важно помнить, что хотя во всех рассматриваемых ядрах количество нитей не более $2m$, сами индексы нитей могут быть достаточно большими и в целочисленные типы данных не помещаться. Чтобы избавиться от связанных с этим проблем, можно удалить из

ядра столбцы, в которых нет единичных элементов, и перенумеровать координаты. После подобных преобразований все номера нитей будут порядка $O(m)$ и оперировать с ними можно будет без проблем.

Суммарное время работы описанного алгоритма — $O(m\bar{n}\log m)$, так как каждый из $\bar{n}$ рекурсивных шагов алгоритма доминируется временем работы алгоритма липкого умножения кос, то есть $O(m\log m)$.

Для завершения вычисления НОП осталось понять, как по $P^{sub}_{p,t}$ посчитать НОП строк p и t. Это делается аналогично с подсчетом НОП по ядру: количество нитей, ведущих от левой границы в правую равно количеству символов p, не входящих в какую-то конкретную НОП. Соответственно, для вычисления НОП требуется посчитать это количество и вычесть его из длины p.

2.5. Выводы

В данной работе реализован алгоритм для решения задачи приближенного поиска по образцу в данных, сжатых КСГ. В отличие от существующих подходов для решения задачи приближенного поиска, предложенный алгоритм решает задачу, не распаковывая сжатый текст предварительно, и соответственно время его работы зависит только от размера сжатого текста, а не от длины несжатого текста, которая для стандартных алгоритмов сжатия может расти квадратично от размера сжатого текста.

3. Адаптация алгоритма для форматов сжатия

Поскольку реализованный алгоритм никак не зависит от формата сжатия явно, и его поведение задается только структурой КСГ, которая ей передается, для корректной работы алгоритма с форматом сжатия необходимо реализовать декомпрессор, за линейное от размера файла время восстанавливающий структуру грамматики по сжатому файлу. Линейное время работы необходимо для сохранения теоретической асимптотики алгоритма, так как его время работы зависит линейно от размера сжатого файла.

3.1. Сжатие LZ78 и LZW

LZ78 и LZW — достаточно давно предложенные алгоритмы сжатия, удобные в частности лаконичностью их реализации.

При сжатии LZ78 создается пустой словарь фраз. По мере сжатия алгоритм просматривает текст символ за символом слева направо и для текущей позиции находит префикс максимальной длины, совпадающий с какой-то фразой из словаря. После этого код фразы подается на выход, и в словарь добавляется новая фраза, удлиненная на один символ, следующий за указанным префиксом. LZW — модификация LZ78, при которой изначально словарь фраз создается не пустым, а содержит односимвольные фразы из всех символов алфавита.

Поскольку одной стандартной реализации для сжатия LZ78 и LZW нет, можно ее реализовать самим, и соответственно реализовать сжатие сразу в формат КСГ. Для этого дополнительно необходимо понять, каким образом в процессе сжатия строится грамматика.

Определим грамматику, задающую общий вид LZ78 и LZW сжатия: сжатие LZ78 и LZW может быть описано в виде КСГ, состоящей из трех секций:

- в первой секции, все правила имеют вид $t_k = \alpha$;

- во второй секции, первое правило имеет вид $u_k = \epsilon$ и все последующие правила имеют вид $u_k = u_i t_j$, где $i < k$;
- в третьей секции, первое правило имеет вид $v_k = \epsilon$ и все последующие правила имеют вид $v_k = v_{k-1} u_j$.

Соответственно, последнее правило третьей секции v_{last} задает итоговую строку, сжатую КСГ. Назовем КСГ такой формы LZ-грамматикой.

В таком представлении несложно заметить, что второй секции соответствует добавление нового слова в словарь, а третьей – приписывание слова к итоговому ответу. Соответственно, для сжатия LZ78 и LZW было реализовано сжатие строк в грамматику данного вида.

3.2. Сжатие UNIX-compress

При сжатии UNIX-compress возникают дополнительные сложности, так как это не теоретически описанный алгоритм, а одна вполне конкретная утилита. Значит, для обработки файлов, сжатых UNIX-compress, необходимо изучить структуру сжатия и честно написать декомпрессор, по сжатому файлу восстанавливающий КСГ.

Задача усложняется тем, что не удалось найти хорошей документации для UNIX-compress, в котором описана структура сжатия в больших деталях, чем упоминание, что это модификация LZW. Хотя UNIX-compress — open-source утилита, ее исходный код сложен для понимания и с проблемой отсутствия понятной структуры компрессора и декомпрессора уже сталкивались другие люди, а на сайте StackExchange [2] была приведена реализация декомпрессора UNIX-compress на языке С.

Соответственно, для поддержки UNIX-compress необходимо было изучить предложенную реализацию и модифицировать ее, строя КСГ вместо итоговой строки. Для этого при добавлении строки в словарь необходимо добавить в грамматику соответствующее правило второй секции, а вместо добавлении строки в конец итоговой строки-ответа необходимо добавить в грамматику соответствующее правило третьей секции.

3.3. Выводы

В рамках данной работы были реализованы алгоритмы сжатия LZW и LZ78, преобразующие несжатую строку в формат КСГ, использующийся алгоритмом. Декомпрессор для строки, сжатой таким образом тривиален. Дополнительно был адаптирован существующий декомпрессор данных для сжатия UNIX-compress для распаковки данных не в несжатую строку, а в КСГ. Это дает возможность сжимать несжатые строки написанными реализациями LZW и LZ78 и запускать на них реализованный алгоритм, а также запускать реализованный алгоритм на файлах, сжатых UNIX-compress.

4. Оценка алгоритма

Для оценки производительности новой реализации алгоритма приближенного поиска было проведено сравнение времени работы реализованного алгоритма с существующими решениями дли задачи приближенного поиска по образцу.

При сравнении все эксперименты повторялись 20 раз на различных сгенерированных образцах и текстах для усреднения итогового результата.

Дополнительно для наглядности тестирования была зафиксирована длина образца, 16, так как основной выигрыш производительности реализованного алгоритма предполагается за счет улучшенной обработки сжатого текста, и для различных длин образцов результаты отличаются не существенно. Все образцы генерировались случайно из набора символов латинского алфавита.

4.1. Эксперименты на искусственных строках

В рамках данной работы была реализована генерация строк, достигающих квадратичное сжатие алгоритмами LZ78 и LZW. Поскольку процесс генерации отличается несодержательными техническими деталями, подробно описана будет только генерация строк, сжимающихся квадратично LZW.

Напомним, что в данном алгоритме при сжатии создается словарь фраз, который инициализируется односимвольными фразами из всех символов алфавита. По мере сжатия алгоритм просматривает текст символ за символом слева направо и для текущей позиции находит префикс максимальной длины, совпадающий с какой-то фразой из словаря. После этого код фразы подается на выход, и в словарь добавляется новая фраза, удлиненная на один символ, следующий за указанным префиксом.

Можно заметить, что более чем квадратичное сжатие для алгоритма LZW не может достигаться: если рассмотреть строку длины n, состоящую из n повторов одного символа, в ее представлении в качестве

словаря будут входить последовательно увеличивающиеся на один в длине строки, а соответственно, их будет $\Theta(\sqrt{n})$.

Тем не менее, возможно построить менее тривиальные примеры строк, сжимаемых квадратично LZW. Структура LZ-грамматики упрощает построение строк, сжимающихся квадратично: например, квадратичное сжатие достигается в частном случае, когда во второй секции LZ-грамматики все правила после первого имеют вид $u_k = u_{k-1}t_j$. В таком случае, каждая строка, добавляющаяся в словарь при сжатии LZW на один длиннее предыдущей, и структура получившейся грамматики будет аналогична КСГ для n повторов одного символа.

Дополнительно наложим условие на правила третьей секции: пусть все правила после первого имеют вид $v_k = v_{k-1}u_k$. В таком случае, строка s_{last}, в которую разжимается последнее правило второй секции u_{last} однозначно задает итоговую строку: в таком случае, все правила второй секции можно восстановить, так как они получаются последовательным добавлением символов s_{last} к пустой строке. Можно восстановить и итоговую строку, так как вследствие дополнительного ограничения на правила третьей секции она получается последовательной конкатенацией строк, в которые разжимаются правила второй секции.

Для наглядности приведем пример построения строки, сжимаемой квадратично LZW, по изначальной строке 'ABCDE'. КСГ такого построения имеет следующий общий внешний вид:

$$\begin{array}{lll} t_0 = \alpha_0 & t_1 = \alpha_1 & \dots \; t_n = \alpha_n \\ u_0 = \epsilon & v_0 = \epsilon & \\ u_1 = t_0 t_{i_1} & v_1 = v_0 u_1 & \\ u_2 = u_1 t_{i_2} & v_2 = v_1 u_2 & \\ \dots & \dots & \\ u_r = u_{r-1} t_{i_r} & v_r = v_{r-1} u_r & \end{array}$$

где α_k – k-й символ алфавита, а $i_1 i_2 .. i_r$ — символы строки t.
Таким образом, построенная по строке 'ABCDE', в которую разжимается правило u_{last}, КСГ генерирует строку
"$AAAABAABCAABCDAABCDE$" (разбиение на правила второй сек-

ции здесь следующее:

"$AA\ AAB\ AABC\ AABCD\ AABCDE$"). Заметим, что в таком случае длина итоговой строки действительно будет расти квадратично от длины строки s_{last}: в первой секции правил константное число, равное размеру алфавита, а во второй и третьей секции правил $\Theta(|s_{last}|)$. Длина итоговой строки это сумма длин строк для всех правил второй секции, т.е. $\Theta(|s_{last}|^2)$.

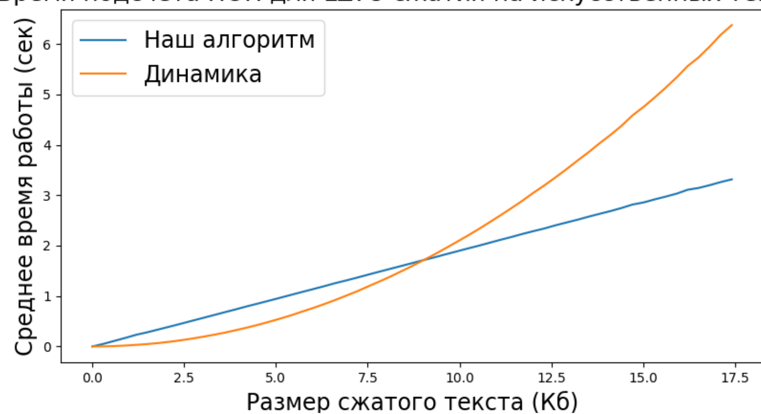
Построения для LZ78 аналогичны. Поскольку UNIX-compress является некоторой модификацией LZW, теоретически возможно аналогичное построение для UNIX-compress. Тем не менее, структура UNIX-compress устроена сложнее, чем LZW и LZ78, и отсутствует хорошая документация этой структуры, поэтому в качестве альтернативы данному построению в качестве строк, сжимаемых квадратично UNIX-compress были взяты строки, состоящие из n повторов одного символа.

Подобная структура имеет смысл в качестве входных данных для эксперимента, потому что наиболее наглядно видно улучшение времени работы по сравнению со стандартными решениями именно когда размер сжатой текстовой строки сильно меньше, чем несжатой.

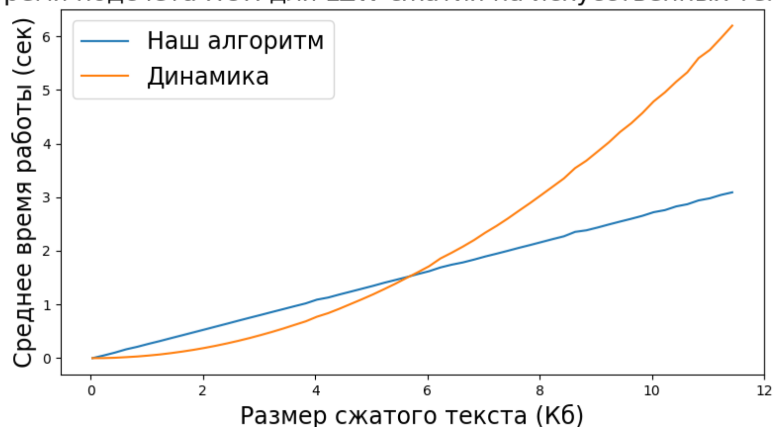
На рисунке 6 приведен график, на котором показано время работы подсчета НОП для специально построенных текстов, квадратично сжимающихся LZW и LZ78, относительно времени работы стандартного динамического программирования для различных размеров сжатых текстов. На данном графике можно увидеть, что хотя для маленьких размеров текстов динамическое программирование работает быстрее, с некоторого момента выигрыш от обработки текстов без распаковки обеспечивает более быстрое выполнение реализованного в данной работе алгоритма.

4.2. Эксперименты на реальных строках

Искусственные тексты были рассмотрены в данной работе из-за существенно различия между размером сжатого текста и несжатого при подобном построении. Тем не менее, на реальных данных сильное сжатие



(a) LZ78



(b) LZW

Рис. 5: Эксперименты на искусственных текстах

не всегда достигается. В качестве примера данных, на которых в реальности имеет смысл задача приближенного поиска по образцу, были рассмотрены тексты на естественном языке.

В качестве исходных данных был взят набор текста с ресурса Project Gutenberg. Соответственно, для генерации нескольких различных естественных текстов фиксированной длины брались различные неперекрывающихся отрезки текста необходимой длины.

Дополнительно была реализована стандартная для многих методов NLP предобработка естественных текстов для использования на них сжатого алгоритма: токенизация с помощью библиотеки `nltk.tokenize`, лемматизация с помощью `nltk.stem` и удаление стоп-слов с помощью `nltk.corpus`.

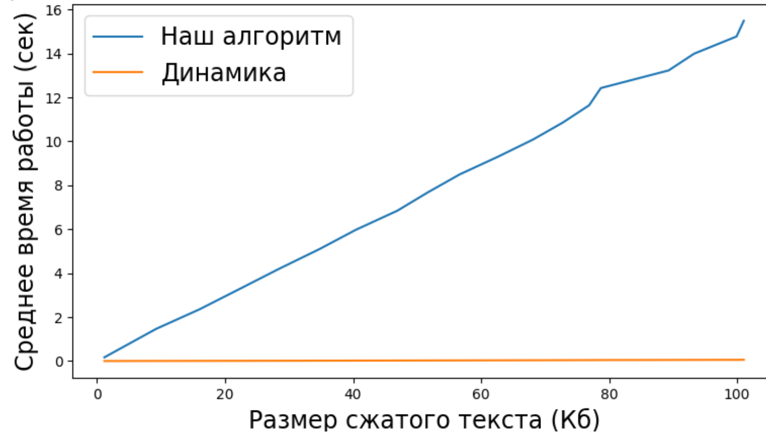
Поскольку такая предобработка удаляет все неалфавитные символы и оставляет только строчные буквы алфавита, для алгоритма LZW достаточен размер алфавита 26.

Тем не менее, даже с предобработкой, уменьшающей размер алфавита и соответственно увеличивающей вероятность повторений кусков текста, на практике сжатие достигается не более чем в константу раз. Соответственно реализованный алгоритм не получает существенный выигрыш за счет обработки сжатого текста и не становится более производительным, чем стандартное динамическое программирование.

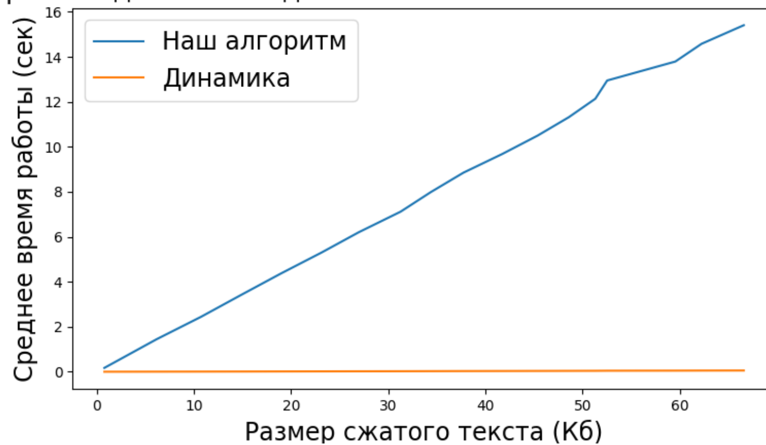
4.3. Эксперименты на UNIX-compress

Использование формата сжатых данных .z, то есть сжатие с помощью UNIX-утилиты compress, вместо реализованных самостоятельно форматов сжатых данных, позволяет дополнительно сравнивать работу реализованного алгоритма с существующими решениями для приближенного поиска по образцу.

В качестве решения для сравнения была выбрана UNIX-утилита agrep. Хотя она решает не общий случай задачи, а частный, где размер образца < 32, среди решений, реализующие общий случай задачи, были найдены только биологические, использующие специальный биоло-

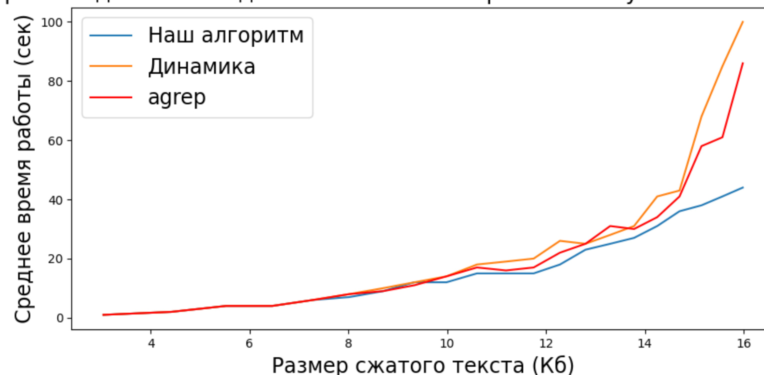


(a) LZ78

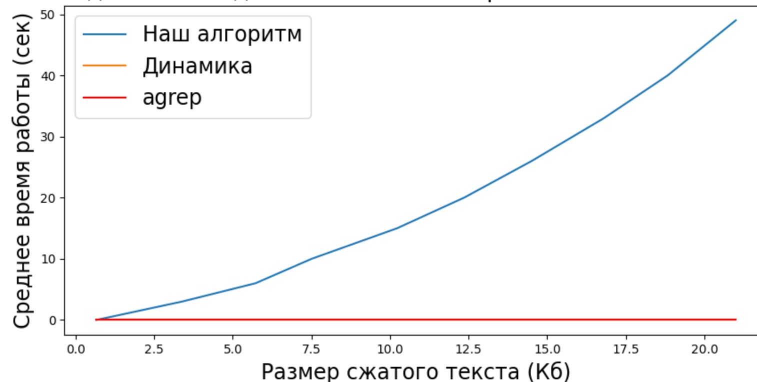


(b) LZW

Рис. 6: Эксперименты на естественных текстах



(a) Искусственные тексты



(b) Реальные тексты

Рис. 7: Эксперименты с UNIX-compress

гический формат входных данных. Таким образом, утилита agrep был взята как наиболее известная и каноническая среди точных решений приближенного поиска.

Запуск экспериментов с UNIX-compress несколько усложнен относительно экспериментов с LZ78 и LZW: для сжатия утилитой compress и приближенного поиска по образцу с помощью утилиты agrep требуется запускать алгоритм в Unix-подобной ОС. Дополнительно, из-за использования внешних средств для сжатия данных и приближенного поиска, нет возможности замерять время работы всех экспериментов с помощью внутренних средств языка программирования непосредственно в реализации. Соответственно, для измерения времени работы экспериментов и их запуска был написан bash-скрипт.

Результаты экспериментов с UNIX-compress представлены на рисунке 7. Для графика искусственных текстов использовалось меньшее

число запусков, для ускорения времени работы. Можно заметить, что на искусственных текстах с некоторого момента реализованный алгоритм становится более производительным, чем стандартное динамическое программирование и agrep, хотя это и происходит и менее выражено, чем в предыдущих случаях. Это может быть связано например с тем, что в используемых реализациях LZ78 и LZW размер алфавита на порядок меньше, чем в UNIX-compress, из-за специального формата входных данных для экспериментов, или же с более сложной структурой UNIX-compress и со сложностью запуска экспериментов для данного формата сжатия.

4.4. Выводы

Полученная реализация приближенного поиска по образцу для сжатых данных была протестирована как на искусственно сгенерированных сильносжимаемых текстах, так и на реальных текстах на естественном языке. В первом случае было продемонстрировано ускорение времени работы алгоритма по сравнению со стандартными методами решения задачи приближенного поиска, но во втором случае выигрыша не наблюдалось.

Такие результаты пока не дают основание считать предложенное решение применимым на практике как альтернатива стандартным методам, в случаях когда необходимо быстро получить точное значение решения задачи приближенного поиска. Для практического применения реализованного алгоритма необходимо либо улучшить константу в его реализации, чтобы он не был существенно медленнее стандартных решений в случаях, когда данные сжаты не сильно, либо рассмотреть другие входные данные — найти формат сжатия, сильнее сжимающий реальные данные или найти реальные данные, которые более существенно сжимаются рассмотренными алгоритмами.

Заключение

Главным результатом данной работы стала реализация нового подхода к решению задачи приближенного поиска по образцу. В отличие от существующих реализаций приближенного поиска по образцу, предложенный подход обрабатывает тексты и файлы, сжатые КСГ без их предварительной распаковки, что позволяет в некоторых случаях решать задачу приближенного поиска по образцу более эффективно, чем существующие решения. Также в рамках данной работы были достигнуты следующие результаты:

- Реализован алгоритм для приближенного поиска по образцу в данных, сжатых КСГ, без их предварительной распаковки.
- Реализованы декомпрессоры, переводящие данные, сжатые LZ78, LZW и UNIX-compress в формат КСГ, использующийся алгоритмом.
- Алгоритм был протестирован на специально сгенерированных искусственных строках, на которых достигается квадратичное сжатие, и на строках естественного языка. Для всех поддерживаемых алгоритмов сжатия на искусственных строках достигнуто заметное улучшение производительности начиная с какого-то размера входных данных, а на строках естественного языка улучшение не наблюдалось.

Дальнейшая работа возможна в следующих направлениях:

- Оптимизировать время работы реализованного алгоритма. Для этого возможно использовать как стандартные оптимизации, используемые в задаче приближенного поиска по образцу, так и оптимизации, использующие алгебраическую структуру реализованного в данной работе алгоритма, например использование SIMD-инструкций AVX-512 от Intel для параллельного причесывания липкой косы.

- Поддержать новые форматы сжатия в реализованном алгоритме и исследовать время работы алгоритма на них. В качестве примера потенциального нового формата сжатия можно привести сжатие с помощью run-length SLP [9].
- Разработать на основе реализованного алгоритма удобный инструмент для приближенного поиска по образцу в данных, сжатых различными методами.

Список литературы

[1] Abboud Amir, Backurs Arturs, Williams Virginia Vassilevska. Tight Hardness Results for LCS and Other Sequence Similarity Measures // 2015 IEEE 56th Annual Symposium on Foundations of Computer Science. — 2015.

[2] Adler Mark. Re: How can I read compressed .Z file automatically. — https://mathematica.stackexchange.com/questions/60531/how-can-i-read-compressed-z-file-automatically-by-mathematica/60879#60879.

[3] Amir Amihood, Benson Gary, Farach Martin. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files // Journal of Computer and System Sciences. — 1996. — Vol. 52, no. 2. — P. 299–307.

[4] Bringmann Karl, Kunnemann Marvin. Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping // 2015 IEEE 56th Annual Symposium on Foundations of Computer Science. — 2015.

[5] Bucak I O, Uslan V. An analysis of sequence alignment: Heuristic algorithms // 2010 Annual International Conference of the IEEE Engineering in Medicine and Biology. — 2010.

[6] Crochemore Maxime, Landau Gad M., Ziv-Ukelson Michal. A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices // SIAM Journal on Computing. — 2003. — Vol. 32, no. 6. — P. 1654–1673.

[7] Gawrychowski Paweł. Faster Algorithm for Computing the Edit Distance between SLP-Compressed Strings // String Processing and Information Retrieval Lecture Notes in Computer Science. — 2012. — P. 229–236.

[8] Gusfield Dan. Algorithms on Strings, Trees and Sequences. — 1997.

[9] Kempa Dominik, Prezza Nicola. At the Roots of Dictionary Compression: String Attractors. — 2017. — 10.

[10] LEVENSHTEIN V. Binary codes capable of correcting spurious insertions and deletion of ones // Problems of Information Transmission. — 1965. — Vol. 1, no. 1. — P. 8–17. — Access mode: `https://ci.nii.ac.jp/naid/10030538758/en/`.

[11] Laurikari Ville. TRE. — `https://github.com/laurikari/tre`.

[12] Lohrey Markus. Algorithmics on SLP-compressed strings: A survey // Groups - Complexity - Cryptology. — 2012. — Vol. 4, no. 2.

[13] Masek William J., Paterson Michael S. A faster algorithm computing string edit distances // Journal of Computer and System Sciences. — 1980. — Vol. 20, no. 1. — P. 18–31.

[14] NCBI. Basic Local Alignment Search Tool. — `https://blast.ncbi.nlm.nih.gov/Blast.cgi`.

[15] Navarro Gonzalo, Tarhio Jorma. LZgrep: a Boyer-Moore string matching tool for Ziv-Lempel compressed text // Software: Practice and Experience. — 2005. — Vol. 35, no. 12. — P. 1107–1130.

[16] Needleman Saul B., Wunsch Christian D. A general method applicable to the search for similarities in the amino acid sequence of two proteins // Journal of Molecular Biology. — 1970. — Vol. 48, no. 3. — P. 443–453.

[17] Pearson William. The FASTA package - protein and DNA sequence similarity searching and alignment programs. — `https://github.com/wrpearson/fasta36`.

[18] Python. difflib — Helpers for computing deltas. — `https://github.com/python/cpython/blob/3.9/Lib/difflib.py`.

[19] Sellers Peter H. The theory and computation of evolutionary distances: Pattern recognition // Journal of Algorithms. — 1980. — Vol. 1, no. 4. — P. 359–373.

[20] Tiskin Alexander. Semi-local longest common subsequences in subquadratic time // Journal of Discrete Algorithms. — 2008. — Vol. 6, no. 4. — P. 570–581.

[21] Tiskin Alexander. Fast Distance Multiplication of Unit-Monge Matrices // Algorithmica. — 2013. — Vol. 71, no. 4. — P. 859–888.

[22] Tiskin A. Bounded-Length Smith-Waterman Alignment // WABI. — 2019.

[23] Ukkonen Esko. Algorithms for approximate string matching // Information and Control. — 1985. — Vol. 64, no. 1-3. — P. 100–118.

[24] Wagner Robert A., Fischer Michael J. The String-to-String Correction Problem // Journal of the ACM. — 1974. — Vol. 21, no. 1. — P. 168–173.

[25] Welch. A Technique for High-Performance Data Compression // Computer. — 1984. — Vol. 17, no. 6. — P. 8–19.

[26] Wikinaut. AGREP - an approximate GREP. — `https://github.com/Wikinaut/agrep`.

[27] Window Subsequence Problems for Compressed Texts / Patrick Cégielski, Irène Guessarian, Yury Lifshits, Yuri Matiyasevich // Computer Science – Theory and Applications Lecture Notes in Computer Science. — 2006. — P. 127–136.

[28] Ziv J., Lempel A. Compression of individual sequences via variable-rate coding // IEEE Transactions on Information Theory. — 1978. — Vol. 24, no. 5. — P. 530–536.

[29] pevalme. Regular Expression Matching on Compressed Text. — `https://github.com/pevalme/zearch`.