Moscow Institute of Physics and Technology
(National Research University)
Phystech School of Radio Engineering and Computer Technology
Department of Theoretical and Applied Informatics

Study major: 03.03.01 Applied Mathematics and Physics
Study minor: Radio Engineering and Computer Technology

SUCCINT INDEX APPLICATION FOR FULL-TEXT SEARCH

(Bachelor's study)

Student:
Sokolov Vadim Andreevich

(student's signature)

Supervisor:
Neganov Alexey Michailovich

(supervisor's signature)

Moscow 2021

Abstract

Goals and objectives of the work.

This paper is dedicated to research on data structures that use succinct indices for textual information storage.

The goal of this work is to verify the efficiency of different data compression methods. Application of succinct index in practice is examined to data of a certain type. A comparison is made both with traditional solutions (suffix array) and with more recent ones (radix tree) that use indexing approaches different from the data structures under study.

The Results Obtained.

It was possible to obtain comparative performance characteristics of data structures under study. Were measured and analysed:

1. the volume of memory used;

2. time used for a substring search;

Compressed data structure - succinct suffix array was implemented in Go programming language. Memory consumption results for compressed index storage and speed of substring search are shown in this paper.

Contents

1 Introduction

Text data processing is used in a wide range of tasks in the modern computer industry. Dealing with text data inverted index [1] is commonly used. This data structure allows one to index words in documents. But there are several problems for which word indexing is not a good solution. For example, in some bioinformatics problems [2] pattern search takes place in DNA-code or protein sequence structure, which is not words and therefore inverted index can't be applied in this case. Similarly, in some Eastern languages (Chinese, Arabic) text can not be divided in separated words. In addition, an inverted index is limited when it comes to searching for a string similar to the required one but not totally identical to it. As an example, during the information search via search engines, where text with spelling mistakes is used as a request, the inverted index can not give the right reply. In these cases, indexes are built on unsorted suffixes of the initial text. An example of this index is a suffix array [3]. Substring search is implemented in a suffix array, which allows us to resolve the problems that cannot be solved by an inverted index. On the contrary, the suffix array can take 50 times more space than the initial text. Therefore it might be more cost-effective to execute simple text scanning in RAM instead of having an index that does not fit in RAM because I/O operations in outer memory are significantly more cost-intensive.

The increase in information search on the Internet leads to additional expenses in data storage and search. In this regard, there is a necessity to research different methods of reduction in consuming memory without significant costs on data search.

One of the possible solutions for this type of problem is the application of succinct data structures [4]. Depending on the level of information compression, data structures can be divided into implicit, succinct, and compact. Succinct data structures use an information volume that is close to theoretically minimal. In addition to that, in contrast to archives and other compressed solutions, it is possible to provide effective search operations. Let us assume that it takes a Z bit for storing a certain amount of data. Succinct data structures take $Z+o(Z)$ bit. For example, the data structure that takes $Z+\ln(Z)$ bit of memory is succinct.

Data is not always compressible. Apart from that, not all the data is reasonable to compress in terms of its efficiency of it in an uncompressed state. This paper proposes to consider compression of the index of suffix array, built for different textual data. At the same time, the text stays uncompressed.

In order to present the data in a compressed state it is necessary to prepare data in a special intermediate format. In this paper, Elias-Fano

encoding algorithm [5] is used. This algorithm allows us to compress an ascending sequence of non-negative integers.

While developing the data structure of the succinct suffix array memory optimization was used consisting of adding indexation over a bitmap that makes it possible to use less memory than similar approaches that use additional data structures.

The goal of this research is to study memory consumption for succinct suffix arrays over texts of various content and comparison with existing approaches. Substring search functions have been developed and the analysis of their efficiency has been carried out.

2 Problem Statement

First of all, compressed representation requires several manipulations over the inital suffix array. Because of that, there is a necessity to use intermediate ψ-array. This array implies a reorganized suffix array that consists of several sets of ascending sequences of indexes.

Ascending sequence of non-negative integers is compressible, which was shown in this paper [5]. In order to compress this sequence it is necessary to use Elias–Fano algorithm, which makes it possible to write ψ-array in a bit vector representation.

One of the characteristics of Elias–Fano representation is the opportunity to recover the values of ψ-array by index. Operations *rank* and *select* are defined for bit vectors, that work for constant time [6].

Objectives of the work:

1. Building of suffix array for given text (symbolic sequence).
2. Building of ψ-array over constructed suffix array.
3. Implementation of Elias–Fano compression algorithm.
4. Implementation of a function for getting an index in suffix array by ψ-representation.
5. Implementation of the index unpacking algorithm in ψ-array.
6. Implementation of a function for a substring search in the initial text.
7. Implementation of benchmarks for comparison of succinct suffix array with traditional suffix array and radix tree.

3 Literature overview

Text retrieval

Currently, the Internet is updated with a large amount of data every day. That is why it is extremely important to organize the search for necessary information in an *effective* way. In search within a document, indexing is applied. *Inverted index* is traditional for text indexing. It is a data structure in which for each word in a document collection a corresponding list contains all the documents of a collection where this word is present. While processing a multiple-word request the intersection of lists is used, that corresponds to each word in a query.

Problems of traditional approaches

Some cases take place in practice where it is impossible to use traditional full-text search by words [7]. A search model in which the task is to find orthographically close words (*fuzzy search*), that is used for spelling correction in numerous text editors, does not allow one to solve the problem using a full-text search. The same applies to different systems where *pattern matching* is used [8]. Another example of texts where it is difficult to apply traditional search methods is several eastern languages. In these languages, words are not separated by spaces which makes *inverted index* difficult to use. Finally, large texts combined from the alphabet with a small number of symbols can be attributed to «inconvenient» texts. A typical example of these texts is DNA and protein structure code. This text is not separated by words that is a key issue in using *inverted index*.

To solve these problems it is necessary to perform a substring search. Existing methods (prefix tree, suffix array) take too much space therefore they are not effective. For example, a prefix tree that contains n words with C average number of symbols in a word takes $O(n \cdot C)$ memory [9]. Let's take a greater look at suffix array as one of the most popular textual information indexation methods.

Full-text search using suffix array

A suffix array is a data structure used in the full-text search that allows one to perform substring search in a text consisting of n symbols in $O(\log n)$ [3]. To store n words in a suffix array it is necessary to use $O(n \log n)$ of memory. For instance, 2^{32} ASCII-text requires $2^{32} \cdot 8 = 32$ GB memory space. For this text, the suffix array must contain 32-bit elements. Therefore, suffix array size reaches $2^{32} \cdot 32 = 128$ GB memory space that is 4 times greater than the initial text size.

A suffix array is a sorted in a lexicographic order sequence of suffixes. In order to better understand the structure of suffix array, consider this data structure using as an example word «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 consisting of these suffixes would be presented as:

10 7 4 1 0 9 8 6 3 5 2

Usually, the end of the document is indicated by a special symbol that is not included in the alphabet of a text stored in the array. As a delimiter in this paper dollar symbol $ has been chosen. Modern approaches to suffix array construction allow one to build data structure in $O(n)$.

Radix tree

A radix tree is a compressed prefix tree. In its turn, a prefix tree is a data structure that represents the interface of an associative array and allows one to store values in key-value pairs. In this paper, the string is chosen as a key, and in contrary to regular trees, the edges can contain either one element (symbol), or a sequence of elements (string).

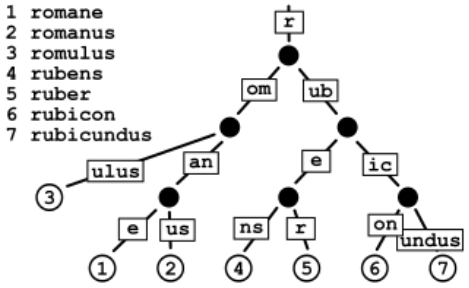


Figure 1 – Radix tree example

Radix tree is commonly used in the Linux kernel to bind a pointer with a long integer key [10]. This data structure is effective from the perspective of search speed and data storage. At the same time, the Radix tree is used in IP-addressing because it is extremely useful for hierarchical IP-addressing structure [11] storage.

Let's take a detailed look at radix tree characteristics. Let us assume that it is required to store keys the size of k storing n elements at the tree. Then insertion of an element, search and removal take $O(k)$ operations [12].

For many dictionary search requests radix tree can be faster and more effective than hash tables. Despite the fact that often the complexity of search in a hash table is considered to be $O(1)$, the necessity for the initial caching of input data is ignored in this assumption. It takes $O(k)$ operations where k is the length of an input string. Radix tree performs the search in $O(k)$, without initial data caching, and has greater cache localization characteristics. It also preserves the order allowing one to perform an ordered scan, get min/max

values, scan over common prefix, etc.

Because of all these advantages, radix tree is widely used in a Redis database, program products of HashiCorp, such as Terraform, Consul, Vault, Nomad, etc [13], that is of additional interest for studying it on textual data and comparison with suffix array.

Succinct data representation

One of the most important problems of this research is a program implementation of succinct index storage for suffix array. That is why it is important to understand what data can be considered succinct.

There are several degrees of information compression. Let us assume that the storage of a piece of data takes Z bit. Succinct data structures take $Z + o(Z)$ bit, implicit – $Z + o(1)$ bit and compact – $O(Z)$ bit [14]. In this paper, it is proposed to compress the suffix array in succinct data representation. Thus, the size of an array is only a little bit greater than the size of the initial text.

Ψ-array

An important part of a theory of succinct index representation is taken by an intermediate ψ-array. A characteristic feature of all the compression algorithms of a compressed suffix array (CSA) is that not the index itself is compressed but its representation as a ψ-array. An auxiliary array can be obtained from a suffix array by the following index manipulations. Let us consider a ψ-array construction by the example of a familiar string **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

Let us call the largest suffix of a substring except for the initial string as a successor. Therefore, for substring issippi$ successor is ssippi$. Ψ-array points out a successor for every chosen suffix. As an example consider $SA[7] = 8$. Its successor is a position in a suffix array that contains $8+1 = 9$. Thus, $\psi[7] = 6$. The main relation between a ψ-array and suffix array:

$$SA[i] = SA[\psi[i]] - 1 \tag{1}$$

It is necessary to emphasize that $\Psi[0] = \$$, because there is no successor for the first element in a suffix array. However, $SA[5]$ is referred to 0 index, that is the whole string overall. In other words, the element of the suffix array is referred to the beginning of the text, that is no other suffix can have it as a child. There is no element equal to 5 in a ψ-array, respectively.

Suffix array reconstruction from a ψ-array

There is an opportunity to reconstruct the initial suffix array by generated ψ-array with operations that are reversive to those that were present in the last paragraph. Let us have a better look at the specific features of this algorithm.

First of all, it is necessary to find the index of the element of the suffix array that has not been present in the ψ-array. In our example, it is equal to 5. From the discussions above, it can be seen that $SA[5] = 0$. Thus, one element of an initial suffix array was reconstructed. From the formula 1 for $i = 5$ follows:

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

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

$$SA[4] = 1$$

Another element of a suffix array is reconstructed! This way the whole remaining sequence can be reconstructed. Time taken to find the first element is equal to $O(n)$ operations. There are several improvements in the search speed of certain elements in a suffix array that require additional data structures storing suffix array values at every $\log n$ step [15]. This approach goes beyond this paper because the main problem of it is to find the best index compression.

Ψ-array compression

Let us have a detailed look at ψ-array:

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

Sorted suffix array looks like that:

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

It is worth mentioning the structure of it: it contains a set of adjusting indices. Let us consider the ascending sequence 2, 3, 8, and 9.

$SA[2] = 7$, $T[7] = i$. $SA[3] = 4$, $T[4] = i$.

It is noteworthy that $SA[6] = 9$, $T[9] = p$. Thus the alternation of the index from increase to decrease corresponds to the switch of symbols from i to p.

Lemma 1. *Two consequent elements of a ψ-array are ascending if corresponding suffixes that they are referred to start from the same symbol.*

Proof. By construction, a suffix array is a set of indices in the initial text, sorted in lexicographic order. It means that $T[SA[i]] < T[SA[i+1]]$ $\forall i \in [0, n)$. These two suffixes have a common first symbol. That is why they are different in the following elements, e.g. in the second one: $T[SA[i]+1] < T[SA[i+1]+1]$.

Ψ-array is referred to a successor of a given suffix: $T[SA[\psi[i]]] < T[SA[i]+1]$. Because $T[SA[i]]$ and $T[SA[i+1]]$ have the same first symbol, their successors are ordered. If $T[SA[\psi[i]]] < T[SA[\psi[i+1]]]$, then because of orderliness the index in the suffix array for the first one should precede the index for the second one. Therefore, indices (values of ψ-array) should be ordered if suffixes have a common first symbol. □

Ascending sequences of non-negative integer numbers are compressible [15]. The easiest way to store these sequences is a bitmap. However, this method does not allow one to have random access to the elements. Let us consider in detail a more advanced algorithm of succinct data representation.

Elias–Fano Encoding

Compression via Elias–Fano method [5] allows one to represent monotonically increasing sequences of non-negative integer numbers in the form of bit vectors. This approach makes it possible to store a non-decreasing sequence of n integer values the size of $[0, m)$, taking $2n + n\lceil \log m/n \rceil$ bit, allowing access to the i-th element in $O(1)$. Comparing the size of the data structure with the minimum possible space in a memory from the theory of information point of view, Elias–Fano coding is a succinct index.

In the beginning, each number from the sequence is encoded by $\log m$ bits of data. Binary elements representation is divided into two parts: the upper one, containing the first $\log n$ bit, and the lower one with the remaining $\log m - \log n = \log m/n$ bit. The union of lower bits takes $n \log m$ bit. Upper bits are a set of $n + m/2^{\log m/n}$ bit [16]. Starting from an empty bit vector, we add 0 to this part as a stop-bit for each possible value represented by the bits of an upper part. For each actually present value, 1 is added, setting it before the corresponding stop-bit.

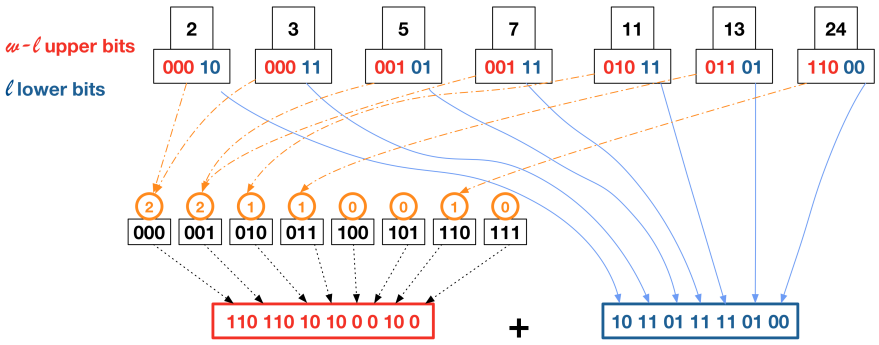


Figure 2 – Example of the code construction for Elias-Fano

Let us consider as an example a sorted sequence 2,3,5,7,11,13,24. Figure 2 shows a coding scheme. The largest number in the universe is 24. Therefore for the representation of each element, it is necessary to allocate 5 bits per element. Then the binary representation is separated into two parts: upper and lower. Starting from the conditions described before, let us choose 3 bits for an upper part and 2 bits for a lower one. Overall there are 7 elements in a sequence. Let us consider number 2. For it, we have $2 = 0b00010$ and 000 as an upper part and 10 as a lower part correspondingly. Repeating this process, a set of upper and lower parts will be obtained for each element. Then these parts are combined together. For upper parts, there are 2^3 variants of sets of values. With each of these numbers, a counter is associated that increases by one if a number from the sequence has the same upper part. So for 2, the set 000 is incremented. A number $3 = 0b00011$ with 000 upper part goes to the same set. For a number $5 = 0b00101$ with 001 upper part, the set 001 is incremented, and so on. Finally, the unary coding is performed, adding as many 1s in a counter representation as it is a value of each counter after which there is a 0-bite. The resulting Elias-Fano encoding is a union of obtained upper and lower parts.

Data resconstruction from the Elias–Fano code

For an initial reconstruction of a monotonically non-ascending sequence of numbers, it is necessary to desing an operation with an access to the i-th element of a sequence, $i \in [0, n)$. To access the lower part it is possible to get corresponding bits because the length of a vector for the element storage is known. To get the upper part it is necessary to perform a *select* operation. *Select(i)* the operation allows one to get a position of the i-th bit set to 1 in a bit-vector. This operation takes $O(1)$ time that provides random access to an element of the sequence without complete decoding of the whole sequence overall [6].

4 Experiment overview

The experiment consists of several main parts. First of all, study of traditional data structure – suffix array is carried out from the perspective of consumed memory, time to build the index and substring search. Then more modern radix tree is studied, in which similar functioning aspects are covered. One of the most important part of the work is a development of compressed suffix array (CSA). At the same time it includes not only testing of CSA with benchmarks similar to other overviewed data structures but study of the degree of index compression in comparison to suffix array. It is important to emphasize that a comparison of algorithm on texts with different context is of particular interest for us.

4.1 Experiment platform

An experiment consisting of running benchmarks was carried out on a machine with the characteristics that are presented in table 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

Table 1 – Computer characteristics

4.2 Suffix array

In order to analyze the data structure, a suffix array from a standard Go library was taken into consideration [17]. It takes as an input a text and constructs a sorted index from it with that it is possible to perform substring search operations. Index construction takes $O(n)$ operations, where n is the initial text size. Substring search takes $O(\log n \cdot |s|)$, where $|s|$ is the substring length. Let us consider benchmarks more carefully. The initial text is loaded from the file, then the substring is chosen that is limited by positions $[leftPos : rigtPos]$.

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

Listing 1: Suffix array example

Listing 1 shows a simplified code of a benchmark for a substring search in a suffix array (less important for general understanding details are skipped). The code is tested for 5 texts with different contents. The substring size for data search is the same for each measurement.

Because suffix array size does not depend on the alphabet size the results for different texts vary within the margin of error. That is why it is enough to show for clarity an example for the same text type. Table 2 shows the results of measurements of the index construction for the Amazon Text Corpora.

Text size, KB	Memory, KB	Time, 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

Table 2 – Suffix array construction

Text size, KB	Memory, KB	Time, 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

Table 3 – Substring search in suffix array

4.3 Radix tree

Radix tree implementation in Go language is taken from the following source: [17] The main reason for this decision is caused by the simplicity of insertion of the benchmark in the existing testing system in the Go language. From the algorithmic point of view, this implementation confirms the theoretical description of a substring search complexity that is $O(1)$.

```
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)
  }
}
```

Listing 2: Radix tree example

Listing 2 briefly shows the radix tree construction function. In the beginning, a set of substrings is generated from the text that is read from the file. In *fillRadixTree* function substrings are added to the tree. Comparison of radix tree construction for Amazon Text Corpora [18] are shown in a table 4.

Text size, Kb	Memory, Mb	Time, 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

Table 4 – Radix tree construction

In listing 3 below the example of a tree substring search function is given. This data structure shows quite good results of a substring search in a constant time. It can be seen in the table 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
}
```

Listing 3: Radix tree lookup

Text size, Kb	Memory, Mb	Time, 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

Table 5 – Substring search in radix tree

4.4 Compressed suffix array

Let us move on to the implementation of CSA. As it was mentioned in the literature overview, it is necessary to construct the index the same way as it was done for a suffix array. It takes $O(n)$ operations [14].

CSA structure

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

Listing 4: CSA structure

In listing 4 a simplified structure of CSA is shown. It consists of the initial text (only the index is compressed, the text remains in the same condition), index, ψ-array, and a text length. The index is build with a function that uses a suffix array construction algorithm taken from Go library [17].

Ψ-array construction

In order to construct a ψ-array, it is necessary to assume that $\psi[0] = \$$. Then make a traversal over the suffix array and find an index that corresponds to a value in the suffix array that is equal to a current one with an addition of 1. Listing 5 shows a pseudocode of the algorithm.

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

Listing 5: CSA construction

Additional data structures

A bitmap from the Roaring.bitmap package [19] is used to store the data. This bitmap implementation is fast and effective. Also, Roaring is used for many products such as Apache Druid, LinkedIn Pinot, Google Procella, etc.

Obtained ψ-array is a set of monotonically non-decreasing sequences of numbers. Elias-Fano algorithm allows one to transfer each of these sequences to a bit-vector separately. The quantity of these sequences is equal to the size of the alphabet used in a text [15]. A question arises: how is it possible to organize the storage of these bit-vectors?

One of the solutions is to use two additional arrays: the first one to store the offset in a ψ-array and the second one to store a symbol corresponding to

the ascending sequence. In this paper, text coding is represented by ASCII code or a less size code. Thus the alphabet size is limited to 128 symbols. Therefore the size of additional arrays is not greater than $2 \cdot m$, where m is the size of the alphabet. An array with offsets is used for fast indexing over an array of bitmaps.

Additional data structures fulfillment and offset calculation are performed during the compression of each separate monotonically non-decreasing sequence of indices. To extract these sequences a notion described in Lemma 1 is used, A termination in ascending of a ψ-array corresponds to a symbol switch. This way it is possible to index bitmaps and fulfill supporting arrays with symbols of the alphabet used.

Elias-Fano

Compression using Elias-Fano is performed for each separate sequence. At the same time, a preliminary construction of the upper and lower parts of the bit representation of the elements of this sequence is performed. The offset is calculated for the following fast access to lower bits. During the compression procedure, numbers are written in a bitmap, where it is possible to index over in a constant time.

In order to get access to an element of the sequence (part of ψ-array), it is necessary to use the $select(i)$ function implemented in a bitmap, taking $O(1)$ time. To check the correctness of the algorithm a function of getting a whole ψ-array was developed appoaching $select(i)$ in it.

Suffix array reconstuction

There is no necessity to completely decode a ψ-array to get access to an element of a suffix array $sa[i]$. There is a need for performing a traversal over ψ-array until the last element is not reached. Counting the number of steps until the last element $h(i)$, an index of a desired element can be found by a simple calculation: $sa[i] = n - h(i)$. This algorithm takes $O(n)$ operations [15].

Final CSA structure

After adding supplementary data structures CSA looks as it is shown in the listing 6.

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

Listing 6: CSA structure

It is necessary to emphasize that now there is no need to store indices and additional ψ-array. Instead of that, the data is stored in a compressed way in a sequence of bit vectors. Apart from that, the initial text is still stored in the same representation.

Thus to store compressed index it takes $n \cdot \log|\sigma| + o(n)$, where $|\sigma|$ is an alphabet size. Binary search over suffix array takes $O(\log n)$ operations. To get an index from a suffix array from a ψ-array it is necessary to perform $O(n)$ operations. In total $O(n \cdot \log n)$ operations are needed to find an element.

CSA testing

Let us consider the efficiency of CSA in the example of five texts with different contexts. In the beginning, it is required to estimate array construction time and necessary memory storage size. Table 6 shows data for Amazon Text Corpora.

Amazon		
Text size, KB	Memory, MB	Time, 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

Table 6 – CSA construction

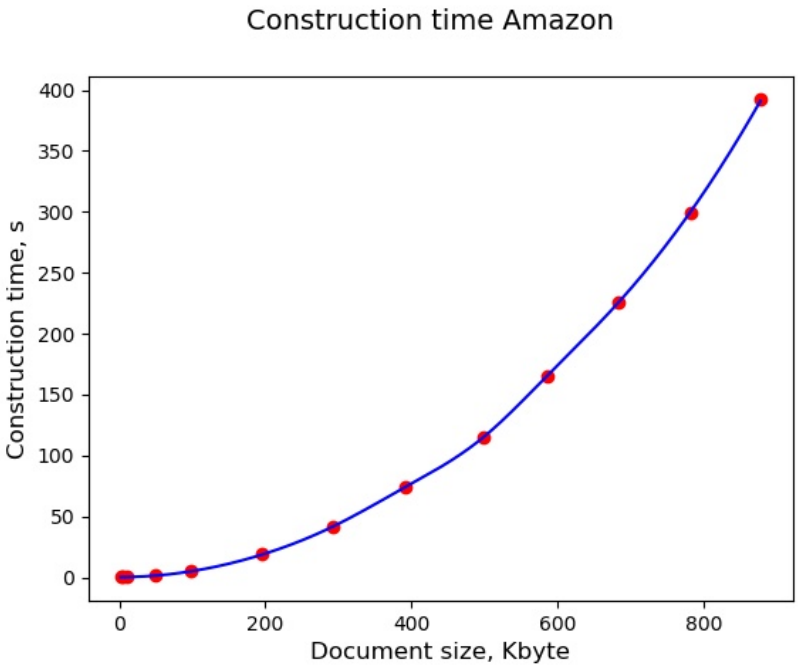
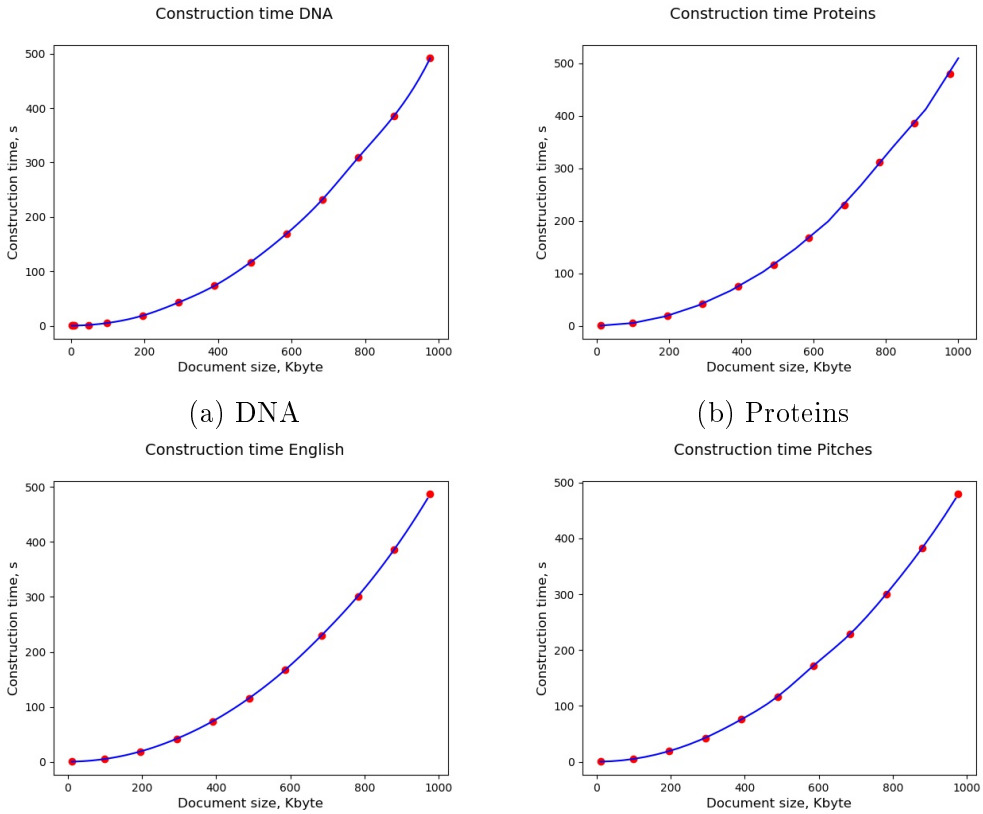


Figure 3 – CSA construction



(a) DNA (b) Proteins

(c) English (d) Pitches

Figure 4 – CSA construction

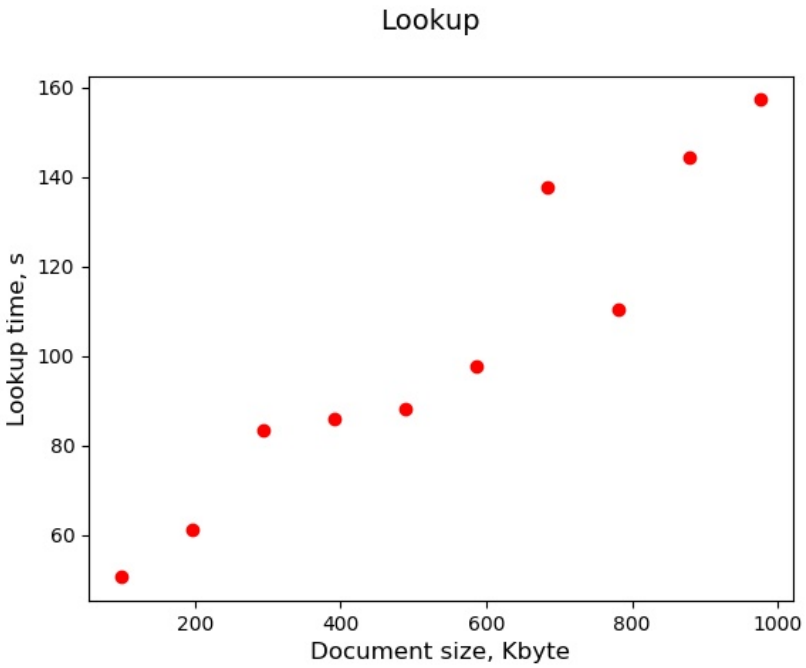


Figure 5 – Substring lookup in CSA

Let us consider the results of a dependency between a substring lookup speed and text size for CSA, suffix array, and radix tree constructed for Amazon Text Corpora. Figure 6 shows that a substring lookup speed for a fixed-size substring does not depend on the text size for the suffix array and radix tree. Lookup time increases with an increase in the size of the initial text for CSA. The radix tree shows the best result, as was expected.

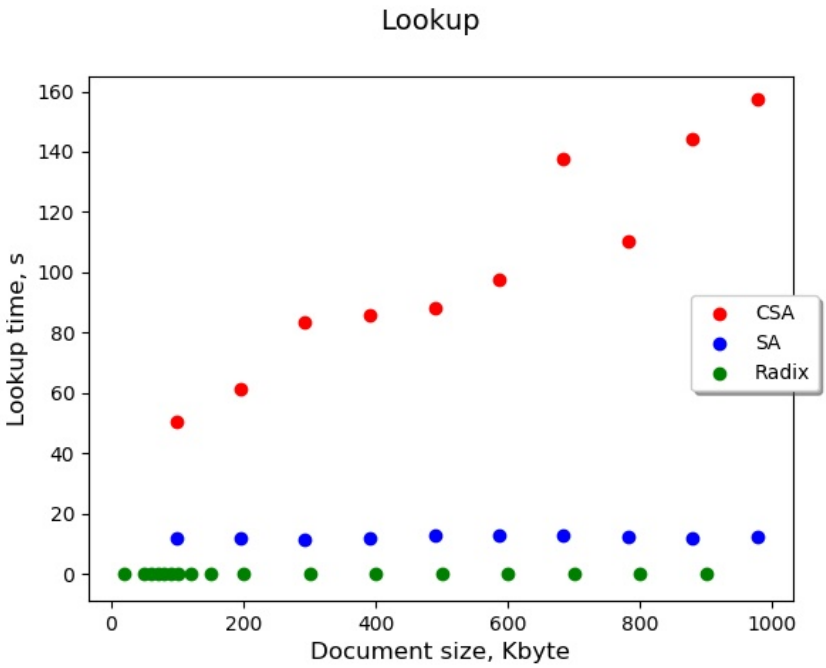


Figure 6 – Substring lookup time in CSA

4.5 Comparison

Figure 7 shows comparative characteristics of substring lookup for different textual data.

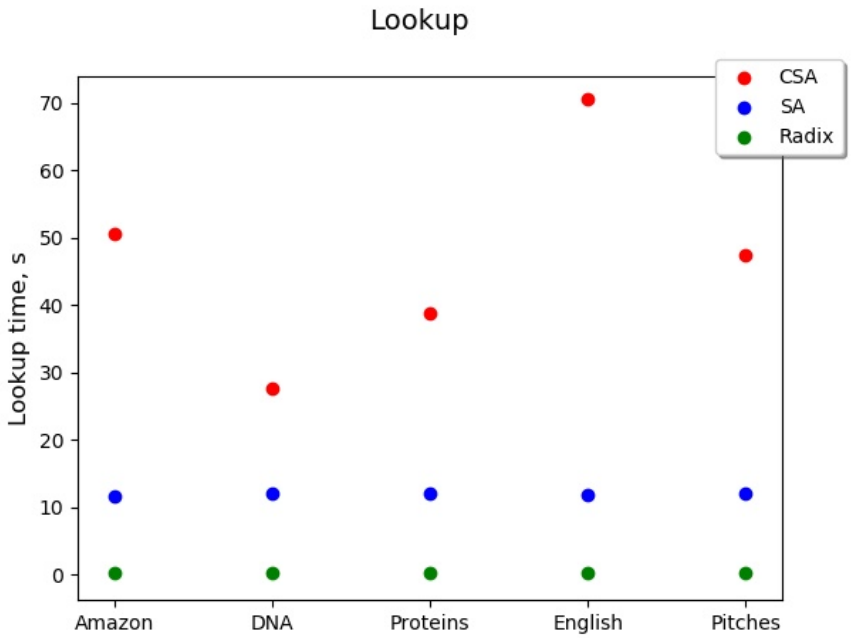


Figure 7 – Substring lookup in CSA

Let us consider the relative compression of the CSA with respect to a suffix array. The mean ratio of memory used for CSA to memory used for suffix array is calculated depending on the size of the initial text. Figure 8 shows a reduction of the ratio, that is compression coefficient increase. Remarkably, for small sizes of the initial text, CSA takes more space than the suffix array. It is explained by the usage of additional data structures described in the previous pages. At the CSA construction, they consume memory comparable by the order to the index size in a succinct representation. For large texts, CSA becomes more effective in comparison to the classic suffix array.

Relative compression CSA/SA for different texts is shown in figure 9. It is possible to explicitly notice the dependency between a succinct index size from the alphabet size, that can not be said about the suffix array and radix tree.

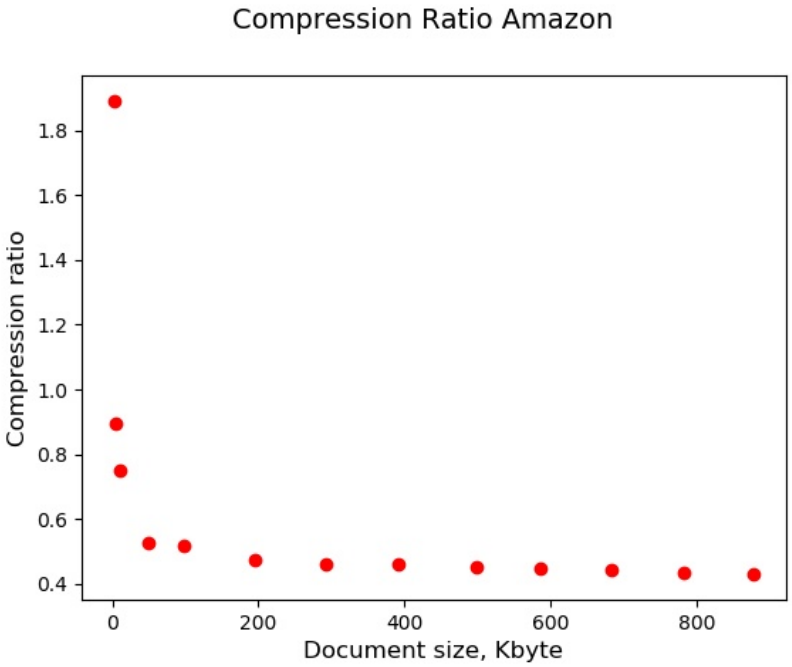


Figure 8 – Compression ratio of CSA to SA

5 Discussion

A considered data structure, the compressed suffix array (CSA) showed its efficiency in terms of memory used for index storage. In comparison to traditional suffix array as it was demonstrated in the graphs of comparable characteristics, with a large enough size of the initial text, CSA consumes less memory than suffix array while preserving the opportunity to perform a substring lookup. As was mentioned in the discussion of dependency between compression and the text length, at a small initial text size suffix array is more effective in terms of memory consumption that is caused by additional costs for maintaining supplementary data structures used for succinct index construction. At the same time, an algorithm of index construction, as well as a substring lookup in CSA, work slower than in the classic one. Nevertheless, there are several methods of the speed increase in these algorithms allowing CSA to achieve results that are not worse than suffix array ones [15].

The results obtained by the comparison of the radix tree with other studied data structures confirm the efficiency of substring lookup in the radix tree in constant time. However, text indexation requires fulfillment of the tree by substrings of the initial text, that takes significantly more memory than

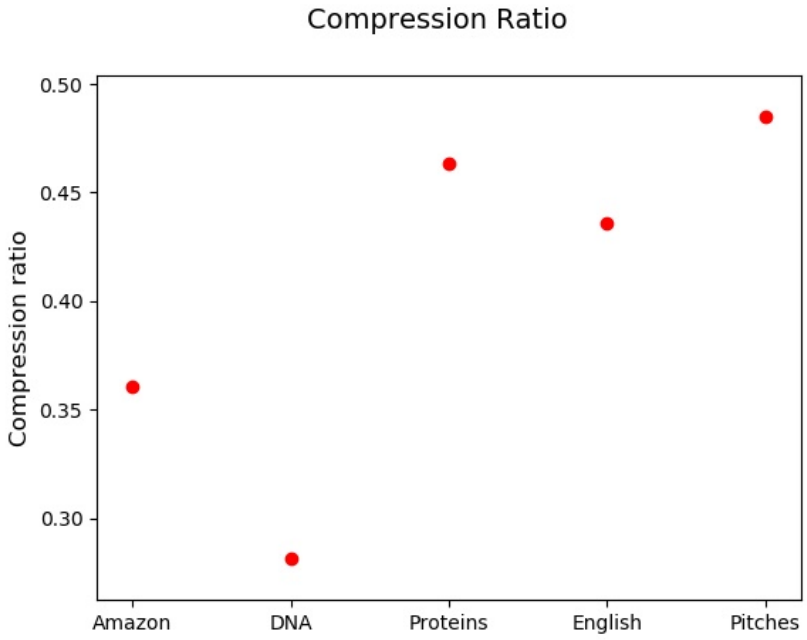


Figure 9 – Compression ratio of CSA to SA

suffix array and CSA. Similarly, the filling speed for a tree construction does not allow one to consider radix tree as an effective method for operations with substrings of the initial text.

6 Conclusion

Using Go language data structure compressed suffix array (CSA) was implemented. It has a functionality of a traditional suffix array allowing one to solve problems of substring lookup. Elias-Fano algorithm was implemented allowing one to present an index in a succinct representation. Comparative characteristics of the work for construction and substring lookup for CSA and suffix array were combined for texts with different contexts. The dependency between memory consumed by data structures and the size of the initial text was studied. A comparison between the radix tree and these data structures was performed by construction time and substring lookup time as well as by memory consumed.

In terms of memory used, CSA showed its efficiency in comparison to suffix array and radix tree, but the algorithm of index construction and the substring lookup algorithm are less fast than similar algorithms in a traditional suffix array. Application of the CSA in real data was considered for applied problems such as substring lookup at a protein sequence and pattern matching at DNA code used in bioinformatics. The efficiency of substring search using the radix tree was demonstrated. The complexity of its application was shown for textual data because of the large data consumption at the tree construction step.

Further studies in this research field can rely on the development of a succinct representation of existing classical data structures. Application of compression methods such as considered in this work Elias-Fano algorithm can make it possible to perform succinct representation of different data types converted to the right format beforehand. Based on the experiments advantages and disadvantages of studied algorithms and data structures functioning with real applied problems were identified. Succinct data structures can be used in algorithms utilizing index storage in outer memory. This topic is a little-studied one which is why index compression is of interest in the field of substring indexing in outer memory.

Bibliography

[1] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6–es, 2006.

[2] Yoshimasa Tsuruoka, Jun'ichi Tsujii, and Sophia Ananiadou. Facta: a text search engine for finding associated biomedical concepts. *Bioinformatics*, 24(21):2559–2560, 2008.

[3] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[4] Guy Joseph Jacobson. *Succinct static data structures*. Carnegie Mellon University, 1988.

[5] Giulio Ermanno Pibiri. *Dynamic Elias-Fano Encoding*. PhD thesis, Master's Thesis, University of Pisa, Pisa, Italy, 2014.

[6] Antonio Fariña, Susana Ladra, Oscar Pedreira, and Ángeles S Places. Rank and select for succinct data structures. *Electronic Notes in Theoretical Computer Science*, 236:131–145, 2009.

[7] Hannah Bast and Marjan Celikik. Efficient fuzzy search in large text collections. *ACM Transactions on Information Systems (TOIS)*, 31(2):1–59, 2013.

[8] Xiao Bai, Cheng Yan, Haichuan Yang, Lu Bai, Jun Zhou, and Edwin Robert Hancock. Adaptive hash retrieval with kernel based similarity. *Pattern Recognition*, 75:136–148, 2018.

[9] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[10] Linux Inside. Data structures in the linux kernel, https://0xax.gitbooks.io/linux-insides/content/datastructures/linux-datastructures-2.html, 2018.

[11] Emile Hugo. How radix trees made blocking ips 5000 times faster, 2019.

[12] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[13] ProgrammerSought. Simple explanation of redis radix tree, 2019.

[14] Hongwei Huo, Longgang Chen, Jeffrey Scott Vitter, and Yakov Nekrich. A practical implementation of compressed suffix arrays with applications to self-indexing. In *2014 Data Compression Conference*, pages 292–301. IEEE, 2014.

[15] David G Andersen. A simple introduction to compressed suffix arrays. 1996.

[16] Antonio Mallia. Sorted integers compression with elias-fano encoding, 2018.

[17] Golang.org. Package suffixarray, 2016.

[18] Jure Leskovec. Web data: Amazon reviews, 2013.

[19] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.