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 ψ-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 ψ-массив в виде битового вектора.

Особенностью представления Элиаса–Фано является возможность восстановления значения ψ-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 ψ-массива по полученному суффиксному массиву.
3. Реализация алгоритма сжатия Элиаса-Фано.
4. Написание функции получения индекса в суффиксном массиве по ψ-представлению.
5. Реализация алгоритма распаковки индекса в ψ-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

Lemire et al. (2016)

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 размером 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).

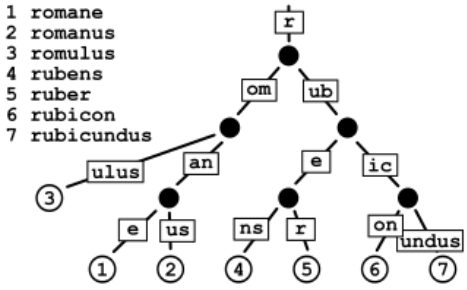


Рис. 1: Пример Radix tree

Radix tree широко применяется в ядре Linux для связи указателя с длинным целочисленным ключом. Эта структура данных 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 ψ-массив. Характерной особенностью всех алгоритмов сжатия succinct suffix array (CSA) является то, что сжимается не непосредственно индекс, а его представление в виде ψ-массива. Вспомогательный массив можно получить из suffix array путем следующих манипуляций с индексами.

Рассмотрим построение ψ-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
Ψ -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$. Ψ-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 Ψ-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 ψ-массиве нет элемента, равного 5.

Восстановление suffix array из ψ-массива

Существует возможность восстановить исходный suffix array по сгенерированному ψ-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 ψ-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 ψ-массива

Рассмотрим подробнее ψ-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
Ψ -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 с i на p.

Лемма 1. *Два последовательных элемента ψ-массива являются возрастающими, если соответствующие суффиксы, на которые они ссылаются, начинаются с одного и того же символа.*

Доказательство. По построению, suffix array представляет собой набор индексов в исходном тексте, отсортированный в лексикографическом порядке. Это означает, что $T[SA[i]] < T[SA[i+1]] \; \forall i \in [0, n)$. Эти два суффикса имеют общий первый символ. Поэтому они отличаются в следующих элементах, например, во втором: $T[SA[i]+1] < T[SA[i+1]+1]$.

Ψ-массив ссылается на потомка для данного суффикса: $T[SA[\psi[i]]] < T[SA[i]+1]$. Т.к. $T[SA[i]]$ и $T[SA[i+1]]$ имеют одинаковый первый символ, их потомки упорядочены. Если $T[SA[\psi[i]]] < T[SA[\psi[i+1]]]$, тогда из-за упорядоченности индекс в suffix array для первого должен находиться перед индексом для второго. Следовательно индексы (значения ψ-массива) должны быть упорядочены, если суффиксы имеют общий первый символ. □

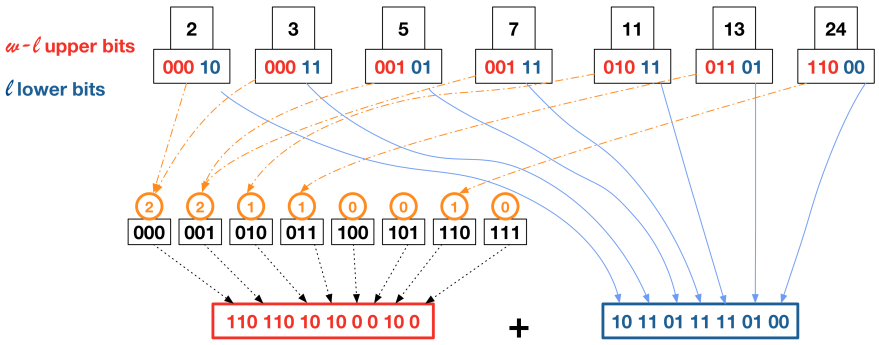


Рис. 2: Пример построения кода Elias-Fano

Возрастающие последовательности неотрицательных целых чисел сжимаемы. Самый простой путь для хранения таких последовательностей – битмап. Однако этот способ не позволяет иметь произвольный доступ к элементам. Рассмотрим более совершенный алгоритм сжатого представления данных.

Кодировка Elias–Fano

Сжатие при помощи метода Elias–Fano позволяет представлять монотонно возрастающие последовательности целых неотрицательных чисел в виде битовых векторов. Этот способ делает возможным хранение неубывающей последовательности n целых чисел размером $[0, m)$, занимая $2n + n\lceil \log m/n \rceil$ бит, предоставляя доступ к i-му элементу за $O(1)$. Сравнив размеры структуры данных с минимально возможным занимаемым местом в памяти с точки зрения теории информации, Elias–Fano кодировка является сжатым (succinct) индексом.

В начале каждое число из последовательности кодируется $\log m$ битами данных. Двоичное представление элементов разбито на две части: верхнюю часть, содержащую в себе первые $\log n$ бит, и нижнюю, с оставшимися $\log m - \log n = \log m/n$ бит. Объединение нижних битов занимает $n \log m$ бит. Верхние биты представляют собой набор из $n + m/2^{\log m/n}$ бит. Начиная с пустого битового вектора, мы добавляем в эту часть 0 в качестве стоп-бита для каждого возможного значения, представляемого битами старшей части. Мы добавляем 1 для каждого фактически присутствующего значения, выставляя его перед соответствующим стоп-битом.

В качестве примера рассмотрим отсортированную последовательность 2,3,5,7,11,13,24. На рисунке 2 показана схема кодирования. Наибольшим числом в наборе (universe) является 24. Поэтому для представления каждого элемента необходимо выделить 5 бит на элемент. Затем нужно раз-

бить бинарное представление на две части: верхнюю и нижнюю. Исходя из ранее оговоренных условий, выберем 3 бита под верхнюю часть и 2 бита под нижнюю. Всего в последовательности присуствует 7 элементов. Рассмотрим число 2. Для него имеем $2 = 0b00010$ и соответственно 000 в качестве верхней части и 10 в качестве нижней. Повторяя этот процесс, получим набор нижних частей для каждого элемента. Затем объединим полученные чати вместе. Для верхних частей имеется 2^3 вариантов наборов значений. С каждым таким набором асссоциируется счетчик, который увеличивается на единицу, если число из последовательности имеет соответствующую верхнюю часть. Так для 2 инкрементируем набор 000. Число $3 = 0b00011$ с верхней частью 000 идет в тот же набор. Для числа $5 = 0b00101$ с верхней частью 001 инкрементируем набор 001 и т.д. Наконец, кодируем счетчики унарно, добавляя столько единиц в представление счетчика, сколько имеет значение каждого счетчика, за которым следует 0 бит. Окончательное кодирование Elias-Fano получается объединением полученных верхней и нижней частей.

Восстановление данных из кода Elias–Fano

Для восстановления исходной монотонной нецбывающей последовательности чисел, необходимо разработать операцию доступа к i-му элементу последовательности, $i \in [0, n)$. Для доступа к нижней части достаточно просто получить соответствующие биты, т.к. длина вектора для хранения элемента известна. Для получения старшей части, необходимо осуществить операцию *select*. Операция *select(i)* позволяет получить позицию i-го бита, выставленного в 1 в битовом векторе. Эта операция может быть выполнена за $O(1)$, что позволяет получать произвольный доступ к элементу последовательности без полного раскодирования всей последовательности целиком.

4 Исследование

Исследование состоит из нескольких главных частей. В первую очередь реализуется изучение работы традиционной структуры данных suffix array с точки зрения потребляемой памяти, времени построения индекса и поиска подстроки. Затем исследуется более современное radix tree, в котором обозреваются аналогичные аспекты функционирования. Одной из важнейших частей работы является разработка сжатого суффиксного массива (CSA). При этом выполняется не только тестирование CSA на бенчмарках, подобных другим обозреваемым структурам данных, но и исследование степени сжатия индекса по сравнению с suffix array. Важно подчеркнуть, что особый интерес представляет сравнение работы алгоритмов на текстах разного содержания. Коснемся подробнее ключевых характеристик машины, на которой проводились эксперименты.

4.1 Экспериментальная платформа

Исследование, включающее в себя запуск бенчмарков, велось на машине с характеристиками, указанными в таблице 1.

CPU	Intel Core i5-9600K 3.70 GHz
RAM	16GB
System Type	64-bit
Operating System	Windows 10
L1 Cache	512 B
L2 Cache	1 KB
L1 Cache	10 KB

Таблица 1: Характеристики компьютера

4.2 Suffix array

Для анализа работы структуры данных взят suffix array из стандартной библиотеки языка Go. Он принимает на вход текст и составляет из него отсортированный индекс, с помощью которого можно осуществлять операции поиска подстроки. Создание индекса занимает $O(n)$ операций, где n – размер исходного текста. Поиск подстроки занимает $O(\log n \cdot |s|)$, где $|s|$ – это длина искомой подстроки. Рассмотрим подробнее бенчмарки. Исходный текст загружается из файла, затем выбирается подстрока, ограниченная позициями $[leftPos : rigtPos]$.

```
1  func BenchmarkLookup(b *testing.B, testStr []byte) {
2      sa := suffixarray.New(testStr)
3      b.ResetTimer()
4      for i := 0; i < b.N; i++ {
5          offset := sa.Lookup(testStr[leftPos: rightPos], -1)
6          if len(offset) < 1 || offset[0] != leftPos {
7              b.Fatalf("mis-match: %v", offset)
8          }
9      }
10 }
```

Листинг 1: Suffix array example

На листинге 1 показан упрощенный код бенчмарка поиска подстроки для suffix array (маловажные для общего понимания детали опущены). Код протестирован для 5 текстов разного содержания. Размер подстроки для поиска данных является одинаковым для каждого измерения.

Т.к. размер suffix array не зависит от размера алфавита, результаты для разных текстов отличаются в пределах погрешности измерения, поэтому для наглядности достаточно привести пример для одного типа текста. В таблице 2 показаны результаты измерений построения индекса для Amazon Text Corpora.

Размер текста, KB	Память, KB	Время, s
9766	58741	1.832
977	6020	0.337
879	5457	0.333
782	4838	0.323
684	4252	0.314
586	3669	0.305
489	3589	0.297
391	2925	0.285
293	2244	0.275
196	1563	0.274
98	881	0.260
49	547	0.258
10	246	0.252
5	212	0.251
2	194	0.256

Таблица 2: Построение suffix array

Размер текста, KB	Память, KB	Время, ms
977	6020	12.251
879	5457	11.857
782	4838	12.331
684	4252	12.969
586	3669	12.777
489	3589	12.680
391	2925	11.795
293	2244	11.555
196	1563	11.752
98	881	11.657

Таблица 3: Поиск подстроки в suffix array

4.3 Radix tree

Реализация radix tree на языке Go взята из следующего источника: radix. Во многом выбор обусловлен удобством встраивания бенчмарка в существующую систему тестирования на языке Go. При этом с алгоритмической точки зрения эта реализация подтверждает теоретическое описание сложности поиска подстроки за $O(1)$.

```
func BenchmarkConstruct(b *testing.B, testStr string) {
  var substringArray []string = createSubstrings(testStr)
  b.ResetTimer()
  r := radix.New()
  for i := 0; i < b.n; i++ {
    fillRadixTree(b.N, r, substringArray)
  }
}
```

Листинг 2: Radix tree example

На приведенном листинге 2 в кратком виде показана функция построения radix tree. В начале генерируется набор подстрок из текста, прочитанного из файла. В функции $fillRadixTree$ происходит добавление подстрок в дерево. Сравнительные характеристики построения radix tree для Amazon Text Corpora приведены в таблице 4.

Размер текста, KB	Память, MB	Время, s
196	20248	276.419
148	11501	128.494
98	5192	57.319
79	3353	36.786
49	1324	14.632
10	52	0.845
5	13	0.414
2	2	0.284

Таблица 4: Построение radix tree

Ниже в листинге 3 приведен упрощенный пример функции поиска подстроки в дереве. Построенная структура показывает очень хороший результат поиска подстроки за константное время. Его можно видеть в таблице 5.

```
func getSubstring(r *radix.Tree,
                  subString string, testStr string) {
  var out interface{}
  var pos interface{}
  // there is only one possible match for a string
  fn := func(s string, v interface{}) bool {
    out, pos = s, v
    return false
  }
  r.WalkPrefix(subString, fn)
  return out.(string), pos
}
```

Листинг 3: Radix tree lookup

Размер текста, KB	Память, MB	Время, s
150	11529	256
120	7449	245
100	5210	243
90	4237	244
80	3370	243
70	2595	237
60	1918	243
50	1334	242
20	212	220

Таблица 5: Поиск подстроки в radix tree

4.4 Compressed suffix array

Перейдем к описанию реализации CSA. Как было упомянуто в обзоре литературы, для начала необходимо построить индекс таким же способом, как это происходит в suffix array. Для этого требуется $O(n)$ операций.

Структура CSA

```
type Csa struct {
  text             string
  suffixOffsets    []int
  psi              []uint64
  length           int
}
```

Листинг 4: CSA structure

На приведенном выше листинге 4 обозначена упрощенная структура CSA. Она состоит из исходного текста (сжимается только индекс, текст остается в прежнем виде), индекса, ψ-массива и длины текста. Индекс строится с помощью функции, в которой за основу алгоритма взят алгоритм построения suffix array из библиотеки языка Go.

Построение ψ-массива

Для построения ψ-массива нужно принять $\psi[0] = \$$. Затем произвести итеративный обход suffix array и найти индекс, соответствующее значение которого в suffix array совпадает с текущим, увеличенным на единицу. В листинге 5 показан псевдокод алгоритма.

```
func ConstructPsi() {
  for i < len {
    if sa[j] = sa[i] + 1 {
      psi[i] = j
    }
  }
}
```

Листинг 5: Построение CSA

Вспомогательные структуры данных

Для хранения данных используется битмап (bitmap) из пакета Roaring.bitmap. Это быстрая и эффективная реализация битмапа. Также Roaring используется во многих продуктах, таких как Apache Druid, LinkedIn Pinot, Google Procella и т.д.

Полученный ψ-массив представляет собой набор монотонно неубывающих последовательностей чисел. Алгоритм Elias-Fano позволяет преобразовывать каждую такую последовательность в битвектор в отдельности. Количество этих последовательностей совпадает с размером алфавита, используемого в тексте. Возникает вопрос, каким образом можно организовать хранение таких битвекторов.

Одним из решений является использование двух дополнительных массивов: первый для хранения оступа в ψ-массиве, второй для хранения символа, соответствующего возрастающей последовательности. В этой работе кодировка текста представляет собой ASCII-код или код меньшего размера. Таким образом, размер алфавита ограничен 128 символами. Следовательно, размер дополнительных массивов не превышает $2 \cdot m$, где m – размер алфавита. Массив с оступами используется для быстрого индексирования по массиву битмапов.

Заполнение дополнительных структур данных и расчет отступов происходит при сжатии каждой отдельной монотонной неубывающей последовательности индексов. Для отделения таких последовательностей используется описанное ранее свойство из Леммы 1. Нарушение возрастания ψ-массива соответсвует смене символа. Таким образом можно идексировать битмапы и заполнить вспомогательный массив с символами используемого алфавита.

Elias-Fano

Сжатие при помощи алгоритма Elias-Fano осуществляется для каждой отдельной последовательности. При этом происходит предварительное построение верхней и нижней частей (младших и старших биты) битового представления чисел из этой последовательности. Рассчитывается отступ для дальнейшего быстрого доступа к младшим битам. В процессе сжатия числа записываются в битмап, в котором можно индексироваться за константное время.

Для того чтобы получить доступ к элементу последовательности (части ψ-массива), необходимо использовать функцию $select(i)$, реализованную в битмапе, работающую за $O(1)$. Для проверки корректности работы алгоритма реализована функция получения всего ψ-массива при помощи вызова $select(i)$.

Восстановление suffix array

Для получения доступа к элементу suffix array $sa[i]$ нет необходимости полностью декодировать ψ-массив. Для этого требуется осуществить проход по ψ-массиву, пока не будет достигнут последний элемент. Сосчитав количество шагов до последнего элемента $h(i)$, найдем индекс искомого значения путем несложного вычисления: $sa[i] = n - h(i)$. Такой алгоритм требует $O(n)$ операций.

Окончательная структура CSA

После добавления вспомогательных данных структура CSA принимает вид, показанный на листинге 6

```
type Csa struct {
  text      string
  bv        []*CompressedText
  seqOffset []int
  seqChar   []byte
  length    int
  alphLen   int
}
```

Листинг 6: CSA structure

Важно подчеркнуть, что теперь нет необходимости хранить индексы и вспомогательный ψ-массив. Вместо этого данные хранятся в сжатом виде в последовательности битвекторов. Кроме того, исходный текст все так же остается в первоначальном виде.

Таким образом, для хранения сжатого индекса требуется $n \cdot \log |\sigma| + o(n)$, где $|\sigma|$ – размер алфавита. При двоичном поиске по suffix array необходимо произвести $O(\log n)$ операций. Для получения индекса в suffix array из ψ-массива требуется осуществить $O(n)$ операций. Суммарно для поиска элемента требуется $O(n \cdot \log n)$ операций.

Тестирование CSA

Рассмотрим эффективность работы CSA на примере пяти текстов различного содержания. Для начала оценим время построения массива и требуемое количество памяти для его хранения. В таблице 6 указаны данные для Amazon Text Corpora.

Amazon		
Размер текста, KB	Память, MB	Время, s
879	2281	392.040
782	2036	299.721
684	1816	225.818
586	1573	165.427
489	1336	114.660
391	1082	73.818
293	817	41.825
196	561	18.803
98	309	4.981
49	163	1.431
10	45	0.313
5	26	0.269
2	17	0.256

Таблица 6: Построение CSA

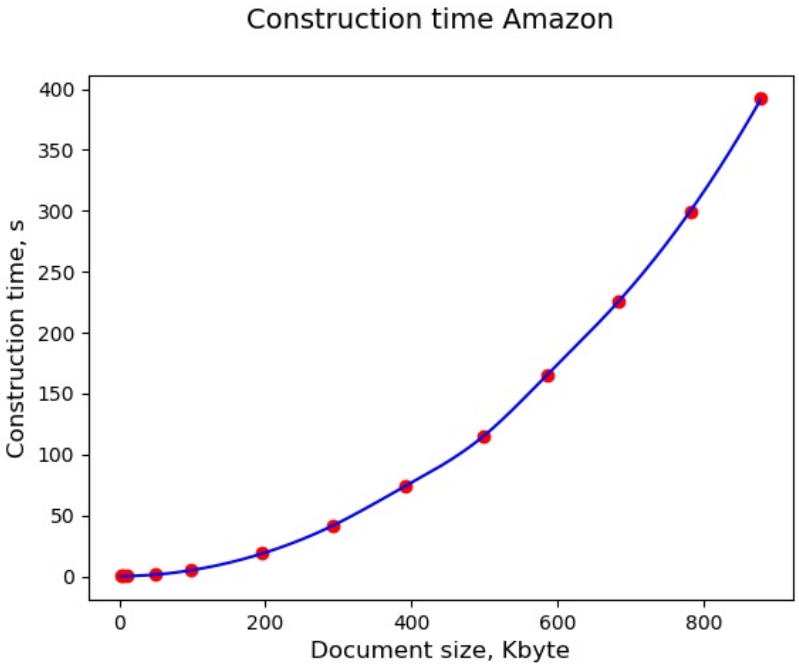
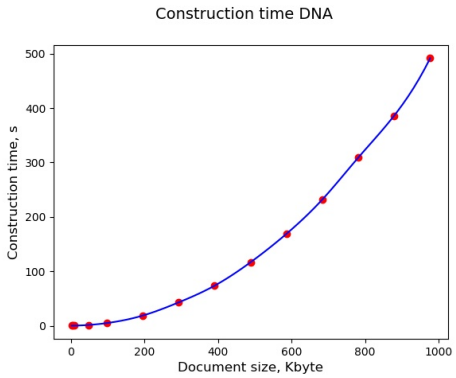
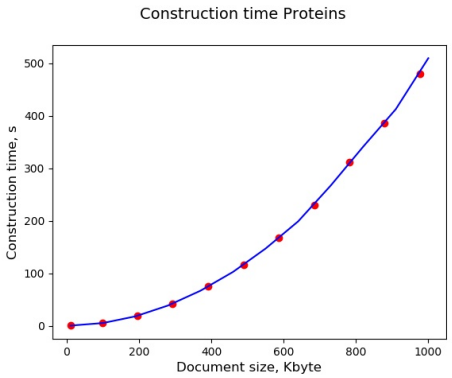


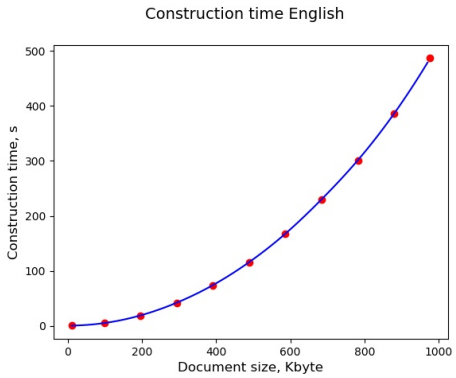
Рис. 3: Построение CSA



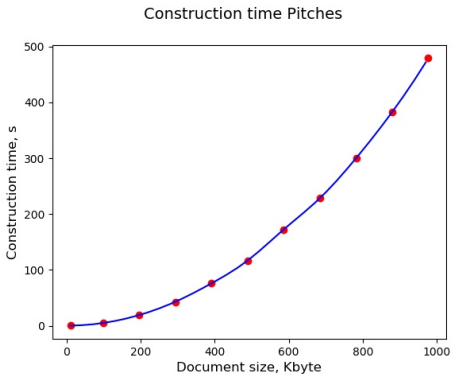
(a) DNA



(b) Proteins



(c) English



(d) Pitches

Рис. 4: Построение CSA

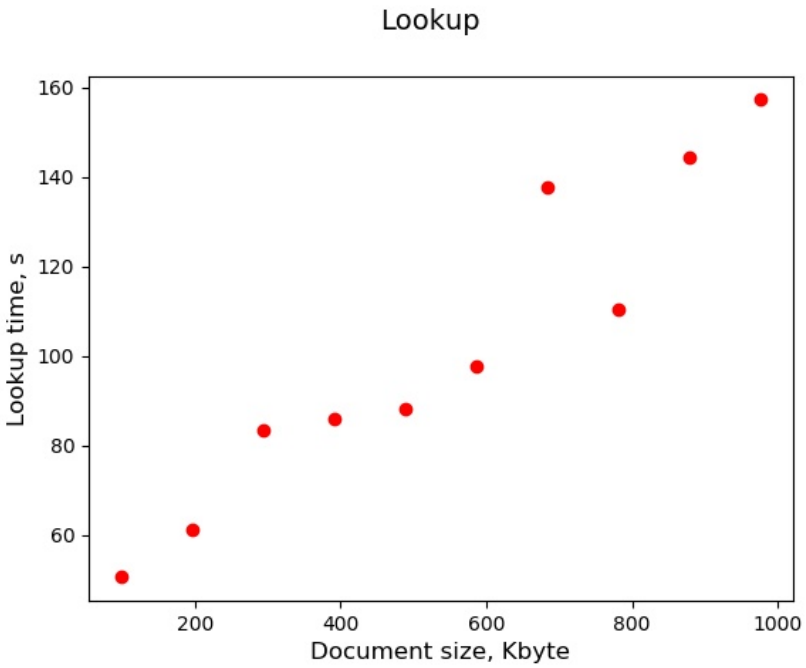


Рис. 5: Поиск подстроки в CSA

Рассмотрим результаты зависимость скорости поиска подстроки от размера текста для CSA, suffix array и radix tree, построенных для Amazon Text Corpora. На рисунке 6 можно видеть, что скорость поиска подстроки фиксированного размера не зависит от размера текста для suffix array и radix tree. Для CSA время поиска увеличивается при увеличении размера исходного текста. Radix tree, как ожидалось, показывает лучший результат.

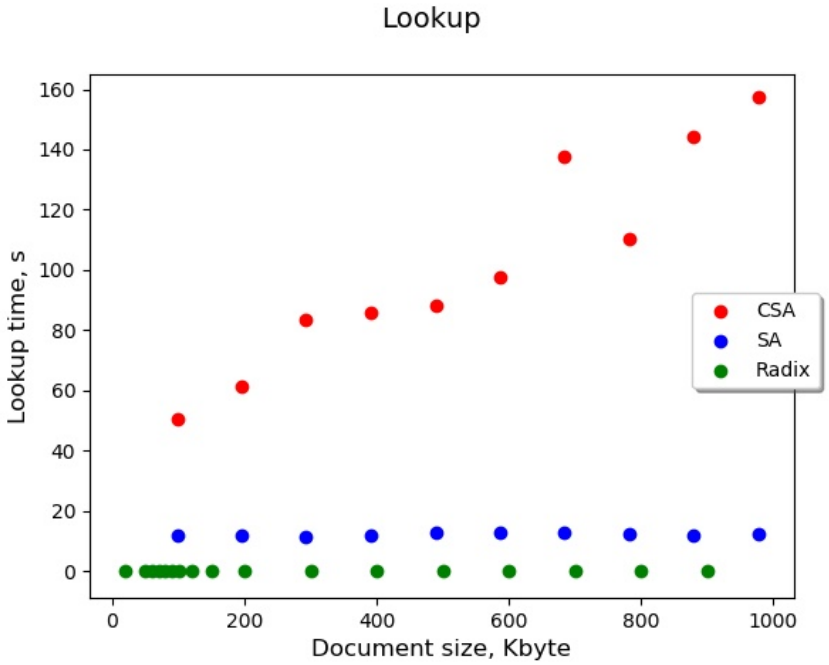


Рис. 6: Поиск подстроки в CSA

4.5 Сравнительные измерения

На рисунке 7 представлены сравнительные характеристики поиска подстроки для различных текстовых данных.

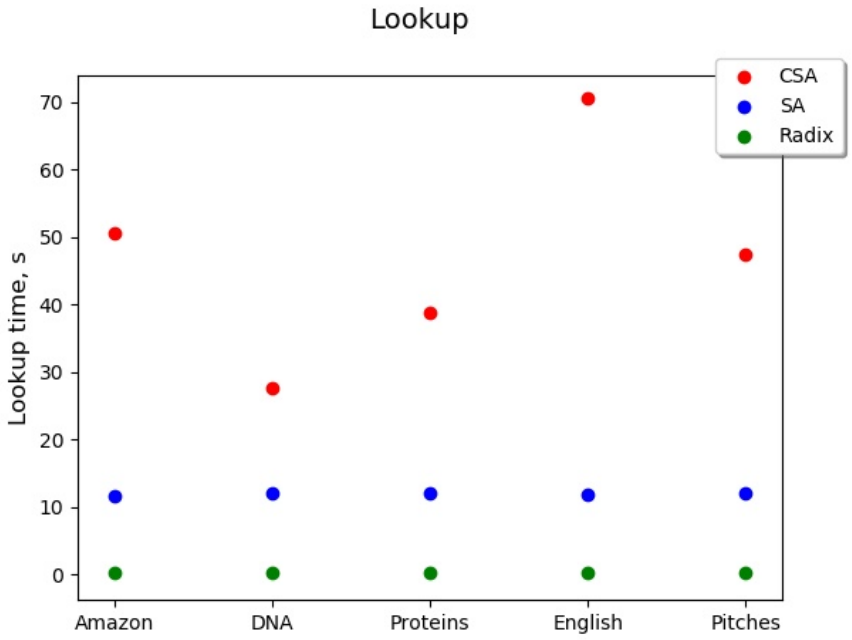


Рис. 7: Поиск подстроки в CSA

Рассмотрим относительное сжатие CSA по сравнению с suffix array. Расчитывается среднее отношение памяти, затраченной для работы CSA, к памяти, затраченной suffix array в зависимости от длины исходного текста. На рисунке 8 можно заметить уменьшение отношения, т.е. увеличение коэффициента сжатия. Примечательно, что для малых размеров исходного текста CSA занимает больше места, чем suffix array. Это объясняется наличием дополнительных структур данных, описанных на предыдущих страницах. При построении CSA они занимают место, сопоставимое по порядку с размером индекса в сжатом виде. Для больших текстов CSA становится более эффективным по сравнению с классическим suffix array.

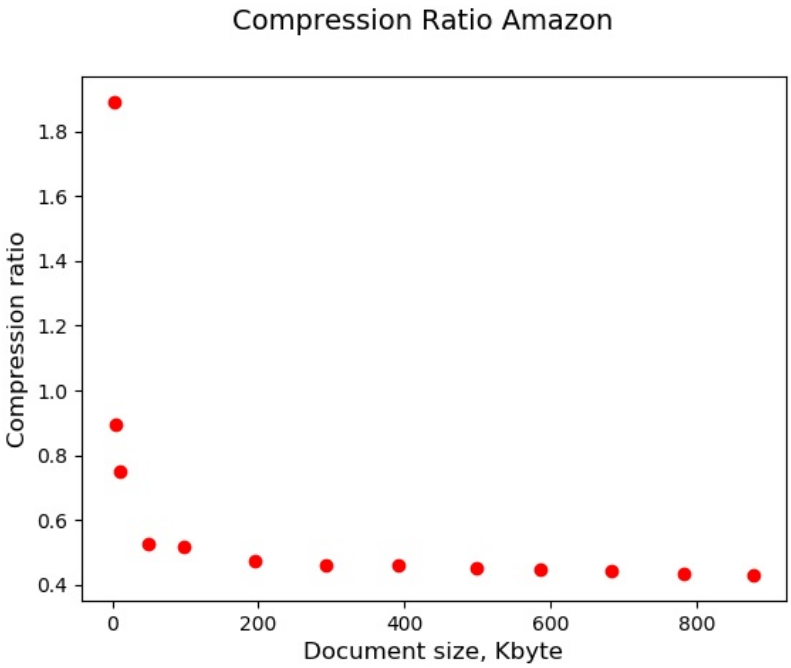


Рис. 8: Сжатие CSA по сравнению с SA

Относительное сжатие CSA/SA для различных текстов показано на рисунке 9. Можно явным образом заметить зависимость размера сжатого индекса от мощности алфавита, чего нельзя сказать о suffix array и radix tree.

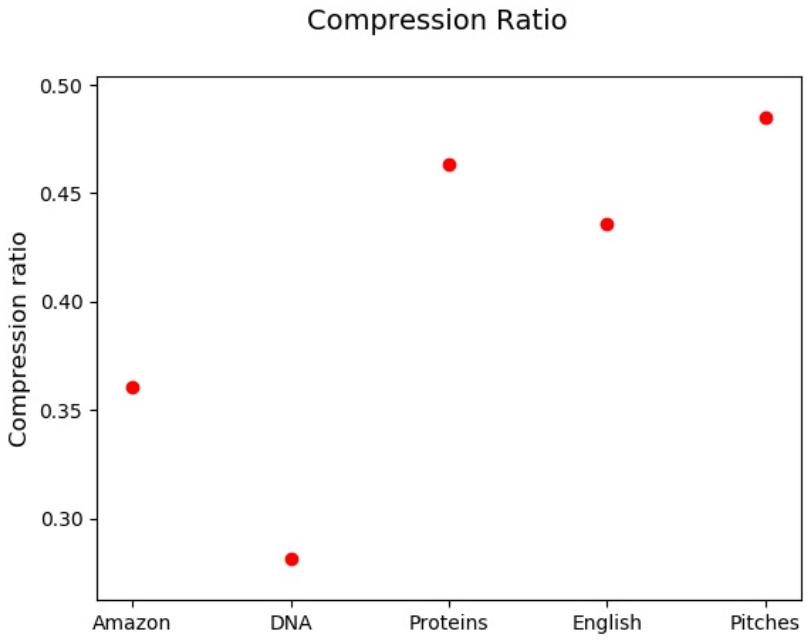


Рис. 9: Сжатие CSA по сравнению с SA

5 Выводы

Рассмотренная структура данных, сжатый суффиксный массив (CSA), показала свою эффективность с точки зрения памяти, занимаемой для хранения индекса. Относительно традиционного suffix array, как было показано на графиках сравнительных характеристик, при достаточно большом размере исходного текста CSA позволяет использовать меньше памяти, чем suffix array, сохраняя при этом возможность осуществлять поиск подстроки. Как было замечено при обсуждении зависимости сжатия от длины текста, при малых значени размера входного текста suffix array является более эффективным с точки зрения потребляемой памяти, что вызвано дополнительными затратами на обслуживание вспомогательных структур данных, использующихся для построения сжатого индекса. В то же время, как алгоритм построения индекса, так и поиска подстроки в CSA работают медленее, чем в исходном. Тем не менее, существуют методы улучшения скорости работы этих алгоритмов, позволяющие CSA достигнуть результатов, не уступающих suffix array.

Результаты, полученные при сравнении radix tree с остальными исследуемыми структурами данных, подтверждают эффективность поиска подстроки в radix tree за константное время. Однако для индексации по тексту необходимо заполнить дерево подстроками исходного текста, занимающими значительно большие объемы памяти по сравнению с suffix array и CSA. Аналогично, скорость заполнения дерева при построении не позволяет считать radix tree эффективным при работе с подстроками исходного текста.

6 Заключение

На языке Go была разработана структура данных сжатый суффиксный массив (CSA), имеющая функционал традиционного suffix array, позволяющего решать задачи поиска по подстрокам. Реализован алгоритм Elias-Fano, позволяющий представлять индекс в сжатом виде. Построены сравнительные характеристики работы по построению и поиску подстроки для CSA и suffix array для текстов различного содержания. Изучена зависимость потребляемой памяти структурами данных от размеров исходного текста. Произведено сравнение radix tree с этими структурами по времени построения и поиска подстроки, а также по затраченной памяти.

С точки зрения потребляемой памяти CSA показал свою эффективность по сранению с suffix array и radix tree, но разработанный алгоритм построения индекса и поиска подстроки является менее быстрым, чем аналогичные алгоритмы в традиционном suffix array. Показана эффективность поиска подстроки при помощи radix tree, подчеркнута сложность его использования для текстовых данных из-за больших затрат памяти при построении дерева.

Дальнейшие исследования в этой области могут опираться на разработку сжатых представлений существующих классических структур данных. Применение методов сжатия, таких как рассмотренный в этой работе алгоритм Elias-Fano, может позволить производить сжатое представление данных различного типа, предварительно преобразованных к нужному формату. Сжатые структуры данных могут быть задействованы в работе алгоритмов, использующих хранение индекса во внешней памяти, что может стать предметом дополнительных исследований.

Список литературы

Lemire, D., Ssi-Yan-Kai, G., & Kaser, O. (2016). Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, *46*(11), 1547–1569.