Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (национальный
исследовательский университет)»
Физтех-школа радиотехники и компьютерных технологий
Кафедра теоретической и прикладной информатики

**Направление подготовки:** 03.03.01 Прикладные математика и физика
**Направленность (профиль) подготовки:** Радиотехника и компьютерные технологии

# СЖАТЫЕ СТРУКТУРЫ ДЫННЫХ В ЗАДАЧЕ ПОЛНОТЕКСТОВОГО ПОИСКА

(бакалаврская диссертация)

**Студент:**
Соколов Вадим Андреевич

*(подпись студента)*

**Научный руководитель:**
Неганов Алексей Михайлович

*(подпись научного руководителя)*

Москва 2021

# Аннотация

**Цели и задачи работы.**

Данная работа посвящена исследованию структур данных, использующих сжатые индексы (succinct index) для хранения текстовой информации.

Целью данной работы является проверка эффективности различных методов сжатия данных. Исследуется применимость сжатых индексов на практике при работе с данными определенного типа. Производится сравнение как с традиционными решениями (suffix array), так и с более современными (radix tree), использующими подход для индексирования, отличный от исследуемых структур данных.

**Полученные результаты.**

Удалось получить сравнительные характеристики работы исследуемых структур данных. Были измерены и проанализированы:

1. объем потребляемой памяти;

2. время, требуемое для поиска подстроки;

На языке Go реализована сжатая структура данных succinct suffix array. Приведены результаты потребления памяти для хранения сжатого индекса и скорости поиска подстроки в тексте.

# Содержание

# 1 Введение

Работа с текстовыми данными находит применение в широком спектре задач современной компьютерной индустрии. Существует ряд проблем, связанных с поиском информации в поисковых сервисах. Рост количества информации в Интернете приводит к дополнительным издержкам при хранении и поиске данных. В связи с этим существует необходимость исследования различных способов уменьшения потребляемой памяти без существенных затрат на поиск данных.

Одним из возможных решений такого рода задач является применение сжатых структур данных (succinct data structures). В зависимости от степени сжатия информации структуры данных различаются на имплицитные, сжатые и компактные. Сжатые структуры используют близкое к теоретически минимальному количеству информации для хранения данных. Кроме того, в отличие от архивов и других сжатых представлений, остается возможность эффективно выполнять операции поиска. Предположим, что для хранения некоторого количества данных требуется Z бит. Сжатые структуры данных занимают $Z + o(Z)$ бит. Например структура данных, занимающая $Z + \ln(Z)$ бит памяти, является сжатой.

Данные не всегда сжимаемы. Кроме того, не любые данные целесообразно сжимать с точки зрения эффективности их использования в несжатом виде. В этой работе предлагается рассмотреть сжатие индекса суффиксного массива, построенного для различных тектовых данных. При этом сам текст остается в несжатом виде.

Для того чтобы представить данные в сжатом виде, необходимо подготовить их в специальном промежуточном формате. В этой работе используется алгоритм Элиас-Фано, позволяющий сжимать возрастающие последовательности неотрицательных целых чисел. Исследование направлено на изучение потребления памяти для сжатого представления суффиксного массива. Реализованы функции поиска подстроки, и произведен анализ их эффективности.

# 2 Постановка задачи

Прежде всего, сжатое представление требует ряда манипуляций над исходным суффиксным массивом. В связи с этим появляется необходимость использования промежуточного $\psi$-массива. Он представляет собой реорганизованный суффиксный массив, в котором присутствует несколько наборов возрастающих последовательностей индексов.

Возрастающая последовательность целых неотрицательных чисел сжимаема, что было показано в работе ... Для сжатия такой последовательности применяется алгоритм Элиаса–Фано, позволяющий записать $\psi$-массив в виде битового вектора.

Особенностью представления Элиаса–Фано является возможность восстановления значения $\psi$-массива по индексу. Для битовых векторов определены операции *rank* и *select*, работающие за константное время.

Непосредственными задачами являются:

1. Построение суффиксного массива для данного текста (набора символов).
2. Построение $\psi$-массива по полученному суффиксному массиву.
3. Реализация алгоритма сжатия Элиаса-Фано.
4. Написание функции получения индекса в суффиксном массиве по $\psi$-представлению.
5. Реализация алгоритма распаковки индекса в $\psi$-массиве.
6. Написание функции поиска подстроки в исходном тексте.
7. Написание бенчмарков для сравнения сжатого суффиксного массива с традиционным суффиксным массивом и radix tree.

# 3 Обзор литературы

1. Common problems and common technologies [+] - Text search + - Self Index + - Problems with self indexation + - Full text search with substrings + 2. SA: overview, complexity, usage [+] 3. Radix: overview, complexity, usage [+] 4. Succinct data [+] 5. CSA [+] 6. PSI [+] 7. Regenerating SA from PSI [] 8. EF [] 9. My CSA [] 10.Complexity []

**Поиск по тексту**

В настоящее время Интернет каждый день пополняется большим количеством данных. Поэтому чрезвычайно важно организовать поиск нужной информации *эффективно*. При поиске по документам применяется индексирование. Традиционным для индексирования текста является *inverted index*. Это структура данных, в которой для каждого слова коллекции документов в соответствующем списке перечислены все документы в коллекции, в которых оно встретилось. При обработке многословного запроса берётся пересечение списков, соответствующих каждому из слов запроса.

**Проблемы традиционных подходов**

На практике встречаются случаи, в которых невозможно использовать традиционный поиск по словам. Модель поиска, в которой задачей является найти орфографически близкие слова (*fuzzy search*), использующаяся для корректирования правописания во многих текстовых редакторах, не позволяет решать проблему при помощи поиска по словам. То же самое касается других систем, в которых используется *pattern matching*. Еще одним примером текстов, в которых невозможно применять традиционный поиск, являются некоторые восточные языки. В них слова не делятся пробелами между собой, что делает трудным использование *inverted index*. Наконец, к "неудобным"можно отнести длинные тексты, составленные из алфавита с малым набором символов. Характерными примерами таких текстов являются ДНК и код белковой структуры. Этот текст к тому же тоже не делится на слова.

Во всем вышеприведенных случаях текст не делится на слова, поэтому для таких примеров необходимо использовать поиск по подстрокам. Существующие подходы (prefix tree, suffix array) занимают много места, поэтому не являются эффективными. Например, для prefix tree, хранящем в себе n слов со средним количеством символов в каждом слове C, требуется $O(n \cdot C)$ памяти. Рассмотрим подробнее suffix array, как один из самых востребованных способов индексации текстовой информации.

**Полнотекстовый поиск при помощи suffix array**

Suffix array – это структура данных, используемая в полнотекстовой индексации, позволяющая выполнять поиск подстроки в тексте разме-

ром n символов за $O(\log n)$. При этом для хранения n слов в suffix array необходимо $O(n \log n)$ памяти. Так для ASCII-текста размером $2^{32}$ требуется $2^{32} \cdot 8 = 32$ гигабит пространства памяти. Для такого текста suffix array должен содержать элементы размером 32 бита. Тогда размер suffix array достигает $2^{32} \cdot 32 = 128$ гигабит пространства памяти, что в 4 раза превышает размер исходного текста.

**Создание suffix array**

Suffix array представляет собой отсортированную в лексикографическом порядке последовательность суффиксов. Для лучшего понимания устройства suffix array, рассмотрим эту структуру данных на примере индексирования слова mississippi:

(0) mississippi
(1) ississippi
(2) ssissippi
(3) sissippi
(4) issippi
(5) ssippi
(6) sippi
(7) ippi
(8) ppi
(9) pi
(10) i

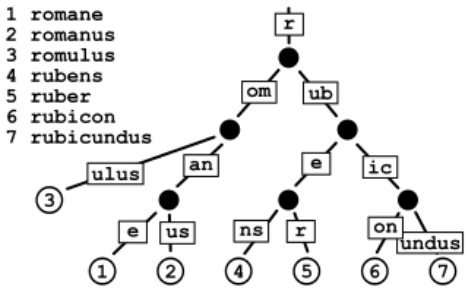
Suffix array, составленный из таких суффиксов, будет иметь вид:

10 7 4 1 0 9 8 6 3 5 2

Обычно конец документа обозначается специальным символом, не входящим в алфавит хранящегося в массиве текста. В качестве такого разделителя в этой работе был выбран символ доллара \$. Современные подходы по построению suffix array позволяют конструировать структуру данных за $O(n)$.

**Radix tree**

Radix tree представляет собой сжатое prefix tree. В свою очередь, prefix tree является структурой данных, которая предоставляет интерфейс ассоциативного массива и позволяет хранить значения в виде key-value пар. В этом исследовании в качестве key выбраны строки, причем в отличие от обычный деревьев, на ребрах radix tree могут храниться как один элемент (символ), так и последовательность элементов (строка).




Radix tree широко применяется в ядре Linux для связи указателя с длинным целочисленным ключом. Эта структура дан-

ных эффективна с точки зрения скорости поиска и хренения информации. В то же время, Radix tree применяется в IP-адресации, так как очень удобна для хранения иерархической структуры IP-адресов.

**Поиск и хранение данных в radix tree**

Рассмотрим подробнее характеристики radix tree. Пусть требуется хранить ключи размером k при хранении n элементов в дереве. Тогда добавление элемента, поиск и удаление элемента занимают $O(k)$ операций.

Для многих запросов поиска по словарю radix tree может быть быстрее и эффективнее, чем хеш-таблицы. Несмотря на то, что обычно сложность поиска по хэш таблице принимают за $O(1)$, при этом игнорируется необходимость сначала хешировать входные данные. На это обычно требуется $O(k)$ операций, где k – длина входной строки. Radix tree выполняет поиск за $O(k)$, без необходимости сначала хешировать и имеют гораздо лучшую локальность кеша. Они также сохраняют порядок, позволяя выполнять упорядоченное сканирование, получать минимальные / максимальные значения, сканировать по общему префиксу и т.д.

Благодаря всем этим преимуществам, radix tree широко распространено в базе данных Redis, программных продуктах компании HashiCorp, таких как Terraform, Consul, Vault, Nomad и т.д., что представляет дополнительный интерес для его изучения на текстовых данных и сравнения с suffix array.

**Сжатое представление данных**

Одной из главных задач этой работы является разработка сжатого хранения индекса для suffix array. Поэтому важно определить, какие данные можно считать сжатыми.

Существуют различные степени сжатости информации. Предположим, что для хранения некоторого количества данных требуется Z бит. Сжатые (succinct) структуры данных занимают $Z + o(Z)$ бит, implicit – $Z + o(1)$ бит и compact – $O(Z)$) бит. В этой работе предлагается сжать suffix array, представив его в succinct виде. Таким образом, размер массива немногих больше превосходит размер исходного текста.

**Ψ-массив**

Важным место в теории сжатого представления индекса занимает промежуточный $\psi$-массив. Характерной особенностью всех алгоритмов сжатия succinct suffix array (CSA) является то, что сжимается не непосредственно индекс, а его представление в виде $\psi$-массива. Вспомогательный массив можно получить из suffix array путем следующих манипуляций с индексами.

Рассмотрим построение $\psi$-массива на примере знакомой строки **mississippi$**:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Text**: | m | i | s | s | i | s | s | i | p | p | i | $ |
| **Offset**: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **Suffix Array**: | 11 | 10 | 7 | 4 | 1 | 0 | 9 | 8 | 6 | 3 | 5 | 2 |
| **$\Psi$ -array**: | $ | 0 | 7 | 10 | 11 | 4 | 1 | 6 | 2 | 3 | 8 | 9 |

Будем называть потомком (successor) самый большой суффикс подстроки, кроме исходной подстроки. Т.е. для подстроки issippi$ потомком является ssippi$. $\Psi$-массив указывает на потомок для каждого выбранного суффикса. Например, рассмотрим $SA[7] = 8$. Его потомком является позиция в suffix array, которая хранит $8+1 = 9$. Таким образом, $\psi[7] = 6$. Главное соотношение между $\Psi$-массивом и suffix array:

$$SA[i] = SA[\psi[i]] - 1$$

Необходимо отметить, что $\Psi[0] = \$$, т.к. для первого элемента в suffix array нет потомка. При этом $SA[5]$ ссылается на 0-й индекс, то есть на всю строку целиком. Иными словами, элемент suffix array ссылается на начало текста, т.е. никакой другой суффикс не может иметь его в качестве потомка. Соответственно в $\psi$-массиве нет элемента, равного 5.

**Восстановление suffix array из $\psi$-массива**

Существует возможность восстановить исходный suffix array по сгенерированному $\psi$-массиву фактически с помощью операций, обратных к тем, которые были представлены в предыдущем параграфе. Коснемся подробнее особенностей работы этого алгоритма.

В первую очередь необходимо найти индекс элемента suffix array, который не встречается в $\psi$-массиве. В нашем примере он равен 5. Из приведенных выше рассуждений следует, что $SA[5] = 0$. Таким образом был декодирован один элемент исходного suffix array. Из формулы ... для $i = 5$ следует:

$$SA[5] = SA[\psi[5]] - 1$$

$$0 = SA[4] - 1$$

$$SA[4] = 1$$

Еще один элемент suffix array восстановлен! Таким образом можно восстановить всю оставшуюся последовательность. Время на поиск первого элемента составляет $O(n)$ операций. Существуют некоторые улушения скорости поиска нужного элемента в suffix array, требующие дополнительных структур данных, хранящих значение suffix array на каждом $\log n$ шаге. Такой способ выходит за рамки этой работы, поскольку главной задачей является максимальное сжатие индекса.

**Сжатие $\psi$-массива**

Рассмотрим подробнее $\psi$-массив:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Text**: | m | i | s | s | i | s | s | i | p | p | i | $ |
| **Offset**: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **Suffix Array**: | 11 | 10 | 7 | 4 | 1 | 0 | 9 | 8 | 6 | 3 | 5 | 2 |
| **$\Psi$ -array**: | $ | 0 | 7 | 10 | 11 | 4 | 1 | 6 | 2 | 3 | 8 | 9 |

Отсортированный набор суффиксов имеет вид:

(0) $
(1) i$
(2) ippi$
(3) issippi$
(4) ississippi$
(5) mississippi$
(6) pi$
(7) ppi$
(8) sippi$
(9) sissippi$
(10) ssippi$
(11) ssissippi$

Стоит обратить внимание на его структуру: он состоит из набора возрастающих последовательностей индексов. Рассмотрим возрастающую последовательность 2, 3, 8, 9. $SA[2] = 7$, $T[7] = i$. $SA[3] = 4$, $T[4] = i$. Примечательно, что $SA[6] = 9$, $T[9] = p$. То есть чередование индексов с возрастания на убывание соответствует смене символа.

**Кодировка Elias–Fano**

# 4 Исследование

## 4.1 Экспериментальная платформа

## 4.2 Suffix array

## 4.3 Radix tree

## 4.4 Compressed suffix array

# 5 Выводы

# 6 Заключение