Random access to compressed text

Fedor Nikitin

Master's thesis April 17, 2007
Department of Computer Science and Statistics
University of Joensuu

Random access to compressed text

Fedor Nikitin

Department of Computer Science and Statistics
P.O. Box 111, FI-80101 Joensuu, FINLAND.

Master's thesis

Abstract

Random access to compressed text can be seen from two points of view. From the first one it is considered as some part of text compression area, where compression methods are designed in special way to allow random access to encoded text. From another one the text is seen as a special data structure (sequence) and the topic goes to area of succinct data structures.
In the thesis current state of mentioned areas is considered. New method which might be useful in practice for random access text compression is developed. Also, adaptation of Fibonacci coding scheme for the random access compression is under discussion.
The theoretical analysis as well as experimental results for proposed methods are presented. These results show competitive ability of the methods compared to modern compression techniques.

Computing Reviews (1998) Categories and Subject Descriptors:

ACM Classification: E.1 [Data structures]; E.2 [Data storage representations]; E.4 [Coding and information theory] — *Data compaction and compression*; F.2 [Analysis of algorithms and problem complexity]; H.3.1 [Information storage and retrieval]: Content Analysis and Indexing — *Dictionaries, Indexing methods*

Key words: algorithms, text compression, succinct data structures, succinct sequences, random access compression, Fibonacci coding, simple dense coding, self-delimiting integers

Contents

1 Introduction **1**

2 Text compression **5**
- 2.1 General text compression methods 5
  - 2.1.1 Ad hoc methods 5
  - 2.1.2 Statistical coding 6
  - 2.1.3 Dictionary-based coding 10
  - 2.1.4 Block sorting compression 12
- 2.2 Compression of natural language 13
  - 2.2.1 Model of English language 13
  - 2.2.2 Tagged and plain Huffman codes 16
  - 2.2.3 End-Tagged-Dense code 17
  - 2.2.4 (s, c) - Dense code 18
  - 2.2.5 Word-based block sorting compression 19
- 2.3 Compression methods for the text of positive integers 19
  - 2.3.1 Elias codes 19
  - 2.3.2 Fibonacci coding 21

3 Succinct representations of sequences **23**
- 3.1 An overview of succinct data structures 23
- 3.2 Succint data structures on bit vectors 29
  - 3.2.1 Queries on bit vectors 29
  - 3.2.2 Succinct data structures with explicit storage 29
  - 3.2.3 Succinct representations within entropy bounds 32
  - 3.2.4 Succinct representations using gap encoding 34
- 3.3 Succinct representations of sequences 35
  - 3.3.1 Queries on sequences 35
  - 3.3.2 Succinct sequences for *Rank*, *Select* and S_q queries 35
  - 3.3.3 Succinct sequences for *Substr* query 39
  - 3.3.4 *Rank* and *Select* queries for large alphabets 40

4 Compression methods for random access **43**
4.1 Simple dense coding scheme with random access 43
4.1.1 Simple dense coding scheme 43
4.1.2 Random access to compressed sequences 46
4.1.3 Space complexity 47
4.1.4 Extensions 50
4.2 Fibonacci coding of sequences with random access 51
4.2.1 Fibonacci coding applied for sequences of symbols 51
4.2.2 Data structure for random access 53

5 Experimental results **57**
5.1 Test files 57
5.2 Compression performance 58
5.3 Performance of random access compression 60
5.4 Discussion 62

6 Conclusions **63**

References **64**

Chapter 1

Introduction

Nowadays, data compression is a widely used area of computer science. Small is beautiful. It is good when we can present some information using as little space as possible. Many methods of data compression were discovered and they found their applications in different fields of science and engineering. However, one compromise is made in classical data compression. This compromise concludes in decreasing of accessibility of compressed information.

Usually compression routine involves two opposite to each other procedures. The first one is called *encoding* and can be thought as some transformation of original information into compressed form. If this transformation is reversible the compression method is *lossless*, otherwise *lossy*. For lossless methods as well as for lossy ones some inverse transformation may be constructed and procedure of performing this transformation is called *decoding*. The difference between lossless and lossy methods lies in fact that for the lossless methods the result of inverse transformation is exactly the original information. Unlikely, for lossy methods the result of inverse transformation under compressed information is only close in some sense to original one.

Accessibility of compressed information in wide sense is meant as an usage of information without the second procedure, i.e. decoding. If for some application we have developed compression method which allows us to perform all needed operations without decoding, we can completely replace the information by its compressed form. This is extremely useful for the applications when both time and space are of importance. Compression decreases the space used by application, but processing time is not increased, because decoding procedure is not involved before usage of information.

In this thesis the emphasis is made on text compression. This part of data compression deals with the methods for efficient representation of texts. By definition text is a sequence of symbols taken from some alphabet which is usually assumed to be finite and fixed. Text compression methods are usually lossless. The reason

is that even small changes in text may lead to completely another meaning of the text. For example substituting the first letter in the word 'sun' by the letter 'f' we have another word which extremely differs from the original one. Nevertheless lossy methods also might be useful for text compression. In the thesis for example I consider the compression method which skips all separators between words in the text and replaces them just by usual space. For some applications like text indexing such losses are tolerable.

Accessibility of compressed texts is an ability to perform operations which are made on usual texts. It may include retrieving particular symbol from the text or extracting some subtext. Popular operation on text is searching for some patterns or strings within the text and so on. The task of preserving accessibility of compressed texts from the text compression's point of view concludes in study such compression methods which allow to do that. Many compression methods were developed but small attention was paid on such properties of compression techniques. If we have fixed the operations which we want to do on compressed text we can study existing methods, modify them somehow or invent new ones to accomplish the goal of preserving accessibility.

Another point of view on the above task is provided us by succinct data structures field. *Succinct data structures* is relatively new area of computer science, which deals with efficient representation of data structures. Efficiency is meant in terms of operations which can be made on data structures. If a fixed set of operations are done fast, using small size close to the size of the original data structure or even less if we compress the data structures somehow, we think about this representation as efficient one. As it is mentioned above the text is just a sequence of some symbols, i.e. certain data structure. Hence we can consider our task as belonging to the field of succinct data structures.

There are several treatments of random access property of compressed text. Let $S = s_0 s_1 \dots s_n$ be original text over alphabet Σ ($\sigma = |\Sigma|$), i.e. $s_i \in \Sigma$ and $S' = s'_0 s'_1 \dots s'_h$ be its compressed form produced by some compression algorithm. S' is a bit sequence, i.e. $s'_i \in \{0, 1\}$. The first treatment is that we say that compression algorithm satisfies random access property if for arbitrary position i^* in S' decoding process can be started in constant time around position i^*. This property informally tells that the rest of the encoded sequence does not depend on its prefix. However, we can not answer the question what will be the first decoded symbol. We say that the compression algorithm satisfies random access property under second treatment if for any position i^* within the original text S it is possible to extract the symbol $s_{i^*} \in S$ in constant time, i.e. we can access any symbol of the sequence S in constant time. In this thesis I mean the second treatment of random access property.

The thesis is organized as follows. The first chapter devoted to the overview of modern text compression methods. I start with consideration of general compression

methods which may be applied not only for texts. There are two main directions: statistical coding and dictionary-based coding. The first one is based on information theory founded by Shanon and the second one is presented by different techniques including the most popular Ziv-Lempel algorithm. Block sorting compression which is quite new approach in text compression based on Burrows-Wheleer transform is also presented in this part.

Because we deal with the texts, the case of the texts on natural language is important. Properties of English texts and compression algorithms for this particular case are collected in separate section of the first chapter. This part is important for the further considerations because algorithms developed in the thesis are proved working well especially on English texts.

I finish the first chapter with the section devoted to encoding of texts of positive integers. Such kind of inputs occur, i.e. in telecommunication applications and the main difference is that no probabilities of the input symbols nor even alphabet can be defined beforehand.

The goal of the second chapter is to present an overview of current results and trends in succinct data structures area with emphasis on representation of sequences. Succinct data structures are aimed to find efficient representations for some classical data structures.

In the first section I list data structures which got the most attention in research. For majority of data structures the best current solutions are considered. The second section of the chapter deals with very important particular case of general sequences which is the case of binary sequences or bit vectors or bit streams. There are two reasons which proves the importance of this consideration. The first one is that historically binary sequence was the first data structures for which succinct representations were under research. The second reason of importance concludes in fact that for many other data structures succinct representation of binary sequence are used as internal building blocks.

Then I consider sequences of symbols taken from finite and fixed alphabet which are natural generalizations of binary sequences. Notice that this part of the second chapter is most relevant to the topic of the thesis. In fact this section describes current results on the subject of efficient representations of text from the position of succinct data structures. As for binary sequence I tried to mention all the best current solutions, but due to the lack of the space some solutions are missing.

In the third chapter two methods allowing random access sequences or texts are described. Simple dense coding is extremely easy technique which achieves good compression ratios for texts on natural language. Theoretical analysis of offered scheme is presented. Some extensions of the scheme are discussed. The rest of the chapter is devoted to Fibonacci coding. This method is quite old and the author of thesis even does not know where this method was firstly described. Notice that in the first chapter this method was already discussed, however for another task. I

show that this method can be easily adapted for the random access to compressed texts and gives competitive compression ratios. I also give theoretical estimations on the size of encoded sequence for Fibonacci method.

The thesis ends by the chapter with the experiments which were done for the proposed methods from preceding chapter. I have compared the compression ratios for the simple dense coding and Fibonacci coding with variety of modern compression methods.

All presented time complexities of algorithms in the thesis are meant in the sense of uniform RAM model of computation with $\Theta(\log_2(n))$ word size.

At the end of this introduction I want to express my thanks to people who have contributed to my work on this thesis. First of all I thank my scientific adviser Dr Kimmo Fredriksson who has provided ideas for the thesis and has answered all my questions. I express gratitude to the best IMPIT coordinator Wilhelmiina Hämäläinen for taking care about all of us and personally for help in choosing topic for the master thesis. I thank Dr Alexander Kolesnikov for the interest to my work and for interesting scientific discussions during study. Special thanks to my friends Denis Komarov and Ilya Mokhov who helped to accommodate me in Joensuu. I express gratitude to Yury Lakhtin and Maxim Dudochkin for support in any kind of situations and all IMPIT students. I thank Guy Jacobson for providing his PhD dissertation and Okanohara Sadakane for sharing paper and C sources.

At last but not least I express thanks to my parents, sister and brother. Thanks to all staff working at the Department of Computer Science and Statistics in the University of Joensuu and Finland for pleasant time.

Chapter 2

Text compression

2.1 General text compression methods

2.1.1 Ad hoc methods

We start discussion about the methods used in text compression with so-called ad hoc methods. Most of these methods are only interesting from historical perspective. Nevertheless, some of them have applications in current compression standards.

An ad hoc method is some simple technique of compression, which uses natural redundancy of data to be compressed. In the past a lot of such methods were invented. These methods do not exploit some theoretical foundations and for them compression is usually achieved by some simple tricks. In this subsection we deal with two such methods. They are *run-length coding* and *move to front coding.*

To get started suppose that an alphabet Σ of finite size σ is given. We are aimed to represent the sequence $S = s_0 s_1 \dots s_{n-1}$ of symbols from the alphabet Σ using as little space as possible.

In some texts it can occur that there are quite long sequences of the same symbol. In other words there are presented many consecutive repetitions of the symbol. The run-length coding utilizes this property. There are many variations of run-length coding scheme, but the main idea can be expressed as follows. Instead of storing the sequence of the same symbol, we store the code of this symbol and the number of times this symbol should be repeated. Every such pair of the symbol and the number is called a *run.* The compression is expected to be achieved, if the lengths of runs are large. Despite of the naivete of this approach, the run-length coding takes advantage of simplicity and is used, for example, in block sorting compression, which is an objective of subsections 2.1.4 and 2.2.5.

Move to front coding (MTF) is another technique which exploits the nearness of the symbols for compression, but not in such strict sense as run-length coding. MTF uses the dynamic list of symbols. Initially, all symbols in the list are presented in

some predefined order. When new symbol from the input comes, we send the index of this symbol in the list as an output and swap this symbol with the symbol on the top. This procedure continues while the end of the input is reached. So, the output of the move to front coding is the sequence of integers, which are the indices of the symbols in the list. We can expect that the most frequent symbols are somewhere in the top of the list during coding procedure and, hence, they will be presented by small integers in output. On the other hand, the codes of infrequent symbols are large integers. There are many methods of coding positive integers, which have the property that small numbers have shorter codes and the large numbers have longer ones. Some of these methods are under consideration in section 2.3. If after MTF transformation one of these methods is applied, we can expect that compression is achieved.

2.1.2 Statistical coding

Statistical coding also sometimes referred as *entropy-based coding,* is one of the fundamental approaches in compression area. The notion of the *entropy* is a basic idea for all statistical coding schemes. Informally, the entropy shows the average number of bits needed to represent a symbol within certain message, based on probability distribution of symbols. Hence, if we multiply this value on the length of the message measured in symbols, we get the number of bits needed to represent whole message.

The notion of the entropy comes from fundamental work by Shannon et al. [SW63]. This work has marked the birth of *information theory.*Suppose that, we have the finite set of events E ($|E| = n$) with probability distribution on it: $P = \{p_1, p_2, \ldots, p_n\}$. Shannon introduces the entropy axiomatically as a function $H(p_1, p_2, \ldots, p_n)$, which satisfies:

- H is a continuous function of $p_1, p_2, \ldots, p_n$.
- If $p_1 = p_2 = \ldots = p_n = 1/n$, H is steadily increasing function of n.
- The value of H is zero when one of $p_i = 1$.
- Consider the full system of events on E, i.e. the system $D_i \subset E$, $i = 1, \ldots, k$; $D_i \cap D_j = \varnothing$, $i \neq j$ and $\bigcup D_i = E$. On any D_i the conditional probability distribution is defined and hence we can consider the entropy H^{D_j}. Then for these values and H the following holds.

  $$H = p(D_1)H^{D_1} + p(D_2)H^{D_2} + \ldots p(D_k)H^{D_k},$$

  where $p(D_j)$ is the probability of the event D_j with the respect to the distribution P on E.

Shannon showed that there exists unique function which complies with the above axioms and it is

$$H(p_1, p_2, \ldots, p_n) = -\alpha \sum_{j=1}^{n} p_j \log p_j.$$

The coefficient α and the base of the log function depends on the unit in which the entropy is measured. In this thesis we measure the information in bit units and consequently $\alpha = 1$ and the logarithms are taken with base 2. The entropy becomes

$$H(p_1, p_2, \ldots, p_n) = -\sum_{j=1}^{n} p_j \log_2 p_j.$$

So far we have talked about events and defined an entropy for the set of events. When we deal with text the event for us is the occurrence of certain symbol from an alphabet in the input text and the probability of this event is the probability of occurrence.

The fundamental importance of the entropy lies in *noiseless source coding theorem* proven by Shannon. Informally, it states that there is no coding scheme that produces the average number of bits per symbol less than the entropy. This result gives us tool for comparison of coding schemes and also defines the meaning of optimal coding scheme. The *optimal coding scheme* is such which achieves the entropy value for the average number of bits per symbol. Much efforts were paid for inventing coding schemes, which are close to this entropy lower bound. We talk about them later in this subsection.

Before we have not talked about where the probability distribution comes from. We thought that it is given beforehand and we used it for the estimation of information content of the message. The *model* is that what supplies this probability distribution. The process of choosing of appropriate model for given input message is called *modelling* and the process which produces encoded sequence of bits is called *coding.* The modelling usually is of art and there is no "best" method.

In order to clarify the modelling process let us consider the simplest model of English language. The English alphabet contains 27 symbols. We do not care about the difference between upper-case and lower-case letters and also assume that there is only one punctuation symbol — space. So, in total we have 28 different letters. For regular English texts the probabilities of each symbol can be assigned not depending on texts. On other hand we can calculate the number of times each symbol occurred in the text and divide it to the length of input text. Such assigned probabilities exactly correspond to the input text, however for this modelling method pass through the text is needed. There are also methods which overcome the last drawback with preserving ability to fit probability distribution to the input.

The review of modelling schemes is not objective of this thesis and interested reader can be referred to wide variety of the books and articles devoted to this

topic. For example, see [BCW90, WMB99]. Nevertheless, we consider context-based modelling, in order to introduce the notion of kth order entropy, which is in use later.

The *context modelling* applied for the texts utilizes the dependencies between consecutive symbols. For example in English the letter 'u' is very probable after the letter 'q'. There are only few contra-examples. One of those is the word 'Iraq'. In general, context modelling scheme looks like as follows. Let us choose some positive integer $k > 0$ as a parameter of the modelling. Any k consecutive letters from the input text is treated as a *context.* For any context we can calculate the probability distribution of letters which follow after this context in the input text. At the end we have the set of these distributions for all possible contexts. We can say that there is distinct model of input for each particular context. Suppose that we have K different contexts C_i, $i = 1, 2, \ldots, K$ and we denote by H^i, $i = 1, 2, \ldots, K$ the entropy based on the corresponding probability distribution. The *kth order entropy* of the input text is given by

$$H_k = \sum_{i=1}^{K} p(C_i) H^i,$$

where $p(C_i)$ is the probability of context C_i. It is not so hard to derive the meaning of the last value from its definition. The kth order entropy is a lower bound on average number of bits, using for encoding symbols under context-based modelling scheme.

Let us leave the modelling and turn attention to the second part of any statistical coding routine. While the probabilities are derived using some model we should encode the input text and try to do it as close to theoretical lower bound as possible. We consider two coding methods which found wide use in practice.

The first method was invented by Huffman and is described in [Huf52]. The general idea of this method is to construct a binary tree with the leaves corresponding to the symbols in alphabet. The adjacent edges are labeled by 0 and 1. For example at each node we assign the 0 label for the edge leading to the left child and 1 for the edge leading to the right child. The code of particular symbol in alphabet is the concatenation of labels through the path leading from the root to the corresponding leaf. Suppose that the probability distribution of symbols is given. The method of construction of this tree in Huffman's algorithm involves the following rules.

- We start with the writing of the leaves of the tree with assigned probabilities of symbols.

- We combine two nodes, which were not in use yet in this step with the smallest values of probabilities into parent node and assign the probability of this node as a sum of probabilities of its children.

- We repeat the previous step while the root of tree is constructed.

The Huffman coding does not achieve the entropy lower bound. However, it can be shown that the redundancy of Huffman code measured as the average code length less the entropy is bounded by $p + \log_2(2(\log_2 e)/e) = p + 0.086$, where p is the probability of the most likely symbol [Gal78].

In general description of Huffman's method we have some freedom in choosing codewords for the symbols. It follows that the different sets of codewords can be generated by this approach for the same probability distribution. These sets of codewords can be thought as an equivalent ones and the freedom of choosing one particular representative means useless redundancy in the *codebook* (the array of entries for each of which the key is symbol and the value is the code for the symbol). The task of avoiding this redundancy was considered by Schwartz and Kallick [SK64]. The *canonical Huffman code* is the Huffman code which can be obtained by some procedure from usual Huffman code, with the property that two equivalent Huffman codes have the same canonical form. In application the codebook also should be transmitted to decoder and compression of it is often involved. The canonical Huffman codebook usually can be compressed better than non-canonical one. The procedure of obtaining of canonical form of Huffman codes is presented by the following steps.

- Sort the codebook by the length of codes in increasing order.
- The first symbol gets the code 00...0 with the same length as before.
- Each subsequent symbol gets the code as the next binary number.
- When the longer codeword is reached we increment the last codeword of shorter length and shift by one position to left. Then we continue with all binary numbers of new length.

The unique coding scheme which is optimal with respect to the entropy is arithmetic coding. Many researchers contributed to developing of it. But the first C implementation was published by Witten et al. [WNC87]. In this thesis I briefly describe the main idea which lies in the basis of this coding scheme and skip the implementation details. Interested reader can be referred to the book by Moffat and Turpin [MT02].

Suppose that we have n symbols from fixed alphabet with non-zero probabilities assigned $p_1, p_2, \ldots, p_n$. The *arithmetic coder* involves the interval of real numbers from 0 to 1. We mean that all real numbers within this interval are presented in binary form. We break this interval into subintervals without intersections. Each subinterval corresponds to one symbol from the alphabet and its length is equal to the probability of the symbol. For the first coming symbol we define the subinterval

which corresponds to it and send the common prefix of its bounds as an output. For the second symbol in turn we divide the subinterval corresponding to the first symbol and send again common prefix and so on. The special symbol 'END' is added to the alphabet and is encoded at the end of encoding process. This symbol is needed for decoding purpose, it gives a signal when we should finish decoding process.

To decode a text we start with the same interval of real numbers from 0 to 1. We divide this interval according frequencies as above. We read input until we uniquely define the subinterval, which was used at the encoding procedure. The bounds of this subinterval should have common prefix equal to the prefix of the encoded input and there is no subinterval which satisfies the same conditions. When the subinterval is defined we output symbol corresponding to subinterval and proceed with this subinterval, i.e. we divide it according frequencies and continue as above.

There are also other statistical coding schemes not presented above. But the two considered ones are most important and popular. They are applied in many modern standards.

2.1.3 Dictionary-based coding

Some methods which are widely in use in text compression exploit so-called *dictionary-based* approach. The compression in such methods is achieved by replacing some phrases (sequence of consecutive symbols) in the text by pointers to some dictionary. The phrases which are not in the dictionary remain in the text without transformation. If dictionary contains many frequent phrases, we can expect good performance.

The main issue of design of dictionary-based method concludes in choosing appropriate dictionary. For this task three approaches can be considered: *static, semi-adaptive* and *adaptive.* In the simplest static approach the same dictionary is used for any input text. The adaptive approach involves two stages. The first one is the passing the input text and constructing the optimal (in the sense of compression ratio) dictionary. The second stage is coding itself. The semi-adaptive approach can be thought as adaptive approach, but without two passes over the text. So, it adapts the dictionary to the input text, however only one pass over the text is made. The most famous semi-adaptive dictionary-based method is Ziv-Lempel coding. I talk about it and its variations later in this subsection.

When the dictionary is chosen, there are many different ways to encode the input text, because we can choose different phrases to be replaced by the indices in the dictionary. The task of splitting input text into the phrases which should be represented as indices is called *parsing.* The task of optimal parsing is proven to be NP-complete and there are many heuristic used in practice. The most popular of them is greedy heuristic. It suggests to replace the longest match from the dictionary

by index.

In the 1977 Jacob Ziv and Abraham Lempel published paper [ZL77] which describes a semi-adaptive dictionary coder. This work has a great significance, because nowadays almost all semi-adaptive dictionary coders exploit the idea presented in that paper. The family of such techniques is called Ziv-Lempel coding, abbreviated as LZ coding.

The main idea of Ziv-Lempel coding is to replace the next phrase by the pointer in the previous text, where it already occurred. One possible representation for the pointer is a pair (m, n), where m is the position in the text and n is the length of the phrase. The decoding is done in straightforward manner. The decoder just replaces all pointers by the phrases, which can be found in already decoded text.

Towards the implementation of this approach two main design decisions should be made. The first one answers the question how far from current position pointer can be targeted. The number of symbols back for the pointer can be limited or not. The second decision answers the question how long the targeted substring might be. The different variations on these decisions offer different coding schemes and, hence, different compression performance.

There are more than 10 alternatives for practical implementation of idea by Ziv and Lempel. We shortly consider two of them LZ77,LZ78.

In the LZ77 method the length of targeted substring is bounded by parameter F. Special sliding window of N symbols is in use. The $N - F$ first symbols of this window constitutes the symbols which are already encoded and where we search for the longest match. The last F symbols is the pattern for which we try to find maximal match in the preceding substring. The pointer in LZ77 method is coded by triplet. the first value is an offset of the longest match from the current position back. The second value is the length of the longest match and the last value is the first symbol in the pattern, which does not match. Using the symbol in the triplet guarantees us that if there is no match in the preceding substring, we can still present the next symbol as a pointer.

Another modification of idea by Ziv and Lempel is compression method LZ78. This method uses a dictionary of phrases. Every new coming phrase to the dictionary is a longest match within dictionary plus one extra symbol. After adding this phrase to the dictionary the coder outputs the pair of two values. The first value is the index in the dictionary, which matches the prefix of new phrase and the second value is the last symbol of the new phrase. If there is no match in dictionary the encoder outputs pair of zero index and next symbol. Let us demonstrate this method by example. Let the input be 'aaabbabaabaaabab', then the output of the

method is

Input :	a	aa	b	ba	baa	$baaa$	bab
Phrase number :	1	2	3	4	5	6	7
Output :	$(0, a)$	$(1, a)$	$(0, b)$	$(3, a)$	$(4, a)$	$(5, a)$	$(4, b)$

The advantage of the method is that one can use trie data structure for the efficient search in the dictionary. In practice the size of the dictionary can occur too big and we should not allow unlimited increase of memory for its use. The simple solution is just to clear the dictionary, when its size is greater than some predefined constant and continue the coding as starting on a new text.

2.1.4 Block sorting compression

The block sorting compression originates from the paper by Burrows and Wheeler [BW94]. The idea of the method is to transform the input text into another, however reversible form and apply some compression method on it. The compression is achieved if the transformation has a property that it enlarges the compressibility property of the text. The decoding process is broken into two steps. Firstly, we should decode the transformed text and then apply inverse transformation.

The transformation from the paper [BW94], nowadays, is known as Burrows-Wheeler transform, because of the inventors. For the given input string of length n the transformed string can be obtained as follows. For any symbol in string we consider its n preceding symbols, starting from the closest one. We assume that the string is cyclic. It means that the first preceding symbol of the first character in string is the last symbol in the string, second preceding symbol of it is the next to the last one and so on. We write these sequences for any symbol, which we call contexts from right to left. See example for the word 'mississippi' below.

`ississippi`	`m`	`sissippimi`	`s`	`s`
`ssissippim`	`i`	`ississippi`	`m`	`m`*
`sissippimi`	`s`	`sippimissi`	`s`	`s`
`issippimis`	`s`	`pimississi`	`p`	`p`
`ssippimiss`	`i`	`ssissippim`	`i`	`i`
`sippimissi`	`s`	`imississip`	`p`	`p`
`ippimissis`	`s`	`mississipp`	`i`	`i`
`ppimississ`	`i`	`issippimis`	`s`	`s`
`pimississi`	`p`	`ippimissis`	`s`	`s`
`imississip`	`p`	`ssippimiss`	`i`	`i`
`mississipp`	`i`	`ppimississ`	`i`	`i`

After construction of the contexts we sort them lexicographically thinking that the rightmost symbol is the most significant, the next to the right is the second of

significance and so on. The column from the right on example above is Burrows-Wheeler transformed string. The asterisk marks the first symbol in the original string.

It is obvious that if we can reconstruct the table in the middle in the previous example, we can get the original string, due to we have the first symbol to be marked. Thus, it is enough to know how to do it. The key observation for the table placed in the middle is that all rows as well as all columns have the same content, i.e. set of symbols. The last column is given by definition. The next to the last column is just its sorted representation. Further, let us suppose that we already reconstructed $i, i+1, \ldots, n$ columns of the table. The $(i-1)$th column can be obtained by the following procedure.

- Extract all different strings placed in $i, i+1, \ldots, n-1$ columns.
- For any particular string using the columns $i+1, i+2, \ldots, n$ define the symbols, which can precede this string.
- Sort these symbols and place them before rows, where extracted string occurs beginning from the small index of row.

The above described procedure is able to reconstruct the table and, hence, the original string. The last proves reversibility of the Burrows-Wheeler transform.

It is obvious that the Burrows-Wheeler transform is just permutation of original string. So, it leads that this transformation does not change 0th order entropy. However, due to that for regular texts there is dependency between consecutive symbols, for the given context some symbols are more likely to be occurred than other ones. The Burrows-Wheeler transform sorts the contexts lexicographically and, hence, we can expect that in the transformed string the same symbols are placed more closely to each other than in the original string. The last property than can be utilized by applying transformations like run-length coding and move to front coding, described in the beginning of this section. These transformations are expected to be achieving of increasing of compressibility of the text.

2.2 Compression of natural language

2.2.1 Model of English language

In this section we deal with compression of natural English texts. It is hard to define precisely what natural language is and no convenient theoretical model of language exists. However anyone can intuitively get idea of it and some regularities in language can be observed and utilized. English language can be broken into the letters and words. The existence of such pieces as words in languages is one of the

most significant features of almost all natural languages. However, contra-examples exist. In this subsection we examine the statistics of letters and words in English and also present some empirical law of natural language. The information content of English from the entropy point of view is also under our consideration. The statistics gathered for regular English can be useful in design of a coder, which exploits the simple static modelling. But, one should be warned about correctness of this statistics for any particular case. For example, despite of 'e' letter is the most probable letter in English, there is normal full-length book over 50,000 words, which does not contain this letter at all [Wri39].

The most frequent symbol in normal English text is space and the most frequent letter is 'e'. The average length of word is about 4.5 letters. If we do not distinguish upper-case and lower-case letters than the most frequent symbols are 'ETAOINSRHLD'. Initial letters of words are distributed differently the most frequent ones are 'TAOSHIWCBPFD'. From this we see that the probability of occurrence of certain letter depends on the position within the word. In English there is also correlation between consecutive letters. A collection of English texts known as Brown corpus often is in use for the studying of statistics of English language. The alphabet of this corpus consists of 94 symbols. In Table 2.1 we summarized the statistics for the first most frequent symbols gathered from Brown corpus. We use special symbol '•' for space character. In the table the statistics for the most frequent digrams and trigrams is also presented. The digram (trigram) is the group of 2 (3) consecutive letters.

In order to get the statistics for the words in English, some problems immediately should be solved. The most important one is how to define the word itself. The simplest definition is that the word is any sequence of non-space letters. However under this definition the sequences 'letter' and 'letter.' are different. So, the problem could conclude in the treatment of numbers, punctuation symbols and so on. Different approaches for resolving such problems give different statistics. But we can expect that these differences are not too much and we can sketch some average statistics of words. I present the statistics for the most frequent English words under the definition that the word is the longest sequence of letters, separated by spaces and multiple spaces as well as other punctuation symbols are ignored. The statistics presented in Table 2.2 was gathered from the same Brown corpus.

An American linguist and philologist George Zipf has observed interesting phenomenon of using words in human writing and speaking. In the work [Zip49] he has published an empirical law, nowadays known as Zipf's law. Let $w(n)$ be the number of words which occurs in corpus exactly n times. This value varies with n as $w(n) \sim 1/n^{\gamma}$, where γ is close to 2. There is also another formulation of this law. Suppose that the words are ranked according to their frequencies, i.e. the most frequent word has rank equal 1 ($r = 1$), for the second most frequent word $r = 2$ and so on. Then, for large ranks, the number $n(r)$ of occurrences of word with rank

Table 2.1: Letter statistic for Brown corpus

Letter	Prob. (%)	Diagram	Prob. (%)	Triagram	Prob. (%)
•	17.41	e•	3.05	•th	1.62
e	9.76	•t	2.40	the	1.36
t	7.01	th	2.03	he•	1.32
a	6.15	he	1.97	•of	0.63
o	5.90	•a	1.75	of•	0.60
i	5.51	s•	1.75	ed•	0.60
n	5.50	d•	1.56	•am	0.59
s	4.97	in	1.44	nd•	0.57
r	4.74	t•	1.38	and	0.55
h	4.15	n•	1.28	•in	0.51
l	3.19	er	1.26	ing	0.50
d	3.05	an	1.18	•to	0.50
c	2.30	•o	1.14	to•	0.46

Table 2.2: Word statistics for Brown corpus

Word	Prob. (%)	Diagram	Prob. (%)	Trigram	Prob. (%)
the	6.15	of the	0.95	one of the	0.03
of	3.54	in the	0.55	as well as	0.02
and	2.70	to the	0.33	the United States	0.02
to	2.51	on the	0.23	out of the	0.02
a	2.14	and the	0.21	some of the	0.02
in	1.90	for the	0.17	the end of	0.01
that	0.97	to be	0.16	the fact that	0.01
is	0.95	at the	0.15	part of the	0.01
was	0.94	with the	0.14	to be a	0.01
for	0.86	of a	0.14	of the United	0.01
with	0.68	that the	0.13	a number of	0.01
as	0.65	from the	0.13	end of the	0.01
he	0.65	by the	0.13	members of the	0.01

Table 2.3: The entropy of English by Shannon

0-model	1-model	2-model	word model
4.03	3.32	3.1	2.14

r is given by $n(r) \sim 1/r^z$, where z is about 1.

In his paper [Sha51] Shannon studied the task of estimating the entropy for regular English. He considered an alphabet of 26 letters plus the space. He got the results for 0,1 and 2 context-based models and also measured the entropy for the word model. These results are summarized in Table 2.3.

2.2.2 Tagged and plain Huffman codes

When we deal with the compression of natural language we can follow mainly two strategies. The first one is to use symbol-based model and to encode the text symbol by symbol. Another strategy is to utilize the fact that text consists of the words. In the last approach an alphabet appears as a set of words occurring in the text.

The work [dMNZBY00] presents the adaptation of the Huffman's method for the word-based alphabet. As it is mentioned above before applying any word-based technique we should precisely define what the word is. The simple approach is to think about the word as a sequence of alphanumeric consecutive symbols. The groups of other consecutive symbols form so-called separator alphabet. Thus, we use two distinct vocabularies (synonym of an alphabet for the word-based models) one for words and another one for separators. Notice that we do not care about separating the codewords from these vocabularies in an output due to that between any two words the separator is placed and vice versa. Additionally, we can use one bit to answer the question is the first codeword for word or separator.

The authors of paper [dMNZBY00] use a different method for dealing with words and separators. It is called *spaceless model.* Assume that we store the words and the separators in one vocabulary and we do not include the simple space there. If during encoding the next separator is space we output nothing and go on, otherwise we output the code for separator. A decoder for any next coming codeword checks whether it corresponds to word or separator. If two consecutive words occur the decoder inserts the space symbol between them. This practice usually gives better result due to that the most frequent separator in ordinary text is space symbol and we use no bits for encoding it.

For presenting the method of [dMNZBY00] we should give some definitions.

- *Binary Huffman code* is the usual Huffman code, where for each symbol in an alphabet the sequence of bits is assigned.

- *Byte Huffman code* is a Huffman code, where for each symbol the sequence of bytes is assigned, i.e. the length of codeword should be multiple to 8.

We consider two versions of Binary Huffman code.

- *Plain Huffman code* is Byte Huffman code, where all bits of each byte are used for the codewords.

- *Tagged Huffman code* is Byte Huffman code, where in each byte the most significant bit is reserved for special use. It is set to 1 in the byte which is the first byte in the sequence of bytes corresponding to the symbol. Other bytes have the most significant bit equal to 0.

The authors of [dMNZBY00] present the method, which uses the word alphabet under spaceless model, canonical Huffman codes of plain and tagged forms and vocabulary of words and separators is compressed by binary Huffman codes. In the paper they presented experimental results, which show that using bytes instead of bits in Huffman codes does not significantly decrease compression ratio. On the other hand, operations with bytes can be done faster, than with bits. As it is also shown in [dMNZBY00] these compression schemes allow to search words and phrases in compressed text without decompression, efficiently.

2.2.3 End-Tagged-Dense code

This compression technique was presented in the work [BINP03]. The End-Tagged-Dense code takes the advantage of simplicity over Tagged Huffman code with preserving its good properties. The idea is to use the most significant bit as a flag within byte which shows the end of the codeword instead of that it starts as in Tagged Huffman code. Namely, the most significant bit in the last byte of the codeword is set to 1 and the most significant bits of other bytes are set to 0. The important thing of this approach is that we can be ensured that the compression scheme satisfies prefix property not depending on what we do with remaining 7 bits in every byte. The last allow us to use any bit sequences of 7 bits within the byte for generation of the codewords. In order to achieve compression we are still going to assign shorter codewords for the most frequent words and longer codewords for the less frequent words. The last can be done straightforward not involving any tree data structure as in Huffman routine. The code assignment is done by the following steps.

- We sort the words in vocabulary by their frequencies in decreasing order.

- For the first 128 words we assign consecutive codes of length 7 and set the most significant bit within byte equal to 1.

- When all 7 bit codes are exhausted we continue with 2 bytes codes with the remaining two most significant bits of 2 bytes for special use as described above.
- When all 2 byte codes are exhausted we continue with 3 byte codes and so on.

We see that the phase of code assignment is extremely simple and can be done really fast, because it does not involve any complicated procedure as for example in Huffman method. The second advantage of this scheme is that there is no need to store the frequencies of the words and the codewords. At the compression stage we need only array of words sorted by frequencies, because the codewords can be calculated on the fly. On the other hand at the decompression stage we also need this array, only. Because from the codeword we can easily get the rank of the compressed word (the position in the array) and by using array obtain the word itself.

2.2.4 (s, c) - Dense code

The (s, c) - Dense code is a generalization of End-Tagged-Dense code firstly presented in the paper [RTT02]. In order to understand how the generalization is done let us take a look on End-Tagged-Dense code from a little another point of view. Any codeword formed by End-Tagged-Dense method is a sequence of bytes. All bytes except the last one has a value within range $[0, 2^7 - 1]$. The value of the last byte varies in the range $[2^7, 2^8 - 1]$. We call the former as a *continuer* and the latter as a *stopper*. The beginning bytes in the codeword can be seen as a sequences of values for numerical system with base 2^7 and the last byte as a some value which exceeds the base of this system or equal to it. The last property allows us to determine the end of the codeword. Now the generalization is straightforward. Choose two integer parameters s and c. The (s, c) stop-cont code is coding scheme which assigns to each word within vocabulary a unique code, which is represented as a sequence of less than c integers and ending by an integer within range $[c, c + s - 1]$. We see that the End-Tagged-Dense code is a (s, c) stop-cont code, where $s = 2^7$ and $c = 2^7$. Obviously, any (s, c) stop-cont code is a prefix code. The (s, c) - Dense code is (s, c) stop-cont code which is obtained by the similar procedure in the previous subsection. First we sort the words in the vocabulary by their frequencies in decreasing order. Then, we use all possible stoppers for most frequent words. For the next words in the vocabulary we should exhaust all possible combinations of one continuer and stopper, then two continuers and one stopper and so on. The (s, c) - Dense code has one significant property that the average length of it is minimal with respect to any other (s, c) stop-cont code [BINP03]. As it is seen the (s, c) - Dense code is a family of codes, because we are free in choosing parameters s and

c. The last can be utilized for achieving the minimal compression ratio for certain text. The algorithm on optimal choosing of these values is presented in [BINP03].

2.2.5 Word-based block sorting compression

The task of adaptation block sorting compression method for word alphabet was considered by Isal and Moffat [IM01]. They used spaceless model of the word alphabet. Recall that the block sorting compression involves the following stages. First we perform the Burrows-Wheeler transformation, then we apply move to front or run-length coding. Additionally, at the last stage some entropy-based coding scheme can be applied. Here we consider approach in which at the second stage move to front coding is used. The first question which should be resolved towards adaption to the word alphabet is how to deal with the Burrows-Wheeler transform in the case of word alphabet. The authors of work [IM01] suggest to present the input text as a sequence of indices in the word alphabet. Then the transform becomes applicable. The second drawback arises in the stage where move to front coding is involved. As it was described above this coding scheme uses the dynamic list of symbols from the alphabet and encodes any input symbol (word in our settings) as an index of it in the list. The size of word alphabet is usually huge and using linear search in the list becomes wasteful. The idea presented in [IM01] which allows to overcome it concludes in using splay trees for storing the list and performing needed operations on it. The *splay tree* is a self-balancing binary search tree with additional splay operation which moves the given node to the root. It was devised by Sleator and Tarjan [ST85]. It has all advantages of binary search tree. Particularly, the search of entry can be done efficiently. Additional splay operation allows to use this data structure in move to front coding. Every node of the tree stores the unique index which is outputted if the symbol in this node occur in the input. The root has the index equal 1. We do not go in further details on implementation of the idea of using splay trees, but just mention that for efficient implementation of it we care that the most frequent symbols should be closer to the root and has smaller indices.

2.3 Compression methods for the text of positive integers

2.3.1 Elias codes

Different methods are usually applied, when we deal with the text of positive integers. This type of the input appears e.g. in communication and some drawbacks can happen for the direct applications of the methods described above. One of the

drawbacks might conclude in absence of probability distribution for the positive integers. If application works in real time, we can not use semi-adaptive modelling approach in order to get this distribution. Moreover, in some applications we even can not construct an alphabet, because we do not know beforehand the range of positive integers in the input. We have knowledge that the input consists of positive integers, only. Different techniques were invented for such cases. For these techniques the property that the small numbers have shorter codewords and the large numbers have longer codewords usually holds. If the input mostly consists of small numbers, the compression is achieved.

In this subsection we deal with the family of Elias codes [Eli75]. There are three different coding schemes which are combined under the name Elias codes. They are Elias delta coding, Elias gamma coding and Elias omega coding. We start with gamma coding.

The *gamma code* for the given positive integer can be obtained by the following steps.

- Write the integer in binary form (it is assumed that the presentation starts from 1).
- Subtract 1 from the number of bits written in the previous step and write before that so many zeroes.

It is obvious that the decoding can be done easily. The code of an integer is its binary form with added zeroes in the beginning. The zeroes are needed for the accomplishing prefix property of the codewords. The length of the code for some integer x can be calculated as

$$2\lfloor \log_2(x) \rfloor + 1.$$

The *delta code* for the encoding uses gamma codes. The steps which are needed to be done to get delta code for positive integer are as follows.

- Write the integer in binary form
- Define the position N' of the most significant 1 bit starting with the number of position equal to 0.
- Encode the number $N = N' + 1$ using gamma codes.
- Append to the right remaining bits with the positions $N' - 1, N' - 2, \ldots 0$.

For the decoding we do the following.

- Read and count the zeroes from the left to the right, until the first one is reached. Let this count be L.

- Read $L+1$ next bits and calculate the value which this bit stream represents. Let this value be M.
- Read $M-1$ remaining bits to the value S.
- The integer is $2^{M-1}+S$.

The length of the code in bits for the positive integer x is given by

$$\lfloor \log_2(x) \rfloor + 2\lfloor \log_2(1+\lfloor \log_2(x) \rfloor) \rfloor + 1.$$

I do not describe the Elias omega code in this thesis. This code exploits the procedure similar to the above and does not have remarkable advantage. Interested reader can find the information in corresponding source [Eli75].

2.3.2 Fibonacci coding

The *Fibonacci coding* is a coding method, which uses well-known Fibonacci numbers. Recall the definition of those. The Fibonacci numbers $\{f_n\}_{n=1}^{\infty}$ are the positive integers defined recursively.

- Two first Fibonacci numbers are 1 and 1, i.e. $f_1 = 1$ and $f_2 = 1$.
- The next number is defined by two previous ones using that $f_n = f_{n-1} + f_{n-2}$.

Thus, the series of Fibonacci numbers begins as 1,1,2,3,5,8,13,21,.... The Fibonacci numbers defined recursively also have a closed-form solution, it is called Binet's formula and given by

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}, \tag{2.1}$$

where ϕ is golden ratio and is calculated as

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803.$$

The important fact about Fibonacci numbers concludes in so-called Zeckendorf's representation. This representation is provided by Zeckendorf's theorem which states that for any positive integer x there exists unique representation as

$$x = \sum_{i=0}^{k} f_{c_i},$$

where $c_i \geq c_{i-1} + 2$, for any $i \geq 1$. The last condition means that the sequence $\{f_{c_i}\}$ does not contain any two consecutive Fibonacci numbers. Moreover, the Zeckendorf's representation of integer can be found by greedy heuristic.

The Fibonacci coding of positive integers uses the Zeckendorf's representation of integer. The code for x is a bit stream of length $l(x)+1$ with positions indexed from the left to the right by $1, 2 \ldots$, where

$$l(x) = \max_{i \geq 1}\{i | f_i \leq x\}.$$

The last bit in the position $l(x)+1$ is set to 1. The value of ith bit is set to 1 if the Fibonacci number f_i occurred in Zeckendorf's representation and is set to 0, otherwise. Due to that the Zeckendorf's representation can be obtained by greedy algorithm it follows that the bit on $l(x)$th position is always set to 1. Hence, at the end of the codeword we have two consecutive ones. On the other hand two consecutive ones can not appear somewhere else within codeword, because of the definition of Zeckendorf's representation. The last allows us to distinguish the codewords for the separate symbols in the encoded sequence. More precisely, two consecutive ones at the end of each codeword make the Fibonacci codes having a prefix property.

Fibonacci dual theorem states even more than Zeckendorf's one. Namely, it states that in Zeckendorf's representation the first Fibonacci number never occurs in the representation. It follows that we can skip the first bit reserved for the first Fibonacci number and therefore we can make the codewords shorter. In spite of Zeckendorf's representation provides us longer codewords where the first bit always might be set to 0 it also could be useful for the separating of codewords within bit stream. By using that we never meet the portion of the bit stream containing more than two consecutive ones.

Chapter 3

Succinct representations of sequences

3.1 An overview of succinct data structures

One of the first works where the problem of data structure optimization was considered is [Jac89b]. The author states the problem of data optimization as follows. In classical data compression approach we can reduce the space occupied by certain data structure usually accompanied by reduction of accessibility of this data structure. We can not use compressed data structure before decompression. The last can be unacceptable for some applications. The research problem in [Jac89b] can be referred as the problem of devising algorithms, which achieve optimal tradeoffs between the size of data structure and accessibility of the data structure. The efficient representations of data structures as a result of the above algorithms are called *succinct data structures.* The accessibility of data structure can be thought in two respects. If the application does not change the data structure then we are only interested in obtaining the information from this data structure. For such applications the efficient representation is regarded as *static succinct data structure.* In [Jac89b] the author only deals with such. On the other hand, if the application can change the data structure, we should represent the data structure with allowing to perform these changes. In that case the efficient representation is called *dynamic succinct data structure.*

Jacobson defines two types of data optimization. The first one is *concrete* data optimization. We can think about data structure as a data type with certain set of already implemented operations on it. The idea is to modify the data with preserving implementations of operations so as to achieve smaller size. The main point here is that we do not change the scheme of representation for the data structure. For the clarification of this idea let's take the example of linked data

structure from [Jac89b]. A linked data structure consists of *nodes* . Each node occupies distinct block of memory and has several pointers to other nodes. Let us assume that some of the blocks of memory have the same content. Then we can save one copy of these blocks in the memory and update the pointers within data structure. It is clear that in some cases the reduction of the memory is achieved and we preserve functionality of the data structure.

The second type of data optimization, presented in [Jac89b] is called *abstract* data optimization. In this approach we assume that the specification of abstract data structure is fixed. The specification is meant as a set of abstract operations on the data structure which can be performed. We are free in implementations of these operations and using the last we try to choose optimal (in some sense) representation of the data structure.

The paper [Jac89b] by Jacobson has initiated a lot of research on succinct data structures. Many authors offered succinct representations for variety of classical data structures. In the rest of this section I present modern results which are achieved in this direction. I concern bit vectors, sequences, balanced parentheses, trees, planar graphs, permutations, dictionaries and suffix arrays.

Succinct representations of bit vectors and sequences are objectives of two next sections. The current results for these two data structures are presented there and here I start with succinct representations of balanced parentheses.

Balanced parentheses data structure is a string of $2n$ parentheses with the condition that the number of opening parentheses is equal to the number of closing parentheses and for any opening parenthesis it is possible to find corresponding opening one within string and vice versa. One of the motivations for the research of this data structure concludes in usefulness of this data structure for representation of XML documents. The queries on balanced parentheses, which are usually of interest, are the following:

- $FINDOPEN(x)$ ($FINDCLOSE(x)$) returns the index of the opening (closing) parenthesis for the given closing (opening) parenthesis x.
- $ENCLOSE(x)$ returns the index of the opening parenthesis of the pair, which is most nearly encloses the given parenthesis x.

Jacobson [Jac89a] offered a constant time solution for the above operations. His solution uses $O(n)$ bits for the storage. Later, Munro and Raman [MR01] gave alternative solution which occupies $2n + o(n)$ bits with the same circumstances. In the paper [GRRR06] much simpler solution is presented. The authors got the same space as in [MR01], but the lower order term is $O(n \log_2 \log_2 n / \log_2 n)$ versus $\Theta(n \log_2 \log_2 \log_2 n / \log_2 \log_2 n)$ in [MR01]. Additionally, they claim that the time complexity of their construction algorithm is smaller.

A tree is one of the fundamental data structure in computer science. We consider unlabeled and labeled trees. The latter is the same as the former with additional mapping, which assigns for each edge of tree some symbol from alphabet Σ.

In a seminal work [Jac89a] Jacobson observed that simple pointer-based solution for representation of static trees is wasteful in space and has offered another more efficient representation. In his paper Jacobson deals with unlabeled type of trees and considers the following navigational queries:

- $PARENT(u)$: returns the parent node of the node u.
- $CHILD(u, i)$: returns the i-th child of the node u.

For the last query it is assumed that all children of each node are ordered. Jacobson has given the solution which uses $2t + o(t)$ bits for the storage and answers the first and the second queries in $O(1)$ and $O(i)$ time, respectively. Here, t denotes the number of nodes in the tree. Later Munro and Raman [MR97] have extended the result by adding new query answered in constant time. This is $SUBTREE_SIZE(u)$ query, which returns the size of subtree rooted from given node. The main idea of succinct representation for ordered tree is to use isomorphism between trees and balanced parentheses. This isomorphism allows us to use the methods for the succinct representations of balanced parentheses for the succinct representations of trees.

A binary tree is a special kind of ordered tree with assumption that each node has at most two children. From the results on general trees it is easily seen that for the binary trees queries as $PARENT(u)$, $LEFT_CHILD(u)$, $RIGHT_CHILD(u)$ and $SUBSTREE_SIZE(u)$ can be done in constant time, using $2t + o(t)$ bits of storage.

Succinct data structures for the general case of labeled trees were considered in the paper [FLMM05]. Using the special *xbw* transform, inspired by Burrows-Wheeler transform for the strings, the authors achieved $2t \log_2 |\Sigma| + O(t)$ bits of space, where t is still the number of the nodes and Σ is the alphabet, where the labels drawn from. For this data structure the queries $PARENT(u)$, $CHILD(u, i)$ and $CHILD(u, \alpha)$ can be answered in $O(1)$ time. The last query takes the node u and the label α and returns the child node having the label α or some special value if such node does not exist. Additionally, to the above, the authors in [FLMM05] showed that their data structure can be used for subpath query. This query for the given sequence p of the labels from Σ and for the given node u returns all nodes such that there exists the path leading to u and concatenation of the labels of the nodes according to this path is equal to p. This query can be done in $O(|p| \log_2 |\Sigma|)$ time for any alphabet Σ and in $O(|p|)$ time if $|\Sigma| = O(polylog(t))$.

Considered ordered tree is a special type of more general data structure called planar graph. Before presenting the results on succinct representations of planar graphs, let me remind some notions.

For the finite set V of arbitrary objects, the *graph* is defined as a pair $G = (V, E)$, where $E \subset V \times V$. The element $v \in V$ is called *vertex* of the graph G and $e = (v_1, v_2) \in E$ is called *edge* connecting vertices v_1 and v_2. The *degree* of the vertex $v \in V$ is the number of edges connected to this vertex. In other words it is the number of elements in the set

$$\{(v, v')|v\prime \in V, (v, v') \in E\}$$

The graph is called *planar* graph if there is injection $I : V \to R^2$, so that it is possible to find a set of continuous lines $\{\ell_i(t)\}_1^{|E|}, t \in [0, 1]$ in the space R^2 without intersections satisfying:

1. The lines $\ell_i(\cdot)$ and $\ell_j(\cdot)$ do not have common points, except boundaries, i.e. $\ell_i(t) \neq \ell_j(t)$ $\forall t \in (0, 1)$ and $i \neq j$.
2. If $e_i = (v_1, v_2) \in E$, then $\ell_i(0) = I(v_1)$ and $\ell_i(1) = I(v_2)$, for any $i \in \{1, \ldots, |E|\}$.

For any finite graph it is possible to construct the mapping I and lines $\{\ell_i\}$, which give the presentation of the graph into the plane. However, the first condition can be satisfied for planar graphs, only. In the following considerations we are assuming that we deal with the graphs in the plane.

A *k-page book embedding* of a graph $G = (V, E)$ is a permutation of points from V and partition of E into k *pages.* The permutation defines the order for the vertices drawn in line on the planes (pages) and i-th element of partition defines edges drawn on i-th plane (page). It is needed that edges on page must not have intersections. The *page number* of the graph G is defined as a minimal number of pages in any book embedding this graph.

The succinct representations of graphs with bounded number of pages were considered by Jacobson [Jac89a] and some results were improved by Munro and Raman in [MR97]. The succinct representation of one-page graphs, i.e. graphs with page number equal to 1 employs the same idea as for trees. Namely, by setting an isomorphism between the one-page graph and balanced parentheses, it becomes possible to use methods used for parentheses. For the graphs with more than one page generalization is done straightforwardly. Each page of the book is represented separately.

The paper [MR97] states two results:

1. An one-page graph on n vertices and m edges can be represented using $2n + 2m + o(n + m)$ bits, in such a way that the adjacency of a pair of vertices, and the degree of any given vertex can be found in constant time, and neighbours of a vertex can be produced in time proportional to the degree of the vertex.

2. A k page graph on n vertices and m edges can represented using $2kn + 2m + o(nk+m)$ bits in such a way that adjacency between a pair of vertices and the degree of a vertex can be found in $O(k)$ time, and the neighbours of a vertex x can be listed out in $O(d(x) + k)$ time where $d(x)$ is the degree of the vertex x.

For the planar graphs the last results are directly applied, due to that any planar graph can be embedded in four pages book in linear time [Yan86].

The permutation π on the set $[n] = \{0, 1, \dots, n-1\}$ is fundamental in computer science. Formally, it is defined as bijection on $[n]$. For two permutations π_1 and π_2 on the set $[n]$ the superposition $\pi_1 \circ \pi_2$ is defined and also is permutation on $[n]$. Particularly, for any $k > 0$ the power of the permutation π is $\pi^k = \pi \circ \dots \pi$. Due to the bijection property we consider the inverse of permutation π and consider the power of permutation for any integer k. π^0 is defined to be identical function. Basically, when we are talking about efficient representation of permutation π, we mean a representation such that the permutation π^k can be calculated rapidly.

In the work [MRRR03] the authors offered two succinct data structures. The first one takes $(1 + \epsilon)n \log_2 n + O(1)$ bits of space and supports calculation of $\pi(i)$ in $O(1)$ time and $\pi^k(i)$ in $O(1/\epsilon)$ time, for any $\epsilon > 0$, any integer k and any $i = \{0, \dots, n-1\}$. The second data structure occupies $\lceil \log_2 n! \rceil + o(n)$ bits and allows to perform $\pi^k(i)$ query in $O(\log_2 n / \log_2 \log_2 n)$ time for any integer k and i from 0 up to $n - 1$.

The paper [Pag02] by Pagh states the problem of storing static dictionary as follows. We are given some finite universe U and its subset $S \subset U$. Let the cardinality numbers of these sets be m and n, respectively. A membership query on static dictionary is query which for any given element $s \in U$ answers on the question does this element belong to the subset S or not. We are interested to get response on the last query in constant time.

Due to the number of different subsets with fixed cardinality number n is equal to the number of complete combinations the value

$$B = \left\lceil \log_2 \binom{m}{n} \right\rceil$$

gives us the number of bits which is needed for representation the subset with n elements from the universe U. The work [Pag02] presents the data structure, answering the membership queries in constant time and the space occupied by this data structure is $B + O(\log_2 \log_2 m) + o(n)$.

Succinct representation of the subset of natural numbers $S \subset \{0, 1, \dots, m-1\}$ for some m is under development in [RRR02]. The authors consider the *Rank* and *Select* queries. The *Rank* query takes natural number i and returns the number of elements in S which are less than i if $x \in S$ and -1 otherwise. The *Select* query for

given natural i returns the i-th smallest number in S or -1 if such number does not exist. This problem is known as succinct representation of indexable dictionaries. The authors of the paper [RRR02] for this problem achieved the space of $B+o(n)+O(\log_2\log_2 m)$ bits with *Rank* and *Select* queries answered in $O(1)$.

The last result has two applications for tries and multisets. The *k-ary trie* is another name for *k-ary cardinal tree.*The k-ary cardinal tree is a tree each node of which has k positions with the labels $0, 1, \ldots, k-1$. These position can contain the edges to children. Usually, it is used for storing associative arrays where keys are strings. The lower bound on the space needed for storing the k-ary trie with n nodes is given by

$$C(n,k) = \left\lceil \log_2 \left(\frac{1}{kn+1} \binom{kn+1}{n} \right) \right\rceil .$$

The additional term in [RRR02] for supporting parent, i-th child and degree of node queries in constant time is $o(n+\log_2 k)$. The problem of succinct representation of indexable dictionaries also finds application in representation of multisets. A multiset of the universe $U = \{0, 1, \ldots, m-1\}$ is natural generalization of regular subset. It is the subset $S \subset U$ with function $f : S \rightarrow \{1, 2, \ldots\}$ whose value $f(s)$ on some element $s \in S$ is treated as multiplicity of this element. Let us make the assumption that $\sum_{s\in S} f(s) = n$. Any multiset with the last condition can be represented with $B(m, n+m)$ bits. The authors of [RRR02] describes the data structure, which uses $B(m, n+m) + o(n) + O(\log_2\log_2 m)$ bits and support the following queries.

- *Rankm(x)*: returns -1 if $x \notin S$ and $\sum_{s\in S, s<x} f(s)$ otherwise.
- *Selectm(x)*: for $x \in \{1, \ldots, n\}$ returns the largest element $s \in S$ such that $Rankm(s) \leq x-1$.
- $Rankm^{+}(x)$: for $x \in U$ returns $\sum_{s\in S, s<x} f(s)$.

All these queries can be done in constant time by using the data structure from [RRR02].

The last data structure under our consideration in this section is suffix array. This data structure finds their applications in text indexing and string matching problems.

We are given a text $T = T[1, \ldots, n]$ from an alphabet Σ of fixed size. We attach the special symbol # to the text, which shows us the end of the text. Let us think that this symbol is already included in the alphabet and original text. Each suffix of this text can be represented by the position in the text where it is started. It leads that there are n suffices for the text. The set of all suffices can be lexicographically ordered, with the assumption that the ending symbol is the smallest symbol within

alphabet. The result is called *suffix array* built on the text T. One of the most common queries on suffix array is *lookup(i)* operation, which for given i returns the pointer in original text T, where the i-th smallest suffix starts. We are interested in succinct representation of the obtained data structure with allowing to perform *lookup()* query, efficiently.

One of most recent results on the above problem is presented in [GV05]. The authors proposed two alternatives. The first one uses $(1 + \frac{1}{2}\log_2 \log_{|\Sigma|} n)n \log_2 |\Sigma| + O(n)$ bits of storage with $O(n \log_2 |\Sigma|)$ preprocessing time and supports $lookup(i)$ query in $O(\log_2 \log_{|\Sigma|} n)$ time. The second alternative is a data structure which occupies $(1 + \epsilon^{-1})n \log_2 |\Sigma| + o(n \log_2 |\Sigma|)$ bits and performs $lookup(i)$ query in $O(\log^{\epsilon}_{|\Sigma|} n)$ time, for any $0 < \epsilon < 1$. The creation time is also $O(n \log_2 |\Sigma|)$.

3.2 Succint data structures on bit vectors

3.2.1 Queries on bit vectors

In this section I consider bit vectors or bit streams. The *bit stream* is a sequence of zeroes and ones of finite length. For the bit vector $S = S_0 S_1 \ldots S_{n-1}$, $S_i \in \{0, 1\}$ the following queries are of interest.

- S_i : returns the bit in the position i.
- $Rank_b(S, i)$: gives the number of b bits up to position i.
- $Select_b(S, i)$: returns the position of ith occurrence of bit b in S.

The useful operations also might be $Prev_b(S, i)$ and $Next_b(S, i)$ which return the positions of the previous and the next bit b near position i, respectively. But due to that these operations as well as access query S_i can be expressed with constant number of *Rank* and *Select* queries, the research of succinct representations of bit vectors basically only deals with these operations.

3.2.2 Succinct data structures with explicit storage

Study of succinct representations of bit vectors was initiated by Jacobson for the purposes of succinct representations of trees [Jac89a]. He offered an auxiliary data structure of $o(n)$ bits, which allows to answer *Rank* query in constant time. Thus, the total size of data structure becomes $n + o(n)$ bits. The task of supporting *Select* query was also considered by Jacobson, but the result was not so good as for *Rank* query. Later, that was improved by Munro [Mun96] and Clark [Cla98]. They achieved $o(n)$ extra bits of space for supporting *Select* query in constant time.

In the rest of this subsection I present two data structures for *Rank* and *Select* queries which are the best current results. The first one supports *Rank* query and it is Jacobson's original solution. The second one was invented by Kim et al. [KNKP05] and it is more efficient alternative of Clark's solution for *Select* query.

To support *Rank* query let us first divide the input bit vector S into blocks of length $b = \lfloor \log_2(n)/2 \rfloor$. Then we group every $\lceil \log_2 n \rceil$ blocks into superblocks. Thus the length of superblock becomes $s = b\lceil \log_2 n \rceil$. For every superblock we precalculate values $R_s[j] = Rank_1(S, js)$, $0 \leq j \leq \lfloor n/s \rfloor$. For storing of these values we need $O(n/\log_2 n)$ bits. For each kth block belonging to jth superblock we store $R_b[k] = Rank_1(S, kb) - Rank_1(S, js)$. It requires $O(n \log_2 \log_2 n / \log_2 n)$ bits. At last for any bit vector B of length b and for position i within it we calculate the value $R_p[B, i] = Rank_1(B, i)$. The last takes $O(\sqrt{n} \log_2 n \log_2 \log_2 n)$ bits of space. In order to perform $Rank_1(S, i)$ query we proceed with the following steps:

1. We define the number of block
$$k = \lfloor i/b \rfloor.$$

2. We calculate the number of superblock
$$j = \lfloor k/\lceil \log_2 n \rceil \rfloor.$$

3. The answer is
$$R_s[j] + R_b[k] + R_p[S_{kb} \ldots S_{(k+1)b-1}, i \ mod \ b]$$

So far, *Rank* query for 1 bit can be performed in constant time. For calculation of *Rank* query for 0 bit we use the formula $Rank_0(S, i) = i + 1 - Rank_1(S, i)$. Obviously we have achieved $o(n)$ extra bits of space and constant time solution for both values of bit.

To calculate $Select_b(S, j)$ query the simplest solution which comes to mind is to use binary search in S with the structure for *Rank* query. Namely, the answer is such position i that $Rank_b(S, i) = j$ and $Rank_b(S, i-1) = j - 1$. But by this approach the *Select* query is calculated in $O(\log_2 n)$ time. The last algorithm can be improved using block structures. The idea is to first search within the superblocks, then in blocks which belong to defined superblock and at last within block.

Due to that there is no clear relation between the $Select_0(S, i)$ and $Select_1(S, i)$ queries we should consider them separetely. However, it is obvious that any data structure suitable for one query can be used for another one. It is sufficient to consider negation of bit stream and construct the data structure for opposite query on it.

The first constant time solution for *Select* query was presented by Clark [Cla98]. In the paper [KNKP05] the authors analyze the behavior of Clark's algorithm and find out drawback of it. The Clark's algorithm becomes worse when the number of ones in vector is fewer. They proposed two algorithms which overcome this drawback. The first algorithm achieves $o(n)$ bits of extra space for supporting *Select* query and the second one does the same with using $n + o(n)$ extra bits in worst case. Let us consider the first one.

First we have to introduce some definitions from [KNKP05]. Let us assume that the bit stream S of length n is given. We divide S into blocks of non-zero length b. If the block contains 0 bits only, we call it *zero-block,* otherwise *nonzero-block.* Then, we introduce three new bit streams built from S.

- The *contracted* bit stream is a bit stream of length $\lceil n/b \rceil$, each position of which corresponds to the block in S and the value in this position is set to 0 if the corresponding block is zero-block and 1 otherwise.
- The *extracted* bit stream S_e is defined as the result of concatenation of all nonzero-blocks in S, according to the order in S.
- The *delimiter* bit stream is a bit stream such that every ith entry is defined to be equal to 1 if the ith and $(i-1)$th 1 bits are in the same block and 0 otherwise. The first entry is equal to 1 by definition.

Let B and C be extracted and contracted bit streams of S, correspondingly obtained by division original stream into the blocks of length $\lceil \sqrt{\log_2 n} \rceil$. The idea presented in [KNKP05] is to construct data structures for supporting *Select* queries on B and C, and to find functional relation between these queries and the *Select* query on S.

The paper [KNKP05] states that this functional relation is

$$Select_1(S, i) = Select_1(B, i) + (Select_1(C, s_b) - s_b)\lceil \sqrt{\log_2 n} \rceil,$$

where

$$s_b = \left\lceil \frac{Select_1(B, i)}{\sqrt{\log_2 n}} \right\rceil.$$

Then authors of paper [KNKP05] suggest two data strutures for supporting constant time *Select* query on bit vectors B and C. The last leads to that constant time *Select* query on original bit vector is also supported.

First of all let us define *rank-look-up* and *select-look-up* tables which will in use later. These look-up tables contain the answers on *Rank* and *Select* queries for bit patterns of length $\lceil (\log_2 n)/c \rceil$ correspondingly, for some fixed integer $c > 1$ and thus enables us to compute these answers in constant time. Actually, we already used rank-look-up table when we dealt with *Rank* query above.

To support *Select* query on B we arrange two-level directory and both look-up tables. The first level of the directory contains the position in B of every $(\lceil \log_2^2 n \rceil)$'th 1 bit. It requires $O(n/\log_2 n)$ bits of space. The second level of the directory stores the position of every $(\lceil \sqrt{\log_2 n} \rceil)$'th 1 bit in the ranges of the first level of the directory. Notice that due to the definition of a bit stream B every block of $\lceil \sqrt{\log_2 n} \rceil$ length contains at least one 1 bit. Hence, the maximal value for the entry in second level directory is $\lceil \sqrt{\log_2 n} \rceil$. Thus, the space occupied by this level of the directory is $O\left(n/\sqrt{\log_2 n} \times \log_2(\log_2^2 n \sqrt{\log_2 n}\ )\right)$ bits.

Now we can perform $Select_1(B, i)$ query on B in constant time. First, we define the values $j_1 = \lfloor i/\lceil \log_2^2 n \rceil \rfloor$ and $j_2 = i/\lceil \sqrt{\log_2 n} \rceil$. Notice that due to the properties of B we have to search additionaly the block of $\sqrt{\log_2 n}\sqrt{\log_2 n}$ length at most. But it can be done by using select-look-up table in constant time. The answer is the sum of the values from the select-look-up table and the values in the first level and the second level of the directory with the indices j_1 and j_2, respectively.

To support *Select* query on C another approach is applied. First, we divide C into the blocks of length $\lceil \log_2 n \rceil$ and define for C and its division delimiter bit stream D. The length of it is $n/\sqrt{\log_2 n}$ in worst case. We attach to this bit stream auxiliary data structure for constant time *Rank* query. The value of $Rank_1(D, i)$ is the number of non-zero blocks up to the block containing ith 1 bit, including the last block itself. We define *mapping array* M. The ith entry of M corresponds to the number of ith non-zero block in C. The space occupancy of this array is given by $O(n/(\log_2 n \sqrt{\log_2 n}) \times \log_2(n/(\log_2 n \sqrt{\log_2 n}))$. In order to perform $Select_1(C, i)$ query we first calculate the number of block, which ith 1 bit belongs to by using M and data structure for *Rank* query on D and then by using select-look-up table the relative position of ith 1 bit in block.

3.2.3 Succinct representations within entropy bounds

So far we have talked about creation of data structures of the size $o(n)$, which can be attached to original bit stream for supporting *Rank* and *Select* queries. More ambitious goal is to achieve the constant time solutions for both queries storing bit stream in compressed form. In this subsection we deal with data structures, which answer *Rank* and *Select* queries and use in total $nH_0 + o(n)$ bits of space, where H_0 is 0-order entropy of bit stream.

Above we already considered the task of succinct representations of indexable dictionaries with supporting *Rank* and *Select* queries. It turns out that the last task is equal to the task of representations of bit streams. Indeed, let us fix the number $n > 0$ and consider all possible subsets of the universe $\{0, 1, \ldots, n-1\}$. Every such subset can be represented as a bit stream, where ith position in it corresponds to the ith element of the universe. It is set to 0 if the corresponding element does

not belong subset and otherwise if it is in the subset the bit in the ith position is set to 1. It is seen that this correspondence is a bijection between all subsets from the fixed universe $\{0, 1, \ldots, n-1\}$ and all bit streams of the length n. As it is described above, the authors in the work [RRR02] defined the *Rank* and *Select* queries on the subsets. These queries due to the bijection can be expressed in the terms of corresponding to the subset bit stream. Let us assume that the subset is $S \subset \{0, 1, \ldots, n\}$ and it corresponds to the bit vector $B' = B'_0, B'_1, \ldots B'_r$. The notation here differs a little from the one presented earlier. We use n as a cardinal number of the universe and r as a number of elements in the subset. We see that due to the defined bijection between S and B' the value of $Select(S, i)$ query on the subset S is equal to the value of $Select_1(B', i)$ query. Another query on the bit stream — $Select_0(B', i)$ can be expressed in the terms of complement subset $\bar{S} = U \backslash S$. Namely, $Select_0(B', i) = Select(\bar{S}, i)$. Thus, the task of supporting *Select* query on the bit stream is reduced to the task of supporting *Select* query on the subset and its complement. Similarly, we do with the *Rank* query. The $Rank_1(B', i)$ query is equal to the $Rank(S, i)$ query on the subset if $B'_i = 1$. If it is not so, i.e. $B'_i = 0$ we calculate $Rank(\bar{S}, i)$ and the answer is $i - Rank(\bar{S}, i)$. We do not need to talk about $Rank_0(B', i)$ because as it was mentioned above there is simple relation between $Rank_1(B', i)$ and $Rank_0(B', i)$ queries. Thus, we see that the task of succinct representation of indexable dictionaries and the task of succinct representation of the bit streams are equal for *Rank* and *Select* queries. Now, we should utilize the results of paper [RRR02] and treat them in the sense of bit streams. In the work [RRR02] the authors achieved $B + o(n)$ bits of space with both queries answered in constant time for the subset and its complement. They called such representation of dictionary as *full indexable dictionary.* The value B is given by

$$B = \left\lceil \log_2 \binom{n}{r} \right\rceil$$

It immediately follows that using the same space occupancy we can support the *Rank* and *Select* queries on bit streams and also access query. The value n in the formula for space is the length of the bit stream and the value r is the number of ones in this bit stream. Let's modify expression for the space.

$$B + o(n) = \lceil \log_2 \binom{n}{r} \rceil + o(n) = \log_2 \frac{n!}{r!(n-r)!} + o(n) =$$

$$= \log_2 n! - \log_2 r! - \log_2 (n-r)! + o(n).$$

Using the Stirling approximation for the factorial function

$$n! = n \ln n - n + o(n) = n \frac{1}{\log_2 e} \log_2 n - n + o(n)$$

we have

$$B + o(n) = \frac{1}{\log_2 e}(n \log_2 n - r \log_2 r - (n-r)\log_2(n-r)) + o(n) =$$

$$= \frac{1}{\log_2 e}(n \log_2 n \pm r \log_2 n - r \log_2 r - (n-r)\log_2(n-r)) + o(n) =$$

$$= \frac{1}{\log_2 e}\left(-r \log_2 \frac{r}{n} - (n-r)\log_2 \frac{n-r}{n}\right) + o(n) \leq nH_0(B') + o(n).$$

Thus, the bit stream is represented in compressed form within entropy bounds with supporting *Rank* and *Select* queries in $O(1)$ time.

3.2.4 Succinct representations using gap encoding

The idea of using *gap encoding* originates from the work [Sad03] by Sadakane. Let us sketch this idea. Every bit stream can be expressed as a sequence of non-negative integers. The first number in this sequence is the position of the first 1 bit in the bit stream. The second number is the distance (the length of the gap) between the first 1 bit and the second one. The third number is the distance between the second and third ones and so on. When the sequence of such numbers is created it can be encoded by some coding method. The *Rank* and *Select* queries can be performed on the sequence of the numbers instead of bit stream, due to that above described transformation is obviously reversible. Depending on the method which was used in encoding different estimations can be done on the size of compressed bit stream. Auxiliary data structures for supporting constant time solutions can be attached to the sequence of the numbers.

In the work [Sad03] Sadakane provided constant time access query using $nH_0 + o(n)$ bits of the space. That was the first work which uses the idea of gap encoding. It inspired many other studies on this idea. To name a few of them.

Grossi et al. showed [GGV04] that using their data structure it is possible to support *Rank* and *Select* queries in $O(\log_2 r)$ time, where r is the number of ones in the bit stream by using $o(nH_0)$ additional bits of space.

The recent result is presented in [MN06]. The authors achieved constant time solutions for *Rank* and *Select* queries. Their data structure occupies in total $\alpha r \log_2 \frac{n}{r} + O(r) + o(n)$ bits of space. The constant α depends on the coding procedure, which is used for encoding of the sequence of the numbers. As it is stated in the paper, for Elias δ-encoding this constant can be taken equal to 1.

3.3 Succinct representations of sequences

3.3.1 Queries on sequences

In this section we turn our attention to general case of sequences. Let Σ be an alphabet of fixed size σ. We consider the sequence $S = S_0 S_1 \ldots S_{n-1}$ with the symbols drawn from the alphabet Σ. The *Rank*, *Select* and S_q queries for sequences are the natural generalizations of the queries on the bit streams, i.e.

- $Rank_c(S, i)$ takes the symbol $c \in \Sigma$ and position i in S and returns the number of times symbol c appears in S up to position i.
- $Select_c(S, i)$ takes the symbol $c \in \Sigma$ and position i in S and returns the position in S of ith occurrence of the symbol c.
- S_q returns the symbol which occupies the position with index q.

The *Substr* query is a new query for the general case of sequences and we did not talk about something similar above.

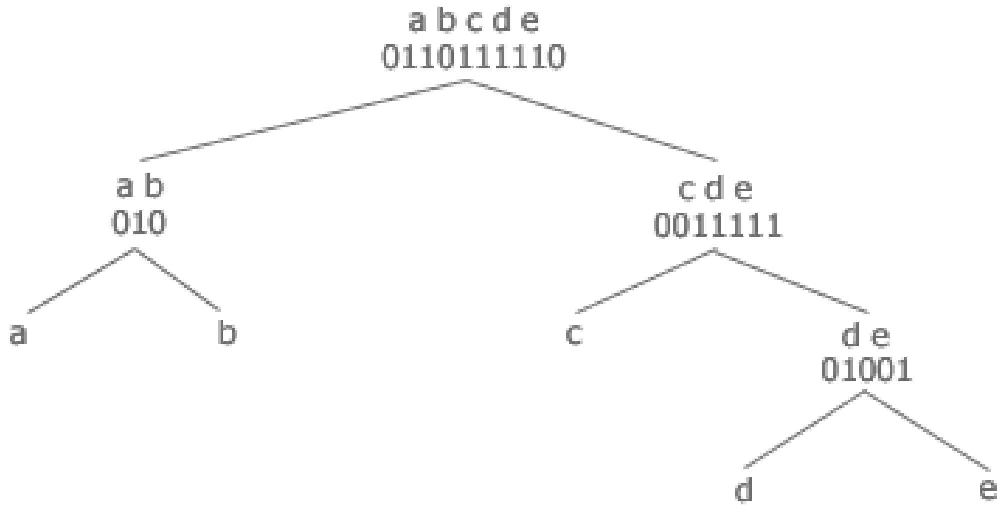
- $Substr_b(S, i)$ takes position i in S and returns substring with the length b starting at the position i. The value b can be fixed or depending on the length of S.

In this section we talk about efficient representation of sequences with supporting above described queries.

3.3.2 Succinct sequences for *Rank*, *Select* and S_q queries

An elegant solution for *Rank*, *Select* and S_q queries on sequences is presented in the paper [GGV03]. It is called *wavelet tree* and it allows to perform all above queries in $O(\log_2 \sigma)$ time, where σ is the size of the alphabet Σ. The wavelet tree is a balanced binary tree, where each node corresponds to some subalphabet of the alphabet Σ and each leaf corresponds to one particular symbol from the alphabet. The root corresponds to the alphabet Σ itself and the subalphabets of the child nodes are obtained by division of parent's subalphabet into two halves. Additionally, in the each node except the leaf ones a bit stream is stored. These bit streams depend upon the sequence for which the wavelet tree is constructed and upon the subalphabet of the node. The length of every bit stream is equal to the length of the subsequence of original sequence which only contains the symbols from the subalphabet of the node. The bit in the bit stream is set to 0 if the symbol on the corresponding position in the subsequence is in the subalphabet of left child. If the symbol belongs to the subalphabet of the right child the bit is set to 1. Let's take an example. Assume

Figure 3.1: Wavelet tree



that we the alphabet consists of the symbols 'a','b','c','d' and 'e' and the input sequence is 'accbdeddea'. The Figure 3.1 demonstrates the wavelet tree for this alphabet and sequence.

Now, let us discover how queries can be done using wavelet tree representation. First, notice that due to that the wavelet tree is balanced tree the depth of it is $O(\log_2 \sigma)$. To perform $Rank_c(S, i)$ we start from the root node and using attached bit stream firstly define the child node whose subalphabet contains the symbol c. Then we calculate *Rank* query for 0 or 1 bit (depending on the node determined in the previous step) and position i. The result can be treated as a position in the bit stream of the child node. For this position and for the symbol c we proceed as for root node. We firstly determine the child then perform corresponding *Rank* query and so on. This recursively defined procedure ends when the next child is a leaf node. If it is so then the result of the last *Rank* query on the bit stream is an answer on initial query. Notice that, if we assume that auxiliary bit streams of the nodes are presented succinctly with supporting *Rank* and access query in $O(1)$ time then the above procedure is done in $O(\log_2 \sigma)$ time, because the wavelet tree can be traversed in that time.

The $Select_c(S, i)$ query can be done using the wavelet tree in backward manner. We start from the leaf node corresponding to the symbol c. On the bit stream attached to the parent node we define the position which corresponds the position of the symbol of our interest. If the leaf node is a left child of its parent we define the position of ith 0 bit in parent bit stream, otherwise the position of ith 1 bit.

This is done by using *Select* query on the bit stream. The obtained position is used then for moving up in the same way. Firstly, we define is the current node left or right child of parent, then perform *Select* query for corresponding bit and the position. We finish when we reach the root node of the tree and the result of *Select* query on auxiliary bit stream of the root is an answer on initial query. As above we see that if *Select* query can be done on auxiliary bit streams in constant time, then the time which is needed for obtaining answer on $Select_c(S, i)$ query is $O(\log_2 \sigma)$. At the last we should understand how to perform access query for sequence using the wavelet tree. Actually, the procedure for it is very similar to one for the *Rank* query. We start with the root node and using access query for attached bit stream determine the value of the bit, then we define how many these bits occur up to position by binary *Rank* query and go to the child node. The result of binary *Rank* query is the position in the child's bit stream for which we do similarly as for parent node. We end when we reach the parent of some leaf, the value of the bit in current position within bit stream answers on the original question. It is easily seen that access query on sequence also takes $O(\log_2 \sigma)$ time, if we perform all queries on bit streams in constant time.

So far, we have shown that if all bit streams in the wavelet tree are presented succinctly with supporting all queries on bit streams in constant time, the *Rank*, *Select* and S_q queries can be done in $O(\log_2 \sigma)$ time. Next let's turn our attention to the space which is required for storing wavelet tree.

The structure of the wavelet tree only depends on the alphabet. If we assume the alphabet with the fixed size it leads that the structure does not depend on the input text. In such conditions for the estimation of the size of the wavelet tree we should get the size which is needed for storing attached bit streams, because the structure of the tree as well as the alphabet takes constant space for storing not depending on the length of the input. Let the frequencies of the symbols of the alphabet be $n_1, n_2, \ldots, n_\sigma$, i.e. the first symbol occurs n_1 times, the second one occurs n_2 times etc. We use the following notation.

$$\left(\begin{array}{c} k \\ k_1, k_2, \ldots, k_m \end{array} \right) = \frac{k!}{k_1! \cdot k_2! \cdot \ldots \cdot k_m!} \quad (m > 1)$$

$$\left(\begin{array}{c} k \\ n \end{array} \right) = \frac{n!}{k!(n-k)!}$$

Let us show that the number of different wavelet trees with the fixed structure which vary from each other in content of the bit streams is given by

$$\left(\begin{array}{c} n_1 + n_2 + \ldots + n_\sigma \\ n_1, n_2, \ldots, n_\sigma \end{array} \right).$$

When the depth of the tree is equal to zero we have only one symbol in the alphabet and hence there is only one wavelet tree. We see that the above formula gives us

also one. Next let the root of the wavelet tree has children. The left child contains the first $\lfloor \sigma/2 \rfloor$ symbols with the frequencies $n_1, n_2, \ldots n_{\lfloor \sigma/2 \rfloor}$ and the right child contains other symbols of the alphabet with the frequencies $n_{\lfloor \sigma/2 \rfloor+1}, n_{\lfloor \sigma/2 \rfloor+2}, \ldots, n_\sigma$. Notice that the number of different wavelet trees in this case can be obtained as multiplication of three terms. The first term is the number of different trees rooted from the left child. The second term is the same for the right child and the last term is the number of bit streams which can appear in the root node. Expressing this mathematically we have

$$\binom{n_1 + \ldots + n_{\lfloor \sigma/2 \rfloor}}{n_1, \ldots, n_{\lfloor \sigma/2 \rfloor}} \binom{n_{\lfloor \sigma/2 \rfloor+1} + \ldots + n_\sigma}{n_{\lfloor \sigma/2 \rfloor+1}, \ldots, n_\sigma} \binom{n_{\lfloor \sigma/2 \rfloor+1} + \ldots + n_\sigma}{n_1 + \ldots + n_\sigma} =$$

$$= \frac{(n_1 + \ldots + n_{\lfloor \sigma/2 \rfloor})!}{n_1! \cdot \ldots \cdot n_{\lfloor \sigma/2 \rfloor}!} \frac{(n_{\lfloor \sigma/2 \rfloor+1} + \ldots + n_\sigma)!}{n_{\lfloor \sigma/2 \rfloor+1}! \cdot \ldots \cdot n_\sigma!} \frac{(n_1 + \ldots + n_\sigma)!}{(n_{\lfloor \sigma/2 \rfloor+1} + \ldots + n_\sigma)!(n_1 + \ldots + n_{\lfloor \sigma/2 \rfloor})!}$$

$$= \frac{(n_1 + \ldots + n_\sigma)!}{n_1! \cdot \ldots \cdot n_\sigma!} = \binom{n_1 + \ldots + n_\sigma}{n_1, \ldots, n_\sigma}.$$

Thus, we have got the formula for the number of different wavelet trees. The number of bits which is needed for representation of particular tree is taken as a logarithm of the last value, i.e. the number of bits is given by

$$\log_2 \binom{n_1 + \ldots + n_\sigma}{n_1, \ldots, n_\sigma} = \log_2 \binom{n}{n_1, \ldots, n_\sigma} = \log_2 n - \sum_{i=1}^{\sigma} \log_2(n_i!) =$$

Using Stirling approximation we proceed

$$= \log_2 n - \sum_{i=1}^{\sigma} (n_i \log_2 n_i - n_i) + o(n) =$$

$$= n + \log_2 n \pm \sum_{i=1}^{\sigma} n_i \log_2 n - \sum_{i=1}^{\sigma} n_i \log_2 n_i + o(n) =$$

$$= n + \log_2 n - n \log_2 n - \sum_{i=1}^{\sigma} n_i \log_2 \frac{n_i}{n} + o(n) =$$

$$= nH_0(S) + n + \log_2 n - n \log_2 n + o(n) \leq nH_0(S) + \log_2 n + o(n) =$$

$$= nH_0(S) + o(n).$$

Thus, we see that the wavelet tree allows us to support *Rank*, *Select* and access queries in $O(\log_2 \sigma)$ time using the space close to the theoretical lower bound, i.e. entropy.

The recent result on the task of supporting queries on general sequences is presented in [FMMN06]. It improves the results of the wavelet tree. Namely, the authors achieved $nH_0(S) + o(n)$ bits of space with performing *Rank*, *Select* and S_q in constant time for the size of the alphabet $\sigma = O(polylog(n))$. For an arbitrary size of the alphabet the queries can be done in $O(\log_2 \sigma / \log_2 \log_2 n)$ time with using $nH_0(S) + o(n \log_2 \sigma)$ bits of space.

3.3.3 Succinct sequences for *Substr* query

The *Substr* query on general sequences was firstly considered in the paper [SG06]. In this work Sadakane and Grossi showed that the sequence $S = S[1, n]$ can be expressed using $nH_k(S) + O(\frac{n}{\log_\sigma n}(k \log_2 \sigma + \log_2 \log_2 n))$ bits of space, where $H_k(S)$ is kth order empirical entropy of the sequence S. On this data structure retrieving of any substring of length $\Theta(\log_\sigma n)$ can be done in constant time. The construction method of their data structure involves Ziv-Lempel coding. Recently, Gonzalez and Navarro got simpler solution [GN06] with the same space and time complexities, using arithmetic coding. In this subsection I shortly sketch this solution.

Let us define $b = \frac{1}{2} \log_\sigma n$ and $b' = \frac{1}{2} \log_2 n$. We divide original sequence S into the blocks of the length b. The construction procedure uses the following values depending on the index of the block $i = 0, 1, \ldots, \lfloor n/b \rfloor$.

- $S_i = S[bi + 1, b(i + 1)]$ is the ith block.
- $C_i = S[bi - k + 1, bi]$ is the kth order context of the ith block. For the first block we use some dummy values.
- P_i is arithmetically encoded ith block using k-order context modelling with initial context value C_i.
- μ_i is the length of P_i in bits.
- $\tilde{P}_i = \begin{cases} S_i, & \mu_i > b' \\ P_i, & \mu_i \leq b' \end{cases}$. Notice that the length of $\tilde{P}_i$ is at most b'.
- $\tilde{\mu}_i$ is the length of $\tilde{P}_i$.

For the representation of the sequence S we store the following information

- Bit vector $W[0, \lfloor n/b \rfloor]$ defined as

$$W[i] = \begin{cases} 0, & \mu_i > b' \\ 1, & \mu_i \leq b' \end{cases}$$

  The bit within this vector shows us whether the block is stored explicitly or not.

- Array of contexts $C[0, rank(W, \lfloor n/b \rfloor)]$. We store the contexts for the encoded blocks only.

- Bit vector $U = \tilde{P}_0 \dots \tilde{P}_{\lfloor n/b \rfloor}$ obtained by concatenation of all bit vectors $\tilde{P}_i$.

- Array of mappings $T[i] : \Sigma^k \times 2^i \to 2^b$, $i = 0, \dots, b'$. The mapping $T[i]$ for the context of the size k and for the bit sequence of the length i returns the first b symbols which is result of decoding of the second argument with initial context equal to the first argument. If the result of decoding is greater we truncate it to b symbols. If the length of the result is less than b we pad the result value with dummy values. Notice that we can not guarantee that any bit sequence with any initial context can be uniquely decodable. However, for such values of arguments the mapping is applied and we can skip these values during construction of the data structure.

- We store two tables $R_g[0, \lfloor n/(bc) \rfloor]$ and $R_l[0, \lfloor n/b \rfloor]$ for locating each element $\tilde{P}_i$ within U. We group every $c = \lceil \log_2 n \rceil$ blocks into superblocks. The first table answers the question where each superblock in U starts. The second table stores the relative positions of each block with respect to the superblock containing it.

To obtain subsequence of length b it is enough to get two blocks which contain pieces of this subsequence. Thus, we only describe the procedure of getting certain block. If we want to get the jth block, we firstly define the location and the length of the corresponding value $\tilde{P}_j$ in U using tables R_g and R_l. The vector W tells us is needed block encoded or stored explicitly. If the block is presented in explicit form we ends. Otherwise, we use the mapping $T[i]$ for the decoding it in constant time.

I should remark that the above presented solution differs a little from the solution in paper [GN06]. The only difference is in definition of mappings $T[i]$, the authors use mapping T and I described array of the mappings $T[i]$. The reason of such modification concludes in some aspects of implementation of arithmetic coder. In the chapter devoted to arithmetic coding I described this method in such way that it becomes impossible to implement exact solution by Gonzalez and Navarro. However, this modification does not break theoretical space complexity of the method and preserves the idea of it. The difference is negligible and I do not go in the details.

3.3.4 *Rank* and *Select* queries for large alphabets

As we see wavelet tree as well as the solution of paper [FMMN06] for arbitrary size of an alphabet allows to perform *Rank* and *Select* queries in time proportional to $\log_2 \sigma$, where σ is the size of the alphabet. In the paper [GMR06] the authors got faster queries with a little worse space occupancy. Namely, they proposed two data

structures. The first one supports *Rank* query in $O(\log_2 \log_2 \sigma)$ and *Select* query in $O(1)$ time and uses $nH_0 + O(n)$ bits of space. The second data structures stores the sequence in explicit form and uses in total $n \log_2 \sigma + o(n \log_2 \sigma)$ bits of space supporting *Select* query in $O(1)$ time and *Rank* and access queries in $O(\log_2 \log_2 \sigma)$ time.

Let us consider the first data structure presented in [GMR06]. The input sequence $S = S[1, n]$ can be expressed as a table T of zeroes and ones with σ rows and n columns. The entry $T[i, j]$ is set to 1 if the j-th element of S is the i-th symbol in the alphabet Σ, otherwise it is set to 0. We think about the alphabet as a range of positive integers $\{1, 2, \ldots, \sigma\}$. Let A be a bit vector obtained by writing the table T in row major order. We can establish the relations between *Rank* and *Select* queries on sequence S with the *Rank* and *Select* queries on bit vector A.

$$Rank_c(S, i) = Rank_1(A, (c-1)n + i) - Rank_1(A, (c-1)n),$$

$$Select_c(S, i) = Select_1(A, rank_1(A, (c-1)n) + i) - (c-1)n.$$

Thus, we reduce the task of supporting *Rank* and *Select* queries on S to the task of supporting binary queries on A. We talk further about the last task.

We divide the bit sequence A into blocks of length σ and introduce two queries with respect to this division. The first query *Rankb* is defined on the integers multiple to σ and the second one *Selectb* is defined for all values.

$$Rankb(i\sigma) = Rank_1(A, i\sigma),$$

$$Selectb(i) = \left\lfloor \frac{Select_1(A, i)}{\sigma} \right\rfloor.$$

Obviously, the value of $Rankb(i\sigma)$ query tells us the number of ones in the blocks up to the ith block and the value of $Selectb(i)$ query is the index of the block, where the ith one occurs.

For each block i we are able to define its cardinality as a value $k_i = Rankb(i\sigma) - Rankb((i-1)\sigma)$ and to construct vector B which is unary encoded sequence of cardinalities, i.e B starts with k_1 ones ended by zero, then k_2 ones with zero at the end and so on. *Rankb* and *Selectb* queries can be expressed in terms of binary *Rank* and *Select* queries on B. More precisely the following relations hold.

$$Rankb(i\sigma) = Rank_1(B, Select_0(B, i)),$$

$$Selectb(i) = Rank_0(B, Select_1(B, i)).$$

Due to that the length of A is equal to σn and it contains exactly n ones, the length of B is equal to $2n$. We can create the data structures on B for supporting constant time binary *Rank* and *Select* queries as described in subsection 3.2.2. Then, in

total we have $2n + o(n)$ bits of space for storing B with supporting constant time queries.

So far, we have created the data structure which is able in constant time perform *Rank* query on A for the indices multiple σ and define the number of block, which contains ith one. To accomplish our goal we next should support local *Rank* and *Select* queries for each block. These local queries for the ith block are defined as

$$Rank_1^i(j) = Rank_1(A, j + i\sigma) - Rank_1(A, i\sigma),$$

$$Select_1^i(j) = Select_1(A, j + Rank_1(A, (i-1)\sigma)).$$

Let A_i be the ith block and E_i is a sorted array of positions of ones in A_i. The operation $Select_1^i$ can be done in constant time using the array E_i. For supporting local *Rank* query special data structure is involved. It is called *y-fast trie* and it was introduced by Willard in [Wil83]. Let F_i be a set which contains every $\log_2 \sigma$th element of E_i. Storing the set F_i as y-fast trie we are able to support *Rank* query on this set in $O(\log_2 \log_2 \sigma)$ time. Due to that the sets E_i are ordered for finding local $Rank_1^i$ query we can use binary search within the range $[\log_2 \sigma Rank(F_i, j), \log_2 \sigma Rank(F_i, j) + \log_2 \sigma]$. It takes $O(\log_2 \log_2 \sigma)$ time. If the sets E_i are presented in compressed form as described in [RRR02], the total space of data structure becomes $nH_0 + O(n)$ bits.

Chapter 4

Compression methods for random access

4.1 Simple dense coding scheme with random access

4.1.1 Simple dense coding scheme

Let us assume that an alphabet Σ ($\sigma = |\Sigma|$) consists of symbols $s_0, s_1, \ldots, s_{\sigma-1}$. Then, for each symbol s_i from the alphabet the probability of appearance of the symbol in the input text is assigned and is equal to p_i. We can also assume that the alphabet is already ordered by probabilities and the first symbol has highest value of probability.

A coding scheme which is called simple dense coding scheme [FN07] assigns for the symbols the binary codes with different lengths in the following way. We assign '0' code for s_0 and '1' for s_1. Then we use all binary codes of length 2. In that way the symbols s_2, s_3, s_4, s_5 get the codes '00', '01', '10', '11', correspondingly. When all the codes with length 2 are exhausted we again increase length by 1 and assign codes of length 3 for the next symbols and so on until all symbols in alphabet get their codes. The table 4.1 demonstrates the coding scheme.

Lemma 1 *For the proposed coding scheme the following holds:*

1. *The binary code for the symbol $s_j \in \Sigma$ is of length $\lfloor \log_2(j+2) \rfloor$.*

2. *The code for the symbol $s_j \in \Sigma$ is a binary representation of the number $j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}$ of $\lfloor \log_2(j+2) \rfloor$ bits.*

Table 4.1: Coding scheme

Symbol	*Probability*	*Code*
s_0	p_0	0
s_1	p_1	1
s_2	p_2	00
s_3	p_3	01
s_4	p_4	10
s_5	p_5	11
s_6	p_6	000
s_7	p_7	001
s_8	p_8	010
s_9	p_9	011
s_{10}	p_{10}	100
s_{11}	p_{11}	101
s_{12}	p_{12}	110
s_{13}	p_{13}	111
s_{14}	p_{14}	0000
...	...	...

PROOF. Let a_n and b_n be indices of the first and the last symbol, which have the binary codes of length n. Then $a_1 = 0$ and $b_1 = 1$. The values a_n and b_n for $n > 1$ can be defined by reccurent formulas

$$a_n = b_{n-1} + 1, \qquad b_n = a_n + 2^n - 1. \tag{4.1}$$

In order to get the values a_n and b_n as functions of n, we firstly substitute the first formula in (4.1) to the second one and have

$$b_n = b_{n-1} + 2^n.$$

By applying the above formula many times we have a series

$$\begin{aligned}
&b_n = b_{n-2} + 2^{n-1} + 2^n,\\
&b_n = b_{n-3} + 2^{n-2} + 2^{n-1} + 2^n,\\
&\dots\\
&b_n = b_1 + 2^2 + 2^3 + \ldots + 2^n.
\end{aligned}$$

Finally, the value of b_n as a function of n becomes

$$b_n = 1 + \sum_{k=2}^{n} 2^k = \sum_{k=0}^{n} 2^k - 2 = 2^{n+1} - 3.$$

Using (4.1) we get

$$a_n = 2^n - 3 + 1 = 2^n - 2.$$

If j is given the length of the code for the symbol s_j is defined equal to n, satisfying

$$a_n \leq j \leq b_n.$$

Accordingly to the above explicit formulas for a_n and b_n we have

$$2^n - 2 \leq j \leq 2^{n+1} - 3.$$

It is equal to

$$2^n \leq j + 2 \leq 2^{n+1} - 1$$

and finally

$$n \leq \log_2(j+2) \leq \log_2(2^{n+1} - 1). \tag{4.2}$$

In accordance to the definitions of a_n and b_n values there exists unique solution of (4.2) for each $j \in \{0, \dots, \sigma - 1\}$. Therefore, the proposition will be proved if we show that $n^* = \lfloor \log_2(j+2) \rfloor$ satisfies (4.2).

The left inequality is obviously true. Let's consider the right one. A simple transformations give us

$$\log_2(j+2) \leq \log_2(2^{n^*+1} - 1),$$

$$j + 2 \leq 2^{\lfloor \log_2(j+2) \rfloor + 1} - 1,$$

$$\log_2(j+3) \leq \lfloor \log_2(j+2) \rfloor + 1.$$

For $j = 0$ the inequality is true. For $j > 0$ we are considering two cases. Firstly, let us assume that $\log_2(j+3)$ is integer, i.e. $\lfloor \log_2(j+3) \rfloor = \log_2(j+3)$. Due to $|\log_2(j+3) - \log_2(j+2)| \leq 1$ and $\log_2(j+2) < \log_2(j+3)$ for any $j \geq 0$ we have

$$\log_2(j+3) = \lfloor \log_2(j+2) \rfloor + 1.$$

For the second case, when the value $\log_2(j+3)$ has a fractional part, we get

$$\lfloor \log_2(j+2) \rfloor = \lfloor \log_2(j+3) \rfloor$$

and the inequality becomes

$$\log_2(j+3) \leq \lfloor \log_2(j+3) \rfloor + 1,$$

which is obviously true and the first statement of the proposition is proved. For the setting the second statement it is sufficient to observe that the code for the symbol $s_j \in \Sigma$ is $j - a_n$. By applying simple transformations we have

$$j - a_n = j - (2^n - 2) = j + 2 - 2^n = j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}.$$

So, the second statement is also proved. □

To encode the input sequence $S = s_0 s_1 \dots s_n$ taken from alphabet Σ we proceed with the following steps.

- Calculate the frequencies of the symbols in sequence S.
- Sort them in decreasing order according frequencies.
- Assign codewords for every symbol in the alphabet.
- Replace every symbol in S by its codeword and concatenate codewords.

Let $S' = s'_0 s'_1 \dots s'_{h-1}$ be an encoded sequence obtained from S, where $s'_i \in \{0, 1\}$. The length h can be calculated as

$$h = \sum_{j=0}^{\sigma-1} n_j \lfloor \log_2(j+2) \rfloor, \tag{4.3}$$

where n_j is the frequency of $(j+1)$th most probable symbol in the alphabet Σ.

4.1.2 Random access to compressed sequences

The fatal problem of the described coding scheme is that we can not distinguish the codewords within the encoded sequence S'. It can be overcame by attaching auxiliary bit vector D. This vector has the same length as sequence S' and bit in the ith position is set to 1 if in this position some codeword begins, otherwise the bit is set to 0.

Besides the ability to separate codewords in S' bit vector D also provides us tool for random access to encoded sequence. Indeed, we can easily extract the codeword of a certain symbol s_i from S, because we know that this codeword in S' starts at the same place where ith one in D occurs and the codeword ends before position where $(i+1)$th one in D appears. Moreover, if some data structure with supporting *Select* function in $O(1)$ time on D is created, above operations can be done in constant time. More formally, the codeword r of the ith symbol of S is given by

$$r = S'[Select_1(D, i) \dots Select_1(D, i+1) - 1].$$

Notice that due to $\sigma \leq n$ and the maximal length of the codeword is $\lfloor \log_2(\sigma + 2) \rfloor$ we have that $r = O(\log_2 n)$. It follows that under RAM model of computation extracting any particular symbol from the encoded sequence can be done in constant time.

It is easily seen that we do not need to store the vector D in explicit form. Only required functionality is supporting *Select* function. Hence, we can store this vector in compressed form with ability to calculate *Select* function on it.

4.1.3 Space complexity

In the previous subsection we calculated the size of the sequence S'. From compression point of view it is always interesting to get relation between the size of encoded sequence and theoretical measure of compressibility of sequence, that is entropy.

The zeroth order empirical entropy of the sequence S is given by

$$H_0(S) = -\sum_{j=0}^{\sigma-1} \frac{n_j}{n} \log_2 \left(\frac{n_j}{n}\right). \tag{4.4}$$

The value $nH_0(S)$ is theoretical lower bound on the size of encoded sequence. The following theorem gives upper bound on the value h.

Theorem 1 *For the values (4.3) and (4.4) the following inequality holds.*

$$h \leq nH_0(S) + n. \tag{4.5}$$

PROOF. Before proving the main inequality (4.5) we have to set auxiliary result:

$$\log_2 j \geq \lfloor \log_2(j+1) \rfloor - 1, \quad \forall j > 0. \tag{4.6}$$

The truth of the last is taken from series

$$\lfloor \log_2(j+1) \rfloor - 1 = \lfloor \log_2(j+1) - 1 \rfloor = \lfloor \log_2(j+1) - \log_2 2 \rfloor =$$

$$= \lfloor \log_2(\frac{j}{2} + \frac{1}{2}) \rfloor \leq \lfloor \log_2(\frac{j}{2} + \frac{j}{2}) \rfloor \leq \log_2 j.$$

Let us return to our main goal. We will follow the method of mathematical induction, where parameter of the induction will be the size of the alphabet Σ, i.e. σ. We have to prove that for any $\sigma \geq 1$ and $n, n_0, n_1 \ldots, n_{\sigma-1} \geq 1$, so that

$$n_0 + n_1 + \ldots n_{\sigma-1} = n,$$

and

$$n_0 \geq n_1 \geq \ldots \geq n_{\sigma-1},$$

the following holds

$$\sum_{j=0}^{\sigma-1} n_j \lfloor \log_2(j+2) \rfloor \leq -n \sum_{j=0} \frac{n_j}{n} \log_2 \left(\frac{n_j}{n}\right) + n$$

or

$$\sum_{j=0}^{\sigma-1} n_j \lfloor \log_2(j+2) \rfloor \leq -\sum_{j=0}^{\sigma-1} n_j \log_2 \left(\frac{n_j}{n}\right) + n. \tag{4.7}$$

For the $\sigma = 1$ we have $n_0 = n$ and it turns

$$n\lfloor \log_2 2 \rfloor \leq n \log_2 1 + n,$$

what is clearly true. Next, let's assume that inequality holds for σ, then we have to prove it for $\sigma + 1$. It becomes

$$\sum_{j=0}^{\sigma} n_j \lfloor \log_2 (j+2) \rfloor \leq -\sum_{j=0}^{\sigma} n_j \log_2 \left(\frac{n_j}{n} \right) + n.$$

We divide both parts of the last by $\sum_{k=0}^{\sigma-1} n_k$ and have

$$\sum_{j=0}^{\sigma} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \lfloor \log_2 (j+2) \rfloor \leq -\sum_{j=0}^{\sigma} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n_j}{n} \right) + \frac{n}{\sum_{k=0}^{\sigma-1} n_k}.$$

By adding and sumaltaneously substracting the value

$$\sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \frac{n}{\sum_{k=0}^{\sigma-1} n_k}$$

to the right side, we get

$$\sum_{j=0}^{\sigma} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \lfloor \log_2 (j+2) \rfloor \leq -\sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \left(\log_2 \left(\frac{n_j}{n} \right) + \log_2 \frac{n}{\sum_{k=0}^{\sigma-1} n_k} \right) +$$

$$+ \sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \frac{n}{\sum_{k=0}^{\sigma-1} n_k} + \frac{n}{\sum_{k=0}^{\sigma-1} n_k} - \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n_\sigma}{n} \right).$$

The last leads

$$\sum_{j=0}^{\sigma} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \lfloor \log_2 (j+2) \rfloor \leq -\sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \right) +$$

$$+ \sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \frac{n}{\sum_{k=0}^{\sigma-1} n_k} + \frac{n}{\sum_{k=0}^{\sigma-1} n_k} - \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n_\sigma}{n} \right).$$

Now, we move the first term of the right part of inequality to the left side and at the same time move the last term of the summation in the left part to the right and have

$$\sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \lfloor \log_2 (j+2) \rfloor + \sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \right) \leq$$

$$\leq \sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \frac{n}{\sum_{k=0}^{\sigma-1} n_k} + \frac{n}{\sum_{k=0}^{\sigma-1} n_k} -$$

$$-\frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_j} \lfloor \log_2(\sigma+2) \rfloor - \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n_\sigma}{n} \right).$$

By the assumption of induction we see that the left part of the inequality is not greater than 1. So, if we prove that the right part is not less than the same value we complete the proof. For this purpose let's consider the right side, separately.

$$\sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \frac{n}{\sum_{k=0}^{\sigma-1} n_k} + \frac{n}{\sum_{k=0}^{\sigma-1} n_k} -$$

$$-\frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_j} \lfloor \log_2(\sigma+2) \rfloor - \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n_\sigma}{n} \right).$$

Using that

$$n = \sum_{k=0}^{\sigma-1} n_k + n_\sigma,$$

we get

$$\sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(1 + \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \right) + 1 + \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} -$$

$$-\frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_j} \lfloor \log_2(\sigma+2) \rfloor + \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(\frac{n}{n_\sigma} \right).$$

It leads to

$$\sum_{j=0}^{\sigma-1} \frac{n_j}{\sum_{k=0}^{\sigma-1} n_k} \log_2 \left(1 + \frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \right) + 1 +$$

$$+\frac{n_\sigma}{\sum_{k=0}^{\sigma-1} n_k} \left[1 - \lfloor \log_2(\sigma+2) \rfloor + \log_2 \left(\frac{n}{n_\sigma} \right) \right].$$

Since n_σ is the smallest number among $\{n_j\}$ then minimal value of the fraction n/n_σ is $\sigma+1$ and due to (4.6) we have that the last term in brackets is non-negative and, hence, all expression is not less than 1. □

The bit vector D also has length h and, hence, we have shown

Theorem 2 *The sequence S can be encoded by simple dense coding scheme in $2n(H_0(S)+1)$ bits.*

Creating additional data structure for supporting *Select* query on D in constant time [Cla98] we have

Theorem 3 *$2n(H_0(S)+1)+o(n)$ bits is enough to represent S in compressed form with random access.*

As it was already mentioned we can compress auxiliary bit stream D with preserving ability to perform *Select* query. This is possible using $h' = hH_0(D) + o(n) + O(\log_2 \log_2 h)$ bits of space [RRR02]. Note that the entropy of the bit vector D can be easily calculated as

$$H_0(D) = -\frac{n}{h}\log_2\left(\frac{n}{h}\right) - \frac{h-n}{h}\log_2\left(\frac{h-n}{h}\right).$$

Thus,

Theorem 4 *The sequence S can be encoded in $h + h'$ bits and any symbol can be extracted from compressed sequence in constant time.*

4.1.4 Extensions

Above described compression procedure can be generalized in two ways. First, there is no problem for applying simple dense coding scheme using context based modelling. The difference is only using several tables with codewords for each context. So the result of the theorem 2 holds for the kth order entropy $H_k(S)$. But notice that we can not so easily organize random access for the encoded sequence, because we have to be able to determine context for every codeword we access.

Let's describe the second possible way of generalization. Consider that we use codewords of length divisible by some integer u. Above we assume that $u = 1$, because we assign codewords of any integer length. If $u = 2$ first we use the codewords of length 2, then 4, 6 and so on. Obviously, that for $u > 1$ the length of encoded sequence S' may only increase, but the key observation is that every bit in vector D which is placed in position non-divisible by u is equal to 0. Hence, we do not need to store bits in such positions. It follows that the length of the vector D decreases.

All results of the case $u = 1$ are easily generalized for arbitrary u as follows.

Lemma 2 *For the simple dense coding scheme with parameter $u > 1$ we have*

1. *The binary code for the symbol $s_j \in \Sigma$ is of length $\lfloor \log_2((2^u-1)j + 2^u) \rfloor_u$.*

2. *The code for the symbol $s_j \in \Sigma$ is a binary representation of the number $j + 1 - (2^{\lfloor \log_2((2^u-1)j+2^u \rfloor_u} - 1)/(2^u - 1)$ of $\lfloor \log_2((2^u-1)j + 2^u) \rfloor_u$ bits.*

Here, $\lfloor x \rfloor_u = \lfloor x/u \rfloor u$ is a nearest integer divisible by u which is not bigger than x.

The length of the vector S' is

$$h = \sum_{j=0}^{\sigma-1} n_j \lfloor \log_2((2^u - 1)j + 2^u) \rfloor_u.$$

Theorem 5 *The number of bits required by S' is at most $n(H_0(S) + u)$.*

The space required by D is then at most

$$\frac{n(H_0(S) + u)}{u}$$

bits. Then

Theorem 6 *The sequence S can be encoded in $n(H_0(S) + u)(1 + 1/u)$ bits.*

We value of u which minimizes the last value in the theorem is given by

$$u = \sqrt{H_0(S)}.$$

Special data structures for supporting *Select* query on D can be created and random access to compressed text can be organized as it was done above for the case $u = 1$.

So far, we assume that we deal with the sequence of symbols S taken from fixed alphabet Σ. Nevertheless, there is no problem to apply our coding scheme for the coding of sequence of positive integers without known beforehand probability distribution function. Simple dense coding scheme assigns for any positive integer p the codeword with the length of $\lfloor \log_2(p + 2) \rfloor$ bits like for the case of sequence of symbols. The codewords within S' is separated using auxiliary bit vector D. For this case we do need to store any tables consisting of the codewords, because everything can be calculated during coding procedure.

4.2 Fibonacci coding of sequences with random access

4.2.1 Fibonacci coding applied for sequences of symbols

In the subsection 2.3.2 we discussed the Fibonacci coding as a coding scheme which is applied for the sequence of arbitrary large positive integers. Before discussion of applicability of this scheme for the sequence of symbols, let us examine the code length of particular integer x, obtained by Fibonacci coding procedure. It is assumed that we use representation provided us by Fibonacci dual theorem. Obviously, the worst case for the code length is when the value x is Fibonacci number itself. Then

the code is sequence of $l(x) - 2$ zeroes ending with two ones. Hence, the length of the codeword for x is precisely $l(x)$. Thus, we have to estimate the value $l(x)$, supposing that x is Fibonacci number. Using the Binet's formula (2.1), we have

$$x = \frac{\phi^{l(x)} - (1-\phi)^{l(x)}}{\sqrt{5}}.$$

Taking logarithms from the both sides we get

$$\log_2(\sqrt{5}x) = \log_2\left(\phi^{l(x)} - (1-\phi)^{l(x)}\right),$$

$$\log_2(\sqrt{5}x) = \log_2\left[\phi^{l(x)}\left(1 - \left(\frac{1-\phi}{\phi}\right)^{l(x)}\right)\right],$$

$$\log_2(\sqrt{5}x) = l(x)\log_2\phi + \log_2\left(1 + \left(\frac{\phi-1}{\phi}\right)^{l(x)}\right).$$

Due to that $0 < (\phi - 1)/\phi < 1$, we have

$$\log_2(\sqrt{5}x) \geq l(x)\log_2\phi.$$

Hence, because of $\phi > 1$

$$l(x) \leq \frac{\log_2(\sqrt{5}x)}{\log_2\phi}.$$

We have shown

Lemma 3 *The length $|c(x)|$ of the Fibonacci codeword $c(x)$ for the positive integer x satisfies*

$$|c(x)| \leq \frac{\log_2(\sqrt{5}x)}{\log_2\phi}. \tag{4.8}$$

So far we considered the Fibonacci coding for the sequence of positive integers, however any coding scheme suitable for the last task can be used for the coding of sequence of symbols by the following way.

Let us assume that we have a sequence $S = S[1, n]$ with the symbols from the alphabet Σ ($|\Sigma| = \sigma$). The statistics of the sequence S is presented as a sequence of decreasing integers $n_1, n_2, \ldots, n_\sigma$. The value n_1 is the frequency of the most probable symbol in Σ, n_2 is of the second most probable one and so on. We can express the sequence S as a sequence S' of integers in the range $[1, \sigma]$ by substituting the symbol with the index of its frequency in the sorted list $n_1, n_2, \ldots, n_\sigma$. Obviously, we only need the list of symbols from the alphabet Σ sorted by the frequencies for performing the inverse transformation $S' \to S$. For the sequence S' we can apply any coding scheme for the positive integers. Particularly, it might be Fibonacci coding. The

important thing is that it is possible to give the estimation of average code length obtained by the way above based on the entropy of the source $H_0(S)$.

The average code length is given by

$$\bar{n} = \sum_{i=1}^{\sigma} \frac{n_i}{n} l(i),$$

where the value $l(i)$ is defined in subsection 2.3.2. Using the estimation (4.8) we have

$$\bar{n} \leq \sum_{i=1}^{\sigma} \frac{n_i}{n} \frac{\log_2(\sqrt{5}i)}{\log_2 \phi} = \frac{1}{\log_2 \phi} \sum_{i=1}^{\sigma} \frac{n_i}{n} \left(\log_2 i + \log_2 \sqrt{5} \right). \tag{4.9}$$

It is not hard to show that for any $i = 1, 2, \ldots \sigma$ the following holds

$$i \leq \frac{n}{n_i}. \tag{4.10}$$

Indeed, due to that $n_1 \geq n_2 \geq \ldots n_\sigma$ we have

$$n_1 + n_2 + \ldots + n_i \leq n \Rightarrow in_i \leq n \Rightarrow i \leq \frac{n}{n_i}.$$

Applying (4.10) to (4.9) we proceed

$$\bar{n} \leq \frac{1}{\log_2 \phi} \sum_{i=1}^{\sigma} \frac{n_i}{n} \left(\log_2 \frac{n}{n_i} + \log_2 \sqrt{5} \right) = \frac{H_0(S) + \log_2 \sqrt{5}}{\log_2 \phi}.$$

Thus, we have proved

Theorem 7 *The sequence S using Fibonacci coding can be represented in*

$$n \frac{H_0(S) + \log_2 \sqrt{5}}{\log_2 \phi}$$

bits.

4.2.2 Data structure for random access

As it was mentioned in the subsection 2.3.2 we have one attractive property of the Fibonacci code. The two consecutive ones can only appear at the end of the codeword and nowhere else. In this subsection I utilize this property for arranging random access.

First notice, if we want to start decoding from the ith symbol we should find the $(i-1)$th pair of two ones, supposing that the pairs are not overlapped. When the

position j of this pair is defined we start decoding from the position $j+2$. Thus, for our task it is enough to be able to determine the position of $(i-1)$th pair of ones in constant time. The query which does it we denote as $Select_{11}$. Notice, that due to that we do not allow the pairs to be overlapped the last query does not answer the question where the certain occurrence of substring 11 starts in the bit stream and it differs from extended *Select* query presented in [MN06]. The objective of this subsection is to present data structure which supports new query in constant time using amount of space asymptotically less than the size of bit stream itself.

The data structure for the $Select_{11}$ query can be constructed using the same idea as for classical *Select* query solution presented in [Cla98]. Thus, the data structure of this subsection can be seen as an adaptation of Clark's data structure for the $Select_{11}$ query.

Let $B = B[1, m]$ be a result of Fibonacci coding procedure for the original sequence $S = S[1, n]$ obtained as it was described in the previous subsection. To support $Select_{11}(B, i)$ query we do the following

- Record the positions of every $(\log_2 m \log_2 \log_2 m)$th non-overlapping 11 sequence in table $R_1 = R_1[0, \dots, \lfloor m/ \log_2 m \log_2 \log_2 m \rfloor]$.
- Let $r(i) = R_1[i] - R_1[i-1]$, $i = 1, 2, \dots \lfloor m/ \log_2 m \log_2 \log_2 m \rfloor$ be the length of range between consecutive 11 sequences in B.
- For the range with $r(i) \geq (\log_2 m \log_2 \log_2 m)^2$ we explicitly store the positions of all 11 in the table R_2.
- If the ith range has a length $r(i) \leq (\log_2 m \log_2 \log_2 m)^2$ we store in the table R_3 the relative positions of every $(\log_2 r(i) \log_2 \log_2 m)$th occurrence of sequence 11 in the range.
- Let $r'(i)$ be a length of range between consecutive entries in the table R_3. If $r'(i) \geq \log_2 r'(i)(\log_2 \log_2 n)^2$ we store all answers explicitly.
- For the range $r'(i) < \log_2 r'(i) \log_2 \log_2 n$, Clark showed that using rank-look-up and second-look-up tables with the size of every entry $16(\log_2 \log_2 n)^4$ it is possible to answer relative *Select* query.

The important remark which makes the Clark's approach applicable is that during construction every sequence 11 of interest is included entirely in range. It allows us to use look-up tables, because the situation when the sequence 11 belongs two ranges at the same time impossible.

The space bound $m + o(m)$ directly follows from Clark's result for $Select_1$ query. Indeed, let us imagine the transformation of the original bit stream $B \rightarrow B'$. This transformation substitute every pair of our interest by single one ending by zero

and other ones are substituted by zeroes. The B' bit stream has a length m and all data structures remain the same in the sense of occupied space. It proves that we have organized data structure for supporting $Select_{11}$ query in $o(m)$ bits.

Due to that m is a linear function of n as shown in Theorem 7 we have proven.

Theorem 8 *The sequence S can be encoded in*

$$n\frac{H_0(S) + \log_2 \sqrt{5}}{\log_2 \phi} + o(n)$$

bits and for any $q > 0$ the position in the encoded sequence from which the decoding of $S[q, n]$ may be started can be defined in constant time.

At the end of this subsection I bring adaption of **darray** solution for *Select* query by Okanohara and Sadakane [OS07]. This practical simple solution will be used in experiments.

We partition the bit vector B into the blocks with L non-overlapping consecutive ones. The array $P[0, \ldots, m/L - 1]$ is constructed to answer the query $Select_{11}(iL)$. If the size $P[i] - P[i-1]$ is larger than L_2 we store all positions of consecutive ones explicitly. If the length of the block is smaller than L_2 we store relative positions of every L_3th pair of ones. To perform *Select* query we first should define the block, then check is its size bigger or smaller than L_2. If it is bigger we obtain stored value, otherwise we lookup correspondent L_3th value and perform sequential search. If we choose $L = O(\log_2^2 m), L_2 = O(\log_2^4 m), L_3 = O(\log_2 m)$ described data structure will take $o(m)$ extra space. Notice that in fact we did not construct constant time solution for *Select* query, because the time of the query is $O(\log_2^4 n / \log_2 m)$. Anyway in practice this solution is quite fast.

Chapter 5

Experimental results

5.1 Test files

In this chapter I present the results of experiments, which were done with different test files. I experimented with simple dense coding and Fibonacci coding. All results are presented for word based models of the file. It means that we take the words of the file as symbols of the alphabet. Experiments for letter based alphabets were also done, but the compression performance were too poor to be competitive with modern compression techniques.

All experiments were done on Intel Celeron 1.5 MHz, 512 RAM, running Linux. I used C language and gcc 4.1.1 compiler with full optimization.

I used Silesia[1] and Canterbury[2] corpuses to obtain files for the experiments. The properties of the files are summarized in the Table 5.1.

Table 5.1: Test files

Name	Content	Type	Size
dickens	Collected works by Charles Dickens	English text	10 192 446 B
world192.txt	Country information around the world	English text	2 473 400 B
samba	Source code of Samba	C source	6 760 204 B
xml	Collected xml files	xml source	5 303 867 B

In compression algorithms different ways of dealing with separators and words are involved. The word is meant as any sequence of alphanumeric characters. Other sequences are treated as separators. The following tables characterizes the proper-

[1]http://www-zo.iinf.polsl.gliwice.pl/ sdeor/corpus.htm
[2]http://corpus.canterbury.ac.nz

ties of test files from the point of view, how many words and separators the files contain.

Table 5.2 shows the number of distinct words in the file, total number of words and entropy of input consisting of the words only.

Table 5.2: Words

Name	Size of dictionary	Total number	Entropy
dickens	34381	1819394	9.92
world192.txt	22917	343139	10.91
samba	29822	924640	10.40
xml	19582	847806	9.10

Table 5.3 shows the number of distinct separators, the total number of separators in the file and also entropy of the input consisting of separators only.

Table 5.3: Separators

Name	Size of dictionary	Total number	Entropy
dickens	1071	1819395	1.84
world192.txt	498	343140	3.17
samba	15544	924641	5.60
xml	1495	847807	4.13

The spless model involves one dictionary for storing words and separators. The simple space is not included. Table 5.4 shows the number of distinct entries, total number of entries in such constructed dictionary and also the entropy of the input under spless model.

5.2 Compression performance

Table 5.5 presents the sizes of compressed files and compression ratios for different methods. gzip -9 is an implementation of Ziv-Lempel compression method. bzip -9 is a compression method based on Burrows-Wheeler transformation. sdc and fibc are simple dense compression and Fibonacci coding, correspondingly. They are supposed to compete with other methods. Word Huffman is an usual statistical Huffman method, where statistics is gathered for words. ETDC is End-Tagged-Dense-Code presented in [dMNZBY00]. I have already talked about this method

Table 5.4: Words and separators under spless model

Name	Size of dictionary	Total number	Entropy
dickens	35451	2274883	9.47
world192.txt	23414	504104	9.81
samba	45365	1624292	9.63
xml	21076	1536221	7.90

above in this thesis. (200,56)-DC is (s, c) dense code which was also already considered. The parameters for the method were chosen empirically. For the last three methods two distinct dictionaries for words and separators were in use.

Table 5.5: Sizes of compressed files and compression ratios

File	gzip -9	bzip2 -9	sdc	fibc
dickens	3 851 823 B (37.7%)	2 799 528 B (27.4%)	3 598 358 B (35.3%)	3 213 794 B (31.5%)
world192.txt	721 413 B (29.1%)	489 583 B (19.7%)	879 488 B (35.5%)	783 329 B (31.6%)
samba	1 361 230 B (20.1%)	1 104 954 B (16.3%)	2 444 843 B (36.1%)	2 177 209 B (32.2%)
xml	653 743 B (12.3%)	427 238 B (8.0%)	1 754 626 B (33.0%)	1 624 502 B (30.6%)

File	Word Huffman	ETDC	(200,56)-DC	WinRar
dickens	2 889 403 B (28.3%)	3 356 527 B (32.9%)	3 257 899 B (31.9%)	2 395 134 B (23.4%)
world192.txt	718 110 B (29.0%)	850 966 B (34.4%)	828 958 B (33.5%)	398 642 B (16.1%)
samba	2 050 132 B (30.3%)	2 601 754 B (38.4%)	2 522 379 B (37.3%)	929 055 B (13.7%)
xml	1 522 683 B (28.7%)	2 048 357 B (38.6%)	1 960 694 B (36.9%)	406 356 B (7.6%)

Tables 5.6 and 5.7 show the compression performance for the test files, when the different approaches to deal with the words and separators are in use. The first approach concludes in maintaining two vocabularies for words and separators, it is noted as 'ws'. For the second approach marked as 'w' we ignore the separators in input file and only store the words assuming that all separators are replaced by space. This leads that compression methods become lossy. The last approach involves spaceless model of input and we note it as 'spless'.

The simple dense compression method involves special parameter u. Above the results for the value $u = 1$ were presented. Table 5.8 shows the sizes of test files and compression ratios for the different values of this parameter and I made experiments when words in the compressed files are only stored.

Table 5.6: Compression results for sdc method

File	ws	w	spless
dickens	3 598 358 B (35.3%)	2 991 239 B (29.3%)	3 487 314 B (34.2%)
world192.txt	879 488 B (35.5%)	726 852 B (29.3%)	882 586 B (35.6%)
samba	2 444 843 B (36.1%)	1 685 773 B (24.9%)	2 644 135 B (39.1%)
xml	1 754 626 B (33.0%)	1 301 877 B (24.5%)	1 877 722 B (35.4%)

Table 5.7: Compression results for fibc method

File	ws	w	spless
dickens	3 213 794 B (31.5%)	2 591 842 B (25.4%)	3 039 437 B (29.8%)
world192.txt	783 329 B (31.6%)	628 147 B (25.3%)	772 283 B (31.2%)
samba	2 177 209 B (32.2%)	1 451 816 B (21.4%)	2 288 745 B (33.8%)
xml	1 624 502 B (30.6%)	1 150 137 B (21.6%)	1 691 418 B (31.8%)

Table 5.8: Compression results for sdc method with different values of u

File	$u = 1$	$u = 2$	$u = 3$	$u = 4$
dickens	2 991 239 B (29.3%)	2 634 185 B (25.8%)	2 570 931 B (25.2%)	2 597 582 B (25.4%)
world192.txt	726 852 (29.3%)	635 991 B (25.7%)	617 844 B (24.9%)	621 709 B (25.1%)
samba	1 685 7733 B (24.9%)	1 472 842 B (21.7%)	1 428 509 B (21.1%)	1 441 753 B (21.3%)
xml	1 301 877 B (24.5%)	1 163 963 B (21.9%)	1 148 230 B (21.6%)	1 162 046 B (21.9%)

5.3 Performance of random access compression

For random access to compressed files the solution **darray** for Select query by Okanohara and Sadakane is used. For sdc coding method this method can be directly applied on auxiliary bit stream. Unlikely, fibc coding method needs some modifications. However, this modification is quite easy and straightforward. Tables 5.9, 5.10, show the sizes for darrays and also ratios of these sizes to the sizes of compressed files.

Tables 5.11 and 5.12 present the sizes and compression ratios for compressed test files supporting random access to the text with the help of darray data structure. For 'ws' column data structures were created on whole bit streams. Unlikely, the column 'wss' present the results when the darray is constructed on the part of the bit stream which corresponds the stream of words. (For wss form we store encoded

Table 5.9: Darray data structures for sdc method

File	ws	w	spless
dickens	388 440 B (10.7 %)	190 969 B (6.3 %)	243 720 B (6.9 %)
world192.txt	77 068 B (8.7 %)	37 950 B (5.2 %)	56 639 B (6.4 %)
samba	199 555 B (8.1 %)	99 035 B (5.8 %)	173 624 B (6.5 %)
xml	190 037 B (10.8 %)	90 242 B (6.9 %)	170 832 B (9.0 %)

Table 5.10: Darray data structures for fibc method

File	ws	w	spless
dickens	396 875 B (12.3 %)	205 687 B (7.9 %)	258 000 B (8.4 %)
world192.txt	88 000 B (10.9 %)	50 227 B (7.9 %)	68 882 B (8.9 %)
samba	217 832 B (10.0 %)	113 233 B (7.7 %)	193 722 B (8.4 %)
xml	197 160 B (12.1 %)	105 453 B (9.1 %)	179 340 B (10.6 %)

words and separators as two different streams)

Table 5.11: Compression results for sdc with darray data structure

File	ws	wss
dickens	3 986 798 (39.1%)	3 789 327 (37.1%)
world192.txt	956 556 (38.6%)	917 438 (37.0%)
samba	2 644 398 (39.1%)	2 543 878 (37.6%)
xml	1 94 663 (36.6%)	1 844 868 (34.7%)

Table 5.12: Compression results for fibc with darray data structure

File	ws	wss
dickens	3 610 669 (35.4%)	3 419 481 (33.5%)
world192.txt	871 329 (35.2%)	833 556 (33.7%)
samba	2 395 041 (35.4%)	2 290 442 (33.8%)
xml	1 821 662 (34.3%)	1 729 955 (32.6%)

The Table 5.13 presents the sizes and compression ratios for compressed files with attached darray data structures when words are only stored.

Table 5.13: Sizes of sdc compressed files with darray data structure for different values of u

File	$u = 1$	$u = 2$	$u = 3$	$u = 4$
dickens	3 182 208 (31.2%)	2 838 080 (27.8%)	2 764 194 (27.1%)	2 796 027 (27.4%)
world192.txt	764 802 (30.9%)	675 043 (27.2%)	660 629 (26.7%)	663 794 (26.8%)
samba	1 784 808 (26.4%)	1 574 520 (23.2%)	1 530 109 (22.6%)	1 546 303 (22.8%)
xml	1 392 119 (26.2%)	1 256 313 (23.6%)	1 244 330 (23.4%)	1 260 798 (23.7%)

5.4 Discussion

From the exprемental results it is seen that proposed compression methods are not best for compression of texts. They work well especially on natural English texts. For non-English texts like files samba and xml they are much worse than popular methods like gzip, bzip and WinRar. However, I mentioned that main goal of these methods is to achieve comparable compression ratios with preserving accessibility of the texts. From this point of view I can say that goal is met for the case of regular English texts. We see that the methods give comparable with other methods results and even attaching auxiliary data structures does not significantly increase the sizes of compressed files. One of advantages of the methods is their simplicity. fibc method gives compression ratios a little smaller than sdc one, but the difference is not so big. Another advantage of proposed methods which I did not consider in the thesis is that these methods support fast and flexible string matching on compressed files. The last is described in the paper [FN07].

Chapter 6

Conclusions

Random access to compressed text is a study of compression algorithms which allow to access the encoded sequence without decompression. We have looked on this topic from two points of view.

First, I considered algorithms for the text compression which potentially can be adapted for compression with ability of retrieving symbols from texts. However this adaptation is not considered, because not so many such results on this topic exist.

Then we turned attention to the succinct data structures and described main current results with the emphasis on the representation of the sequences. I noted that the task of efficient representation of the sequences in fact is the task of text compression with preserving accessibility.

New method was proposed for the random accessed text compression. I described coding procedure, gave theoretical analysis of the method and discussed some generalizations. Also, I showed that Fibonacci coding method is well-suited technique for retrieving symbols from encoded texts.

I demonstrated competitive ability of offered methods with the experiments which were done on several files and they showed that these methods give good results for the texts on natural languages.

Bibliography

[BCW90] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression.* Prentice Hall, 1990.

[BINP03] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. *Proceedings of the 25th European Conference on Information Retrieval Research (ECIR'03)*, pages 468–481, 2003.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC, 1994. Research Report 124. 10th May 1994.

[Cla98] D. R. Clark. *Compact pat trees.* PhD thesis, University of Waterloo, 1998.

[dMNZBY00] E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*, 18(2):113–139, 2000.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[FLMM05] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[FMMN06] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 2006. To appear.

[FN07] K. Fredriksson and F. Nikitin. Simple compression code supporting random access and fast string matching. *Proceedings of WEA'07, Lecture Notes in Computer Science*, 2007. To appear.

[Gal78] R. G. Gallager. Variations on a theme by huffman. *IEEE Trans. on Information Theory*, 24(6):668–674, 1978.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, 2003.

[GGV04] R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *SODA'04: Proc. 15th Annual ACM-SIAM Syposium on Discrete Algorithms*, pages 636–645, 2004.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 368–373, 2006.

[GN06] R. González and G. Navarro. Statistical encoding of succinct data structures. *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, pages 295–306, 2006.

[GRRR06] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[GV05] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[Huf52] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE 40*, pages 1098–1101, 1952.

[IM01] R. Y. K. Isal and A. Moffat. Word-based block-sorting text compression. *ACSC '01: Proceedings of the 24th Australasian conference on Computer science*, pages 92–99, 2001.

[Jac89a] G. Jacobson. Space-efficient static trees and graphs. *IEEE:1989:ASF*, pages 549–554, 1989.

[Jac89b] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie-Mellon, January 1989. Tech Rep CMU-CS-89-112.

[KNKP05] D. K. Kim, J. C. Na, J. E. Kim, and K. Park. Efficient implementation of rank and select functions for succinct representation. *Proc. WEA'05*, pages 315–327, 2005.

[MN06] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006. Special issue on "The Burrows-Wheeler Transform and its Applications". To appear.

[MR97] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. *IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[MRRR03] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. *ICALP*, pages 345–356, 2003.

[MT02] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publisher, 2002.

[Mun96] J. I. Munro. Tables. *FSTTCS*, pages 37–42, 1996.

[OS07] D. Okanohara and K. Sadakane. Practical entropy comressed rank/select dictionary. *Proceedings of ALENEX'07*, 2007.

[Pag02] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2002.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, 2002.

[RTT02] J. Rautio, T. Tanninen, and J. Tarhio. String matching with stopper compression. *DCC '02: Proceedings of the Data Compression Conference (DCC '02)*, page 469, 2002.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[SG06] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1230–1239, 2006.

[Sha51] C. E. Shannon. Prediction and entropy of printed english. *Bell Syst. Tech. J*, 30:50–64, 1951.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Commun. ACM*, 7(3):166–169, 1964.

[ST85] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[SW63] C. E. Shannon and W. Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, 1963.

[Wil83] D. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., 1999.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.

[Wri39] E. V. Wright. *Gadsby; a story of over 50,000 words without using the letter "E"*. Wetzel Publishing Co., Inc., 1939.

[Yan86] M. Yannakakis. Four pages are necessary and sufficient for planar graphs. *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, 28-30 May 1986, Berkeley, California, USA*, pages 104–108, 1986.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, 1949.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.