Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Compressing and Indexing Highly Similar Strings using LZW

Bc. Ondřej Perutka

Supervisor: prof. Ing. Jan Holub, Ph.D.

2nd May 2015

Acknowledgements

I would like to thank to my supervisor, prof. Ing. Jan Holub, Ph.D., for leading me the right way all the time, prof. Esko Ukkonen, Ph.D. for giving me an interesting insight into the world of bioinformatics and also to all of my friends, family and people I love for their endless help, support and inspiration.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 2nd May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Perutka, Ondřej. *Compressing and Indexing Highly Similar Strings using LZW.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tato práce se zabývá vývojem nové kompresní metody založené na LZW a zarovnání řetězců. Algoritmus je pojmenován ALZW a je navržen pro kompresi velmi podobných řetězců. Daná množina řetězců je komprimována pomocí předem určeného referenčního řetězce. V porovnání s podobně zaměřeným RLZ a všeobecně použitelným GZipem umožňuje ALZW velmi rychlou kompresi a pro podobné genetické sekvence dosahuje dobrých kompresních poměrů. V případě lidského chromozomu 20 dosahuje algoritmus dokonce lepších výsledků, něž podobně zaměřený algoritmus RLZ.

Klíčová slova datová komprese, LZW, ALZW, velmi podobné řetězce, DNA, vyhledávání v komprimovaném textu

Abstract

A new compression method based on LZW and sequence alignment is presented in this thesis. The algorithm is called ALZW and it is designed for compression of Highly Similar Strings. Strings in a given set are compressed relatively to a given reference sequence. Compared to similarly targeted RLZ and general purpose GZip, the algorithm offers very fast compression and it achieves good compression ratios for similar genomic sequences. It is even able to outperform the RLZ algorithm in case of human chromosome 20.

Keywords data compression, LZW, ALZW, highly similar strings, DNA, searching in compressed texts

Contents

List of Figures

List of Tables

Introduction

As the DNA sequencing methods have become relatively cheap and fast, a lot of DNA sequencing projects, such as *1000 Genomes*[1] and *Genome 10K*[2], have emerged. These projects yield massive amounts of data. In order to minimize storage costs, an effective compression method is needed. It is known that similarity of any two human genomes is more than 99 %. Similarly, there is only a small difference between genomes within particular species. This redundancy can be used for compression. The method presented in this thesis is focused on compression of highly similar DNA sequences; however, it can be easily generalized for an arbitrary input alphabet.

Related work

Except all general purpose compression methods, such as Lempel-Ziv family, there are several compression methods designed specifically for compressing biological sequences.

RLZ presented by S. Kuruppu et al. in [1] is one of these methods. This method is designed for compression of a set of highly similar DNA sequences. The approach is to compress all sequences in a dataset relatively to a reference sequence using LZ77. It also supports efficient random access to the compressed sequences. An optimized version of RLZ was presented in [2]. This version improves compression ratio of the original RLZ algorithm by exploiting *non-greedy* parsing.

A method very similar to RLZ was presented by S. Grabowski and S. Deorowicz in [3]. They used the same LZ77 compression scheme with a single reference sequence but there are several differences in comparison with RLZ. For example matches are found using hashing rather than suffix array and

[1] `http://www.1000genomes.org/`
[2] `https://genome10k.soe.ucsc.edu/`

extra phrases from compressed sequences are used in addition to the reference sequence.

Another interesting method by P. Procházka and J. Holub presented in [4] and called *BIO-FMI* is also designed for compressing sets of similar biological sequences. It is based on tracking changes between compressed sequences and a given reference sequence using known alignments and Wavelet Tree FM-index.

There are also other DNA related compression algorithms such as *GenCompress*, *biocompress-2* and *Comrad*. GenCompress, presented by X. Chen et al. in [5], is based on finding approximate repeats in DNA sequences. Biocompress-2 was presented by S. Grumbach and F. Tahi in [6] and it is based on detection of regularities, such as palindromes, in a given sequence. Comrad was presented by S. Kuruppu et al. in [7] and it is based on identifying exact repeated content in collections of input sequences using multi-pass iterative dictionary construction.

Structure of the thesis

The text starts with definitions of notions from the theory of formal languages and the graph theory that are used in the following chapters. Chapter 2 contains a description of the original LZW algorithm followed by considered modifications and the final version of the proposed ALZW algorithm. The issue of searching in ALZW compressed texts and interesting implementation details are discussed in Chapter 3 and 4. The thesis is ended with experimental evaluation of the ALZW compression and decompression algorithms, presented pattern-matching algorithms and comparison with other compression methods.

Chapter 1

Definitions

All necessary notions from the theory of formal languages and graphs used throughout this thesis are defined below in this chapter.

1.1 Graph theory

Definition 1.1. *Graph G is a pair $G = (V, E)$ of vertices (nodes) V and edges E, where V is a finite set and $\forall e \in E : e = \{u, v\} \wedge u, v \in V$.*

Definition 1.2. *Directed graph G is a pair $G = (V, E)$ of vertices V and edges E, where V is a finite set and $E \subseteq V \times V$.*

Definition 1.3. *Subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$.*

Definition 1.4. *Given a graph $G = (V, E)$ and $e \in E$ such that $e = \{u, v\}$, where $u, v \in V$, nodes u and v are called neighbours and relation between u, v and e is called incidence.*

Definition 1.5. *Degree of a node is number of incident edges with the node. Similarly input and output degree of a node is defined for directed graphs as number of edges entering and exiting the node.*

Definition 1.6. *Path in a graph $G = (V, E)$ is defined as a sequence:*

$$v_1\, e_1\, v_2\, e_2\, v_3\, \ldots\, v_{n-1}\, e_{n-1}\, v_n\ ,$$

where $v_i \in V$ for all $i \in \{1, 2, \ldots, n\}$, $e_i \in E$ and $e_i = \{v_i, v_{i+1}\}$ for all $i \in \{1, 2, \ldots, n-1\}$ and all vertices are distinct. If the first and the last vertex are not distinct, then the path is called a cycle. Length of a path is number of its edges.

Remark. *Note that the same definition of the path, cycle and their lengths can be also done for directed graphs.*

Definition 1.7. *Distance between two nodes in a graph is length of the shortest path between them.*

Definition 1.8. *An undirected graph is connected if there is a path between any two nodes. A directed graph is (weakly) connected if its undirected equivalent is connected. The directed graph is strongly connected if there is a directed path $u \to v$ and $v \to u$ for any pair of vertices u and v.*

Definition 1.9. *Tree is a connected undirected graph without cycles. Leaf is a node with degree 1, all other tree nodes are internal.*

Definition 1.10. *Directed tree is a directed graph which is a tree if directions of edges are ignored. Rooted tree is a directed tree with one node called a root and all edges oriented from the root to leaves.*

Definition 1.11. *Child of a node u in a rooted tree $T = (V, E)$ is any node $v \in V$, for which $(u, v) \in E$.*

Definition 1.12. *Parent of a node u in a rooted tree $T = (V, E)$ is a node $v \in V$, for which $(v, u) \in E$.*

Definition 1.13. *Ancestor of a node u in a rooted tree is defined as any node except u on the path from the root to the node u.*

Definition 1.14. *Tree depth is defined for rooted trees as maximum distance between the root and any leaf.*

1.2 Theory of formal languages and data compression

Definition 1.15. *Alphabet Σ is an arbitrary non-empty finite set of symbols.*

Definition 1.16. *String ω over an alphabet Σ is a finite sequence of symbols from Σ. Empty string, i.e. sequence containing no symbols, will be denoted by ε.*

Definition 1.17. *Given an arbitrary string ω of length n over an alphabet Σ such that $\omega = a_1 a_2 \dots a_n$, where $a_i \in \Sigma$ for all $i \in \{1, 2, \dots, n\}$, length of the string will be denoted by $|\omega|$. It also holds that $|\varepsilon| = 0$ for any Σ.*

Definition 1.18. *Given an arbitrary string $\omega = a_1 a_2 \dots a_n$ over an alphabet Σ, substring α of length m starting at position i (may be also called a factor) is defined as:*

$$\alpha = \omega[i..i + m - 1] = a_i a_{i+1} a_{i+2} \dots a_{i+m-1} \ ,$$

where $1 \leq i$ and $i + m - 1 \leq n$.

Definition 1.19. *Given an arbitrary string* $\omega = a_1 a_2 \dots a_n$ *over an alphabet* Σ*, subsequence* α *of length* m *is defined as:*

$$\alpha = a_{i_1} a_{i_2} \dots a_{i_m} \ ,$$

where $1 \leq i_1 < i_2 < \dots < i_m \leq n$*.*

Definition 1.20. *Given an arbitrary string* $\omega = a_1 a_2 \dots a_n$ *over an alphabet* Σ*, prefix* α *of length* $m \leq n$ *is defined as a substring of length* m *starting at position* 1*. A prefix of length* $m < n$ *is called a proper prefix.*

Definition 1.21. *Given an arbitrary string* $\omega = a_1 a_2 \dots a_n$ *over an alphabet* Σ*, suffix* α *of length* $m \leq n$ *is defined as a substring of length* m *starting at position* $n - m + 1$*. A suffix of length* $m < n$ *is called a proper suffix.*

Definition 1.22. *A proper prefix of a string, which is also a suffix of the string, is called a border.*

Definition 1.23. *Language* L *over an alphabet* Σ *is a non-empty set of strings over the alphabet* Σ*.*

Definition 1.24. *The set of all strings over an alphabet* Σ *will be denoted by* Σ^**.* Σ^+ *denotes the set of all non-empty strings over the alphabet* Σ*.*

Definition 1.25. *Finite automaton FA is a tuple* $FA = (Q, \Sigma, \delta, q_0, F)$*, where* Q *is a non-empty finite set of states,* Σ *is an input alphabet,* δ *is a function* $\delta : Q \times \Sigma \to \mathcal{P}(Q)$*, where* $\mathcal{P}(Q)$ *denotes powerset of* Q*,* $q_0 \in Q$ *is an initial state and* $F \subseteq Q$ *is a set of final states.*

Definition 1.26. *Deterministic finite automaton* $DFA = (Q, \Sigma, \delta, q_0, F)$ *is a finite automaton, where* $\delta : Q \times \Sigma \to Q$*.*

Definition 1.27. *Suffix-trie of a string* ω *over an alphabet* Σ *is a rooted tree, where each edge is labeled by a symbol from* Σ*, so that all paths can be represented as strings over* Σ*. There are no two outer edges of any node labeled by the same symbol. Suffix-trie contains all suffixes of* ω *and all paths starting in the root node are prefixes of a suffix of* ω*. This also implies that all paths in the suffix-trie are factors of* ω*.*

Definition 1.28. *Code* K *is a triplet* $K = (S, C, f)$*, where* S *is a finite set of source units,* C *is a finite set of codewords and* f *is an injective mapping* $S \to C^+$*.*

Definition 1.29. *Let us denote* B *as a language of all non-empty strings over a binary alphabet* $\{0, 1\}$*. Binary encoding of a language* L *with an alphabet* Σ *is a code* $K_{L,B} = (\Sigma, \{0, 1\}, f)$*. Given any string* $\omega \in L$ *of length* n *such that* $\omega = a_1 a_2 \dots a_n$*, function* $b : L \to B$ *defined as* $b(\omega) = f(a_1) f(a_2) \dots f(a_n)$ *will be used as a shorthand notation for binary encodings.*

Remark. *Note that the binary encoding can be also defined for finite sets of strings as a simple concatenation of binary encodings of all strings in a particular set.*

Definition 1.30. *Compression ratio of a binary encoding b_1 in comparison with a binary encoding b_2 for a string $\omega \in L$, where L is a language, is defined as:*

$$c_r = \frac{|b_1(\omega)|}{|b_2(\omega)|} \,. \tag{1.1}$$

Definition 1.31. *Suppose we have a language L over an arbitrary finite alphabet Σ and function $d : L \times L \to \mathbb{N}$ defined as Levenshtein distance [8]. A set of Highly Similar Strings (HSS) is a finite set $H \subseteq L$ where:*

$$\forall x, y \in H \quad : \quad x \neq y \Rightarrow 1 - \frac{d(x,y)}{|x| + |y|} > \sigma \tag{1.2}$$

and σ is a minimum required similarity.

Definition 1.32. *The problem of compressing HSS is a problem of finding a binary encoding:*

$$b_{HSS} : \{H : H \subseteq L,\ H \text{ is a set of HSS}\} \to B \,,$$

for which $c_r < 1$ compared to a given binary encoding of the language L.

Definition 1.33. *Given an information source $S \in \{s_1, s_2, \ldots, s_n\}$ and probabilities of source units $P = \{p_1, p_2, \ldots, p_n\}$, average entropy of the source is:*

$$H(S) = -\sum_{i=1}^{n} p_i \log_2 p_i \,. \tag{1.3}$$

Definition 1.34. *Given a message $T \in S^+$, where S is an information source, empirical entropy of 0-th order is defined as:*

$$H_0(T) = -\sum_{a \in S} \frac{n_T^a}{|T|} \log_2 \frac{n_T^a}{|T|} \,, \tag{1.4}$$

where n_T^a is number of symbols a in T. Empirical entropy of k-th order is defined as:

$$H_k(T) = \frac{1}{|T|} \sum_{w \in S^k} |w_T| H_0(w_T) \,, \tag{1.5}$$

where w_T is concatenation of symbols following w in T.

Chapter 2

Algorithm development

Several modifications of the original LZW algorithm were studied in order to create an algorithm for compression of HSS. All of these modifications, the original algorithm and also the final version of the compression algorithm will be described in this chapter.

2.1 Lempel-Ziv-Welch algorithm

This general-purpose compression algorithm, originally presented by T. A. Welch in [9], is based on a tree-like dictionary, where numbers from $\mathbb{N}$ are sequentially assigned to all nodes of the tree except its root. The dictionary initially contains only symbols from the used input alphabet. On each match, a corresponding phrase in the dictionary is extended by one symbol and the algorithm outputs number of the phrase. Pseudocode of this algorithm follows (see Algorithm 1). An example of an LZW dictionary is presented in Figure 2.1.

The main disadvantage of this algorithm in context of sets of HSS is its focus on repetitions in a single sequence. In addition, a phrase can be extended only by a single character on each occurrence of the phrase. This drawback is partially covered by LZY [10], LZMW [11] and LZAP [11]. These methods adapt faster to a given input; however, they are focused on compression of a single sequence rather than a set of HSS. They still need multiple occurrences of phrases to be able to adapt to them. The following dictionary properties has turned out to be important (not only) for compression of HSS:

1. *fast dictionary adaptation* – A complete phrase should be inserted into the dictionary as soon as possible to ensure that matching subsequences between two strings are encoded using a small amount of codewords (one in the best case).

2. *high dictionary depth* – Phrases should be as long as possible, so that lengths of codewords are not longer than phrases they represent.

Algorithm 1 Lempel-Ziv-Welch algorithm (LZW)

Input: string $S = s_1 s_2 \dots s_n$ over an alphabet Σ
Output: $LZW(S)$

```
1:  D ← ∅                                              ▷ dictionary
2:  P ← ε
3:  for all a ∈ Σ do
4:      InsertPhrase(D, a)
5:  for i ← 1..n do
6:      N ← P.s_i
7:      if N ∈ D then
8:          P ← N
9:      else
10:         output EncodePhrase(D, P)
11:         InsertPhrase(D, N)
12:         P ← s_i
```

3. *small dictionary size* – It is important to keep size of the dictionary (i.e. number of nodes) as low as possible to keep memory usage and lengths of codewords down.

2.2 Considered modifications

The first considered modification of the original algorithm originates in an observation that LZW dictionaries are subgraphs of suffix-tries. This modification is based on construction of suffix-trie for a given *reference sequence*. Then the suffix-trie is used as a dictionary for compressing the other sequences from a given set of HSS. The method greedily covers all sequences from the set by factors of a chosen reference sequence. Omitting the fact that the greedy coverage is not the best approach (see [2]), the greatest drawback of this modification is its memory complexity implied by exploiting suffix-tries. Their space complexity is $\mathcal{O}(n^2)$, where n is length of an input string.

The second modification was an attempt to address this drawback by splitting all sequences in a given set to blocks of a fixed size. Unfortunately, this method complicates later searching in compressed sets since a codeword may represent different phrases in different blocks.

The third modification addresses the same problem by limiting depth of the dictionary. This approach made the whole compression ineffective as lengths of used codewords became greater than lengths of phrases they represented.

The idea of the next modification was a stage to the final version of the compression algorithm. It is focused on pruning the dictionary from phrases which are not used during compression. It requires phrase counting and two

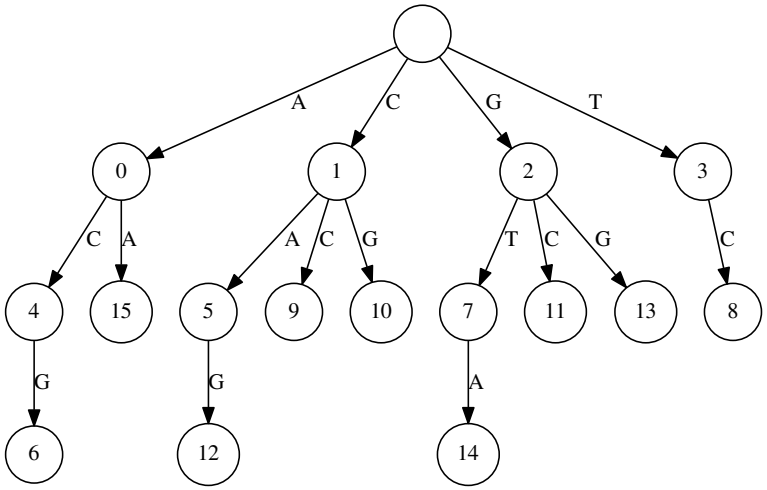

Figure 2.1: Example of an LZW dictionary for input string *ACACGTCCGCAGGTAACG*.

passes over the whole set of sequences. Except many other complications, the most important one is the fact that the dictionary is not implicitly encoded in the sequence of codewords. It would require saving the dictionary explicitly which would make the compression ineffective.

The final version of the compression algorithm is focused on building the dictionary only from phrases used in compression rather than pruning phrases that are not used. The method is based on sequence alignment and expects a set of aligned sequences as its input.

2.3 Sequence alignment

Sequence alignment is a well-known problem closely related to finding the longest common subsequence in a set of at least two strings. We talk about *pairwise alignment* if the set contains exactly two strings. Otherwise, we talk about *multiple alignment*. The basic approach for construction of a pairwise alignment of strings $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_m$ is based on computing the following function using dynamic programming:

$$\mathrm{lcs}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \mathrm{lcs}(i-1, j-1) + 1 & \text{if } x_i = y_j \\ \max(\{\mathrm{lcs}(i-1,j), & \\ \qquad \mathrm{lcs}(i, j-1)\}) & \text{otherwise} \end{cases} \tag{2.1}$$

and then reconstructing the alignment using backtracking.

A generalized version of this algorithm by S. B. Needleman and C. D. Wunsch [12] is used in bioinformatics for *global alignment* (i.e. alignment of whole strings) of protein/nucleotide sequences. This method uses gap penalties and special scoring matrices as weight functions. Here is an example of the function computed by Needleman-Wunsch algorithm using gap penalty -1 and PAM250 scoring matrix:

$$\mathrm{nw}(i,j) = \begin{cases} -j & \text{if } i = 0 \\ -i & \text{if } j = 0 \\ \max(\{\mathrm{nw}(i-1,j-1) + \mathrm{PAM250}(x_i, y_j), & \\ \qquad \mathrm{nw}(i-1,j) - 1, & \text{otherwise} \\ \qquad \mathrm{nw}(i,j-1) - 1\}) & \end{cases} . \quad (2.2)$$

There is also a variant of the Needleman-Wunsch algorithm presented by T. F. Smith and M. S. Waterman in [13] for computing so called *local alignments*. Local alignments are substring pairs/tuples with high similarity.

Here is an example of a simple global pairwise alignment created by solving the longest common subsequence problem:

```
A C A C G T C C G C - - A G G T A A C G
| | | | | |   | | |     | |     | | | |
A C A C G T A C G C A C A G - - A A C G
```

Time complexity of all three of these algorithms is $\Theta(mn)$ which makes them useless for large sequences. Fortunately, many heuristic algorithms for local and global alignment have been developed for use in bioinformatics (e.g. ACANA [14], ClustalW [15], BLAST [16], SPA [17]).

2.4 Alignment-based LZW algorithm (ALZW)

As already mentioned, the final version of the compression algorithm proposed in this thesis expects a set of aligned sequences as its input (pairwise alignments with a single reference sequence are enough). Selection of an algorithm for construction of sequence alignments is outside of the scope of this thesis since different algorithms are suitable in different contexts. Moreover, genomes of same species are usually sequenced in short "reads", which are aligned to a single reference genome, so all new sequences are saved aligned to that reference genome. For example, this is the case of 1000 Genomes project. There is no need for alignment computation in order to use ALZW algorithm for compression of sequences from 1000 Genomes project.

The process of compression (Algorithm 2) works as follows: The dictionary initialization is similar to the original LZW algorithm. The dictionary contains only the input alphabet and two extra codewords for insertion and deletion. Detailed initialization is not included in the pseudocode in order to keep it

simple. The insertion and the deletion placeholders are denoted by INS and DEL. For each pairwise alignment, the algorithm sequentially processes pairs of aligned symbols. The algorithm distinguishes four basic cases:

1. sequence of deletions,
2. sequence of insertions,
3. sequence of symbols that corresponds to a phrase in the dictionary and contains at least one mismatching symbol,
4. sequence of symbols containing no mismatches.

In case of deletion (Algorithm 3), the special codeword for deletion is output followed by number of deleted symbols. In case of insertion (Algorithm 4), the inserted subsequence is greedily covered by phrases already present in the dictionary. Then the insertion codeword is output followed by number of inserted phrases and codewords of these phrases. (The information of the number of inserted phrases is used later on decompression as the reference sequence pointer must not be incremented for these phrases.) The numbers of inserted phrases and deleted symbols are encoded using Elias delta code [18] denoted by Δ. The last two cases are handled by Algorithm 5. Let us denote the dictionary by D, the current subsequence by w, the next symbol by a and test of a phrase existence by $\in_P$. The algorithm always starts with empty w and it must handle four different situations:

1. $w \in_P D$, $wa \in_P D$,
2. $w \in_P D$, $wa \notin_P D$ and wa contains at least one mismatching symbol,
3. $w \in_P D$, $wa \notin_P D$, wa contains no mismatching symbols and ID of the next node added into D would be a power of two (therefore the width of all codewords[3] would be increased),
4. $w \in_P D$, $wa \notin_P D$ and wa contains no mismatching symbols.

In the first case, the algorithm simply extends the current subsequence by a. In the second case, the algorithm outputs the codeword for w and a is used as the first symbol of the following subsequence. In the third case, the algorithm outputs separately the codeword for w followed by the codeword for a. The whole phrase wa is then inserted into the dictionary and the algorithm starts over with empty w. This procedure is hidden inside the *InsertPhrase* function in Algorithm 5 on line 8. In the last case, the algorithm inserts wa into the dictionary and extends the current subsequence by a.

For example, let us suppose we would like to compress the sequence from the alignment presented above in Section 2.3. The first sequence from the

[3]number of bits required for encoding a codeword

Algorithm 2 Alignment-based LZW algorithm (ALZW) – compression

Input: set $S = \{S_1, S_2, \ldots, S_m\}$ of sequences aligned to a single reference sequence R
Output: $ALZW(S)$

```
1: D ← InitDictionary()
2: for i ← 1..m do
3:     EncodeSequence(D, S_i, R)

4: function EncodeSequence(D, S = s_1 s_2 ... s_n, R = r_1 r_2 ... r_n)
5:     i ← 1
6:     while i ≤ n do
7:         if s_i = DEL then
8:             i ← EncodeDEL(i, D, S)
9:         else if r_i = INS then
10:            i ← EncodeINS(i, D, S, R)
11:        else
12:            i ← EncodeMM(i, D, S, R)
```

Algorithm 3 ALZW deletion encoding

```
1: function EncodeDEL(i, D, S = s_1 s_2 ... s_n)
2:     j ← i
3:     while s_j = DEL ∧ j ≤ n do
4:         j ← j + 1
5:     output EncodePhrase(D, DEL)
6:     output Δ(j − i)
7:     return j
```

alignment will be our reference sequence. Our input alphabet will be $\Sigma = \{A, C, G, T\}$. Since the dictionary initially contains only six nodes – the input alphabet, the insertion codeword and the deletion codeword (the root node does not count), we need only three bits to encode a codeword. The algorithm reads the first three symbols (subsequence ACA) and extends the corresponding phrase in the dictionary. It cannot continue extending the current subsequence since the next node added into the dictionary would have ID = 8 and we would need one more bit to encode the codeword. Therefore the algorithm outputs codeword 7 (phrase ACA) followed by codeword 1 (phrase C) and inserts ACAC into the dictionary. The algorithm continues by reading the next two symbols (subsequence GT). The next symbol is A. Since the phrase GTA is not in the dictionary and A is a mismatch, the algorithm outputs codeword 9 (phrase GT) and A is used as the first symbol of the following subsequence. The current subsequence is extended by one more

Algorithm 4 ALZW insertion encoding

```
1: function EncodeINS(i, D, S = s_1 s_2 ... s_n, R = r_1 r_2 ... r_n)
2:     P ← ε, B ← ε, c ← 1
3:     while r_i = INS ∧ i ≤ n do
4:         N ← P.s_i
5:         if N ∈_P D then
6:             P ← N
7:         else
8:             b ← EncodePhrase(D, P)
9:             B ← B.b
10:            P ← s_i
11:            c ← c + 1
12:        i ← i + 1
13:    b ← EncodePhrase(D, P)
14:    output EncodePhrase(D, INS)
15:    output Δ(c)
16:    output B.b
17:    return i
```

Algorithm 5 ALZW match or mismatch encoding

```
1: function EncodeMM(i, D, S = s_1 s_2 ... s_n, R = r_1 r_2 ... r_n)
2:     P ← ε
3:     m ← s_i = r_i
4:     while r_i ≠ INS ∧ s_i ≠ DEL ∧ i ≤ n do
5:         N ← P.s_i
6:         m ← m ∧ s_i = r_i                          ▷ match indicator
7:         if N ∉_P D ∧ m then
8:             InsertPhrase(D, N)                     ▷ only phrases containing no
   mismatching symbols can be inserted into the dictionary
9:         if N ∈_P D then
10:            P ← N
11:        else
12:            output EncodePhrase(D, P)
13:            P ← s_i
14:            m ← s_i = r_i
15:        i ← i + 1
16:    output EncodePhrase(D, P)
17:    return i
```

symbol. The next symbol is G but the algorithm cannot continue extending the subsequence since ACG is not in the dictionary and it contains a mismatch (as in the previous case). The algorithm outputs codeword 6 (phrase AC) and G is used as the first symbol of the next subsequence. The current subsequence is extended by one more symbol. The algorithm cannot continue extending the subsequence since the next symbol is insertion. Therefore it outputs codeword 10 (phrase GC) and starts processing the insertion. The whole insertion can be covered by one phrase (phrase AC, codeword 6). The algorithm outputs codeword 5 (the insertion codeword) followed by $\Delta(1)$ (for one inserted phrase) and codeword 6 (for the inserted phrase). The next output is codeword 11 (for the subsequence AG following the insertion). The next subsequence is a two-symbol deletion. The algorithm outputs codeword 4 (the deletion codeword) followed by $\Delta(2)$ (for the deleted symbols). The last output is codeword 14 (for the subsequence AACG following the deletion). The complete sequence of codewords is:

$$7, 1, 9, 6, 10, 5, \Delta(1), 6, 11, 4, \Delta(2), 14.$$

The dictionary constructed by compressing this alignment is in Figure 2.2.

The process of decompression (Algorithm 6) works as follows: The dictionary is initialized in the same way as in the case of compression. A reference sequence pointer is set to point to the first symbol of the reference sequence. A codeword is read from the input. The following situations may occur:

1. it is the deletion codeword,
2. it is the insertion codeword,
3. it is a regular codeword and it is present in the dictionary,
4. it is a regular codeword and it is not present in the dictionary.

In the first case, the number of deleted symbols is read from the input and the reference sequence pointer is incremented by that number. In the second case, the number of inserted phrases is read from the input followed by codewords of these phrases. The phrases are output and the reference sequence pointer remains unchanged. In the third case, a phrase corresponding to the codeword is output and the reference sequence pointer is incremented by length of the phrase. In the last case, symbols from the current position in the reference sequence are read and a corresponding phrase in the dictionary is being extended until the codeword occurs in the dictionary. Then the corresponding phrase is output. The width of all codewords (i.e. number of bits required for encoding a codeword) is incremented by one every time a match phrase (a phrase containing no mismatching symbols) is followed by a matching symbol. In such case, the last phrase inserted into the dictionary is extended by

Algorithm 6 Alignment-based LZW algorithm (ALZW) – decompression

Input: common reference sequence R, sequence $C = c_1c_2 \dots c_n$ containing ALZW codewords

Output: set of decompressed sequences S

```
1: D ← InitDictionary()
2: S ← ∅, i ← 1
3: while i ≤ n do
4:     (i, s) ← DecodeSequence(i, D, R, C)
5:     S ← S ∪ {s}

6: function DecodeSequence(i, D, R = r_1 r_2 ... r_m, C = c_1 c_2 ... c_n)
7:     A ← ε, j ← 1
8:     while j ≤ m do
9:         if c_i = EncodePhrase(D, DEL) then
10:            j ← j + Δ^{-1}(c_{i+1})
11:            i ← i + 2
12:        else if c_i = EncodePhrase(D, INS) then
13:            k ← Δ^{-1}(c_{i+1})
14:            P ← DecodeINS(k, i + 2, D, C)
15:            A ← A.P
16:            i ← i + k + 2
17:        else
18:            P ← DecodeMM(c_i, j, D, R)
19:            A ← A.P
20:            j ← j + |P|
21:            i ← i + 1
22:    return (i, A)
```

Algorithm 7 ALZW match or mismatch decoding

```
1: function DecodeMM(c, i, D, R = r_1 r_2 ... r_n)
2:     P ← ε
3:     while c ∉_C D do
4:         P ← P.r_i
5:         if P ∉_P D then
6:             InsertPhrase(D, P)
7:         i ← i + 1
8:     return DecodeCodeword(D, c)
```

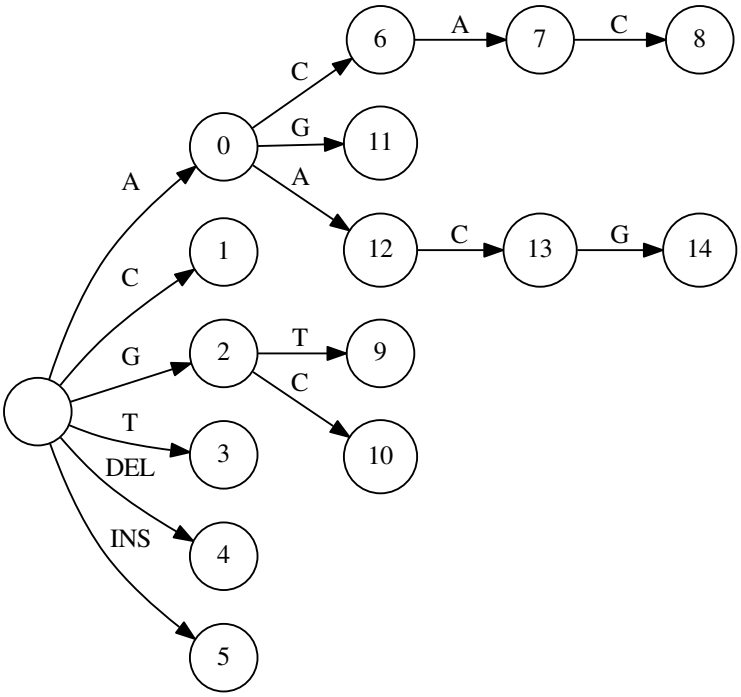

Figure 2.2: Example of an ALZW dictionary created by compressing the alignment presented in Section 2.3.

Algorithm 8 ALZW insertion decoding

```
1: function DecodeINS(k, i, D, C = c_1 c_2 ... c_n)
2:     P ← ε
3:     m ← i + k − 1
4:     for j ← i..m do
5:         p ← DecodeCodeword(D, c_i)
6:         P ← P.p
7:     return P
```

that symbol. This procedure is not present in the pseudocode in order to keep it simple. The process continues by reading another codeword from the input.

Note that there are two types of dictionary look ups used in the presented algorithms. The first type is a simple phrase look up (e.g. look up for phrase ACAC), which can be easily implemented by traversing the dictionary tree from its root. The second type is a codeword look up (e.g. look up for codeword 12). The codeword look up is required only for decompression and an extra

index is needed in order to implement it. To reflect the difference between these two types of look ups, the test of a phrase existence is denoted by $\in_P$ whereas the test of a codeword existence is denoted by $\in_C$.

Also note that the *DecodeSequence* function can be changed in a way that the algorithm will produce original alignments rather than simple sequences. It requires a simple change in decoding of insertions and deletions. Placeholders must be inserted into the reference sequence on positions of inserted symbols or a new reference sequence containing placeholders must be created. It depends whether we want to reconstruct a multiple or pairwise alignments. The same holds for deletions – placeholders must be put into the created sequences for every deleted symbol.

2.5 Complexity

Time complexity of Algorithm 3, 4 and 5 is linear with length of a given match/mismatch, insertion or deletion subsequence. The claim holds for Algorithm 4 and 5 since the test of a phrase existence, the *InsertPhrase* function and the *EncodePhrase* function can be all implemented as simple dictionary traversals. It requires only a constant number of steps per every symbol of a given subsequence. The claim also holds trivially for Algorithm 3. Therefore, time complexity of the *EncodeSequence* function used in Algorithm 2 is linear with length of a given alignment. Maximum length of an arbitrary pairwise alignment is sum of lengths of both original sequences. (The case where all symbols from one sequence are deleted and all symbols from the other sequence are inserted). Considering only pairwise alignments as input of the algorithm, the time complexity of the *EncodeSequence* function is $\mathcal{O}(r + s)$, where r is length of a given reference sequence R without insertion placeholders and s is length of a given sequence S without deletion placeholders. Time complexity of the whole compression algorithm is then:

$$\mathcal{O}(mr + \sum_{i=1}^{m} s_i) \,, \tag{2.3}$$

where m is number of compressed sequences and s_i is length of sequence S_i for all $i \in \{1, 2, \ldots, m\}$. Lower bound on the compression time complexity is:

$$\Omega(mr) \tag{2.4}$$

in case all input sequences are empty. Note that the deletion placeholders would be still read.

Space complexity of Algorithm 2 is given by maximum size of dictionary D and temporary buffers B, N and P. The temporary buffers may be extended only by a single symbol for each symbol of a given subsequence. Maximum length of the subsequence is equal to the length of the whole sequence. Space

complexity of these buffers is then $\mathcal{O}(\max_{i\in\{1,2,\ldots,m\}} s_i)$. The dictionary can be extended by at most one node per each symbol of every compressed sequence. The overall space complexity of Algorithm 2 is then:

$$\mathcal{O}(\sum_{i=1}^{m} s_i) \ . \tag{2.5}$$

The lower bound on the compression space complexity is trivially:

$$\Omega(1) \ . \tag{2.6}$$

It is again the case when all input sequences are empty.

Time complexity of decompression (Algorithm 6) depends on complexities of functions *DecodeMM* and *DecodeINS* (Algorithm 7 and 8). Complexity of the first one is $\mathcal{O}(r)$, where r is again length of the reference sequence, since a given codeword c may represent the whole reference sequence. Complexity of the second one is $\mathcal{O}(nr)$, where n is number of codewords, because of the same reason. The test of a codeword existence and the *InsertPhrase* function take only constant amount of time. Assuming codewords in the dictionary are indexed using a hashtable, the test can be implemented as a simple constant-time look up into the hashtable. The *InsertPhrase* function was explained above. The *DecodeCodeword* function requires time for a single codeword look up and time needed for reconstruction of a sequence the codeword represents. Time complexity of the whole decompression is then:

$$\mathcal{O}(nr) \ . \tag{2.7}$$

The lower bound on the decompression time complexity is:

$$\Omega(n) \tag{2.8}$$

in case all codewords represent only deletions.

Space complexity of Algorithm 6 is a little less clear in terms of the input size. Note that it cannot be more than in the case of compression since the same data structures are used – the temporary buffers for storing subsequences and the dictionary. Size of the dictionary is at most $\mathcal{O}(nr)$. It is the case when phrases does not have a common prefix and number of encoded sequences is proportional to number of codewords. Upper bound on the temporary buffer used inside the *DecodeMM* function is $\mathcal{O}(r)$ since – as mentioned above – a given codeword may represent the whole reference sequence. Upper bound on the temporary buffer used inside the *DecodeINS* function is $\mathcal{O}(nr)$ for the same reason. The overall space complexity is then:

$$\mathcal{O}(nr) \tag{2.9}$$

The lower bound on the decompression space complexity is trivially:

$$\Omega(1) \ . \tag{2.10}$$

It is again the case when all codewords represent only deletions.

Chapter 3

Searching in compressed text

Searching in compressed texts and searching in general are complex problems. The main goal is to exploit the fact that the compressed representation is shorter than the original text and to use the information stored in dictionary. A method presented by J. Lahoda and B. Melichar in [19] will be discussed in this chapter together with three general purpose algorithms for exact pattern matching. The goal of this chapter is not showing a best algorithm applicable in all possible contexts. It is rather focused on demonstrating different methods that can be used for pattern matching on ALZW compressed sequences.

3.1 Naive approach

The most straightforward and easiest solution of this problem is using a naive pattern matching algorithm (see Algorithm 9) on a decompressed sequence. The algorithm simply traverses the sequence from left to right and attempts to match a given pattern on every position in the sequence.

Algorithm 9 Naive pattern matching

Input: decompressed sequence $S = s_1 s_2 \dots s_n$, pattern $P = p_1 p_2 \dots p_m$
Output: indices of occurrences of P in S (if any)

```
1: l ← n − m + 1
2: for i ← 1..l do
3:     j ← 1
4:     c ← true
5:     while j ≤ m ∧ c do
6:         c ← c ∧ s_{i+j−1} = p_j
7:         j ← j + 1
8:     if c then
9:         output i
```

Time complexity of this solution is $\mathcal{O}(mn+\tau)$, where m is length of a given pattern P, n is length of the decompressed sequence and τ is time required for decompression. Space complexity is $\mathcal{O}(1+\gamma)$, where γ is space used for decompression.

3.2 Boyer-Moore-Horspool algorithm

A better approach is using a more sophisticated pattern matching algorithm rather than the naive one. An algorithm presented by R. N. Horspool in [20] was chosen due to its simplicity (see Algorithm 10) and the fact it is one of the fastest exact string matching algorithms.

Suppose we have a sequence S of length n and a pattern $P = a_1a_2 \ldots a_m$ of length m. Let $Q = a_1a_2 \ldots a_{m-1}$ be a prefix of P. The algorithm goes through the sequence S from left to right and attempts to match the pattern from its end. Then it shifts the pattern to the right to a position where the rightmost symbol of the current subsequence matches with the rightmost occurrence of the same symbol in the prefix Q. The pattern is shifted by m if there is no such symbol in Q. See Figure 3.1 for example.

Algorithm 10 Boyer-Moore-Horspool algorithm

Input: decompressed sequence $S = s_1s_2 \ldots s_n$, pattern $P = p_1p_2 \ldots p_m$
Output: indices of occurrences of P in S (if any)

```
1:  for all a ∈ Σ do
2:      shift[a] ← m
3:  l ← m − 1
4:  for i ← 1..l do
5:      shift[p_i] ← m − i

6:  i ← 1
7:  while i ≤ (n − m + 1) do
8:      j ← m
9:      c ← true
10:     while j ≥ 1 ∧ c do
11:         c ← c ∧ s_{i+j−1} = p_j
12:         j ← j − 1
13:     if c then
14:         output i
15:     i ← i + shift[s_{i+m−1}]
```

The worst case time complexity of this algorithm is $\mathcal{O}(nm)$ as in the case of the naive search. However, as the algorithm can shift a given pattern by more than one symbol at a time, the average-case complexity is $\mathcal{O}(n)$ (see [21]

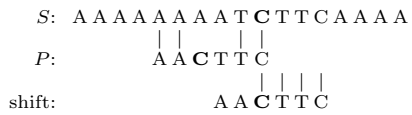

Figure 3.1: Example of the shift operation used in Algorithm 10.

for more details). The overall average-case time complexity of this solution is then $\mathcal{O}(n+\tau)$, where τ is time required for decompression. Space complexity is $\mathcal{O}(\sigma+\gamma)$, where σ is size of an input alphabet and γ is space used for decompression.

3.3 Pattern matching using deterministic finite automata

Using deterministic finite pattern matching automata is another solution based on decompression of all sequences. In order to construct a pattern matching automaton for a given pattern P, we need to construct a skeletal machine and a border array first.

Definition 3.1. *Given a pattern $P = p_1p_2\ldots p_m$ over an alphabet Σ, skeletal machine $SM = (Q, \Sigma, \delta, q_0, F)$ for the given pattern is a deterministic finite automaton, where:*

- $Q = \{q_0, q_1, \ldots, q_m\}$,
- $\delta(q_{i-1}, p_i) = q_i$ *for all* $i \in \{1, 2, \ldots, m\}$,
- $\delta(q_0, a) = q_0$ *for all* $a \in \Sigma$, $a \neq p_1$,
- $F = \{q_m\}$.

Definition 3.2. *Let $\beta : \Sigma^+ \to \Sigma^*$ be a function giving the longest border of a given non-empty string over an alphabet Σ, border array of a given string $S = s_1s_2\ldots s_n$ is defined as $B = (b_1, b_2, \ldots, b_n)$, where $b_i = |\beta(s_1s_2\ldots s_i)|$ for all $i \in \{1, 2, \ldots, n\}$.*

Border array for a given string of length n can be constructed using Algorithm 11 in $\Theta(n)$ time (see [22] for a detailed proof).

Given a pattern $P = p_1p_2\ldots p_m$, its skeletal machine and its border array, we can use the border array to compute all the remaining transitions of the corresponding pattern matching automaton. Let us suppose we have matched $Q = p_1p_2\ldots p_j$, where $j < m$, the last matched position in a given string X is i and we cannot continue since $p_{j+1} \neq x_{i+1}$. The longest proper suffix of Q

Algorithm 11 Construction of a border array

Input: string $S = s_1 s_2 \dots s_n$
Output: border array $B = (b_1, b_2, \dots, b_n)$ of the string S

```
1: b_1 ← 0
2: for i ← 2..n do
3:     j ← b_{i−1}
4:     while s_i ≠ s_{j+1} ∧ j > 0 do
5:         j ← b_j
6:     if s_i = s_{j+1} then
7:         b_i ← j + 1
8:     else
9:         b_i ← 0
```

which is also a prefix (the longest border) can be reused. If the symbol x_{i+1} cannot be matched with a symbol following the border prefix, we continue with the longest border of the current border until we can match the symbol or until we end up with empty border. This procedure can be used in construction of deterministic pattern matching automaton for a given pattern, see Algorithm 12. Example of a pattern matching automaton is in Figure 3.2. Time complexity of this algorithm is $\Theta(\sigma m)$, where σ is size of a given input alphabet and m is length of a given pattern.

Algorithm 12 Construction of a pattern matching automaton

Input: pattern $P = p_1 p_2 \dots p_m$ over an alphabet Σ
Output: pattern matching automaton $M = (Q, \Sigma, \delta, q_0, F)$ for the pattern P

1: Construct border array $B = (b_1, b_2, \dots, b_m)$ for the pattern P using Algorithm 11.

2: Append a special symbol $\# \notin \Sigma$ to the pattern P.

```
3: Q ← {q_0, q_1, ..., q_m}
4: F ← {q_m}
5: for all a ∈ Σ, a ≠ p_1 do
6:     δ(q_0, a) ← q_0
7: for i ← 1..m do
8:     δ(q_{i−1}, p_i) ← q_i
9:     for all a ∈ Σ, a ≠ p_{i+1} do
10:        δ(q_i, a) ← δ(q_{b_i}, a)
```

Since using a pattern matching automaton requires only a constant number of steps per each symbol of a given input string, time complexity of the whole

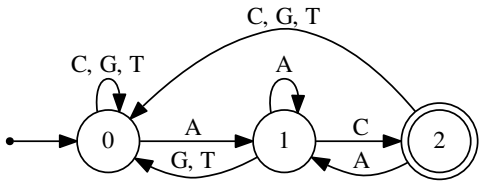

Figure 3.2: Example of a deterministic finite pattern matching automaton for pattern "AC" and input alphabet $\Sigma = \{A, C, G, T\}$.

solution is $\mathcal{O}(\sigma m + n + \tau)$, where n is length of a decompressed sequence and τ is time required for decompression. Space complexity is $\mathcal{O}(\sigma m + \gamma)$, where γ is space used for decompression.

3.4 Pattern matching on regular collage system

This algorithm, originally presented by J. Lahoda and B. Melichar in [19], is designed for pattern matching on so called *regular collage systems*. Collage systems, presented by T. Kida, Y. Shibata, M. Takeda, A. Shinohara and S. Arikawa in [23], can be used as an abstraction of various compression methods including LZW and ALZW.

Definition 3.3. *Collage system is a pair (D, S) of a dictionary D and a sequence S. The dictionary is a sequence of n assignments:*

$$X_1 = expr_1, X_2 = expr_2, \ldots, X_n = expr_n \ ,$$

where each $expr_k$ can be constructed as follows:

a	*for any* $a \in (\Sigma \cup \{\varepsilon\})$,	(primitive assignment)
$X_i X_j$	*for* $i, j < k$,	(concatenation)
${}^{[j]}X_i$	*for* $i < k$ *and* $j \in \mathbb{N}$,	(prefix truncation)
$X_i^{[j]}$	*for* $i < k$ *and* $j \in \mathbb{N}$,	(suffix truncation)
$(X_i)^j$	*for* $i < k$ *and* $j \in \mathbb{N}$.	(j times repetition)

The sequence $S = X_{i_1}, X_{i_2}, \ldots, X_{i_m}$ is a sequence of assignments from D.

There are several types of collage systems. The important two used by Lahoda and Melichar are *regular* and *simple* collage systems. Dictionary of a regular collage system can contain only primitive assignments and concatenations. Simple collage systems are defined as regular collage systems where for each concatenation assignment $X_i X_j$ either X_i or X_j is a primitive assignment.

3.4.1 Algorithm

Suppose we have a deterministic finite pattern matching automaton (see Figure 3.2 for example of such automaton). The algorithm is based on an observation that there is only a limited amount of phrases in the dictionary D that behave differently in context of the automaton (see Definition 3.5).

Definition 3.4. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite pattern matching automaton and δ^* a transitive-reflexive closure of δ. Any two strings $u, w \in \Sigma^*$ are in relation $\sim_M$ if and only if the following holds for all $q \in Q$:*

1. *$\delta^*(q, u) = \delta^*(q, w)$,*
2. *exactly one of the following conditions holds:*
   a) *there is a prefix u' of u and a prefix w' of w such that $\delta^*(q, u') \in F$ and $\delta^*(q, w') \in F$,*
   b) *for all prefixes u' of u and all prefixes w' of w holds that $\delta^*(q, u') \notin F$ and $\delta^*(q, w') \notin F$.*

In other words, strings u and w are in relation $\sim_M$ if and only if they stop in the same state for any initial state q of the finite automaton M and they both either go through at least one final state of the automaton or they do not go through any final state. Note that relation $\sim_M$ is equivalence (see [19] for a detailed proof).

Definition 3.5. *Two phrases behave differently in context of a given deterministic finite pattern matching automaton M unless they are in relation $\sim_M$.*

Definition 3.6. *Let us consider an arbitrary ordering of states Q of the automaton M. Signature $S(u)$ is defined for each string $u \in \Sigma^*$ as a vector $S(u) = ((q'_1, f_1), (q'_2, f_2), \ldots, (q'_{|Q|}, f_{|Q|}))$, where $q_i \in Q$ and $f_i \in \{\text{true}, \text{false}\}$ for all $i \in \{1, 2, \ldots, |Q|\}$. It also holds that $q'_i = \delta^*(q_i, u)$ and f_i is* true *if and only if there is a prefix u' of u such that $\delta^*(q_i, u') \in F$, otherwise f_i is* false*.*

Note that strings u, w are in relation $\sim_M$ if and only if $S(u) = S(w)$.

As already mentioned, there is only a limited amount of strings that behave differently in context of a given deterministic finite pattern matching automaton. We can use the equivalence $\sim_M$ to divide Σ^* into disjoint classes. We pick a shortest string, so-called *representative*, out of each of these classes. All these strings together form a set of representatives W. The set W is finite since there cannot be more classes of equivalence than number of distinct signature vectors. Algorithm 13 describes construction of the set of representatives for a given deterministic finite automaton M. See Table 3.1 for example of a set of representatives for the automaton from Figure 3.2.

Algorithm 13 Construction of a set of representatives

Input: deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$
Output: set of representatives W for the automaton M

```
1: W ← ∅
2: S ← ∅
3: U ← {ε}
4: while U ≠ ∅ do
5:     remove a string w from U such that ∀u ∈ U : |u| ≥ |w|
6:     if S(w) ∉ S then
7:         W ← W ∪ {w}
8:         S ← S ∪ {S(w)}
9:         for all a ∈ Σ do
10:            U ← U ∪ {w.a}
```

Table 3.1: Example of a set of representatives for the automaton from Figure 3.2.

	$S(u)$		
u	0	1	2
ε	(0, false)	(1, false)	(2, false)
A	(1, false)	(1, false)	(1, false)
C	(0, false)	(2, true)	(0, false)
G	(0, false)	(0, false)	(0, false)
AC	(2, true)	(2, true)	(2, true)
CA	(1, false)	(1, true)	(1, false)
CC	(0, false)	(0, true)	(0, false)
ACA	(1, true)	(1, true)	(1, true)
ACC	(0, true)	(0, true)	(0, true)

Definition 3.7. *Characteristic automaton $M_H = (Q_H, W, \delta_H, q_{H0}, \emptyset)$ is defined for a given deterministic finite automaton M and its set of representatives as:*

- $Q_H : Q_H = W$,
- $\delta_H : Q_H \times W \to Q_H : \delta_H(q_H, w) = u$, *for* $u, w \in W$, *where* $q_H w \sim_M u$,
- $q_{H0} : q_{H0} = \varepsilon$.

In case of simple collage systems (note that this is also the case of ALZW), we can use a simplified version of the characteristic automaton. The simplified characteristic automaton $M_H = (Q_H, \Sigma, \delta_H, q_{H0}, \emptyset)$ takes only Σ as its alphabet and its transition function is defined as:

$\delta_H : Q_H \times \Sigma \to Q_H : \delta_H(q_H, a) = u$, for $a \in \Sigma$ and $u \in W$, where $q_H a \sim_M u$.

An example of such automaton is in Figure 3.3.

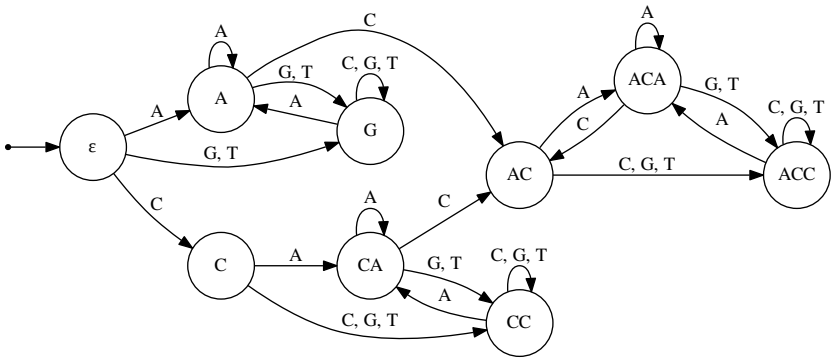

Figure 3.3: Example of the simplified version of the characteristic automaton for the deterministic finite automaton from Figure 3.2 and its set of representatives.

Algorithm 14 describes pattern matching on regular collage systems in context of ALZW. Dictionary D is built exactly as in case of the decompression but no codeword is decompressed. Representatives of all phrase-representing codewords (i.e. no insertion or deletion codewords or delta-encoded numbers) are generated using Algorithm 15. The representatives' signatures are used then to determine the next state of the pattern matching automaton M and to check if any final state of the automaton was reached. In such case, exact positions of a given pattern must be found in the decompressed subsequence. The pattern matching automaton M or an arbitrary string matching algorithm can be used to find exact positions of the pattern. Note that the match positions of P may start in codewords preceding the current codeword, so the string matching algorithm must start in a predecessor of the current codeword. Length of the decompressed subsequence between such predecessor and the current codeword must be at least $(|P| - 1)$.

If the representative of the current codeword is not known, it is generated using known representatives in the ancestor line of the codeword. Ancestors of the codeword are traversed until an ancestor with a known representative is found. The characteristic automaton is used then to find the representative of the current codeword. Note the difference between LZW and ALZW. In the first case, this procedure always takes only a constant number of steps since the parent of the current codeword must have been present in the already processed part of the sequence of codewords. In the latter case, the procedure may cause decompression of the whole codeword since adding a phrase into ALZW dictionary is not based on repetitions.

Algorithm 14 Lahoda-Melichar algorithm in the context of ALZW

Input: reference sequence R, sequence $C = c_1c_2 \dots c_n$ containing ALZW codewords, pattern P

Output: indices of occurrences of P (if any)

For simplicity, let us assume the given sequence of codewords represents only a single sequence.

For the given pattern P construct deterministic pattern matching automaton M, set of representatives W, its corresponding set of signatures S and characteristic pattern matching automaton M_H. Symbol S will be also used to denote mapping between sets W and S.

$D \leftarrow \mathit{InitDictionary}()$
$r \leftarrow \mathit{GetRoot}(D)$
$U \leftarrow \{(r, \varepsilon)\}$ ▷ *codeword* → *representative* map
$q \leftarrow q_0$
for $i \leftarrow 1..n$ **do**
    update D using c_i
    if c_i is a phrase-representing codeword **then**
        $w \leftarrow \textsc{GetRepresentative}(c_i, D, U, M_H)$
        $s \leftarrow S(w)$
        $(q, f) \leftarrow s(q)$ ▷ get signature element corresponding to q
        if f **then**
            output find exact positions of P in c_i and its predecessors

Algorithm 15 Get representative for a given codeword

function $\textsc{GetRepresentative}(c, D, U, M_H = (Q_H, W, \delta_H, q_{H0}, \emptyset))$
    $X \leftarrow \varepsilon$
    $r \leftarrow \mathit{GetRoot}(D)$
    while $(c, \cdot) \notin U \ \wedge \ c \neq r$ **do**
        $a \leftarrow \mathit{PhraseLastSymbol}(D, c)$
        $c \leftarrow \mathit{ParentCodeword}(D, c)$
        $X \leftarrow X.a$
    $w \leftarrow U(c)$
    $k \leftarrow |X|$
    for $i \leftarrow k..1$ **do**
        $a \leftarrow X[i]$
        $w \leftarrow \delta_H(w, a)$
        $c \leftarrow \mathit{ChildCodeword}(D, c, a)$
        $U \leftarrow U \cup \{(c, w)\}$
    return w

3.4.2 Complexity

Time complexity of the preprocessing phase consists of time needed for construction of the deterministic exact string matching automaton M, the set of representatives W, the corresponding set of signatures S and the characteristic automaton M_H. As already shown, automaton $M = (Q, \Sigma, \delta, q_0, F)$ can be constructed in $\mathcal{O}(\sigma m)$ time, where σ is size of the alphabet and m is length of a given pattern P, yielding an $(m + 1)$ state deterministic automaton. Upper bound on the set of representatives, in case of an exact string matching automaton, is $\mathcal{O}(m^2)$ (see [19] for details). Number of candidates added into set U used in Algorithm 13 is $\mathcal{O}(\sigma m^2)$ since for each representative exactly σ other candidates are added. Set U can be implemented using queue, so it takes only a constant number of steps to remove the shortest candidate (it is always the first one) or to insert a new candidate. Sets W and S can be implemented using hashset. Hash for a given string or signature can be precomputed in time of their construction, so getting hash of a particular string or signature will later take only a constant number of steps. (In case of a hash collision, the elements must be still compared.) Signature for a given candidate $w = xa$, where x is a prefix and a is the last symbol of w, can be computed in $\mathcal{O}(m)$ steps, since we can use the already known signature of the prefix x. The look up for a signature in S will also take at most $\mathcal{O}(m)$ steps. Therefore, the overall time complexity of Algorithm 13 is $\mathcal{O}(\sigma m^3)$. Note that only a simple characteristic automaton is needed in case of ALZW. Such automaton can be constructed in $\mathcal{O}(\sigma m^3)$ since for every representative $w \in W$ and all $a \in \Sigma$ we have to compute signature of wa in order to get the next state. Time complexity of the whole preprocessing phase is then:

$$\mathcal{O}(\sigma m^3) \,. \tag{3.1}$$

Space complexity of the preprocessing phase is:

$$\mathcal{O}(m^3 + \sigma m^2) \tag{3.2}$$

since size of representatives is proportional to m and for each representative we need to store its signature of size $\Theta(m)$ and a next state for all $a \in \Sigma$. (Size of the automaton M is $\mathcal{O}(\sigma m)$.)

Time complexity of the searching phase is $\mathcal{O}(|D| + |C|)$, where D is a dictionary and C is a sequence of codewords, since for each node added into D we can compute its representative in constant time using M_H and representative of the node's parent. Note that this may lead to $\mathcal{O}(n)$ time complexity, where n is length of the decompressed sequence, in case all codewords must be decompressed in order to compute representatives or to find exact positions of a given pattern. If the sequence of codewords encodes only a single sequence, space complexity of the dictionary is $\mathcal{O}(|R|)$, where R is a reference sequence. Time complexity of the searching phase in terms of the input size is then:

$$\mathcal{O}(|R| + |C|) \,, \tag{3.3}$$

time complexity of the whole algorithm is then:

$$\mathcal{O}(|R| + |C| + \sigma m^3) \tag{3.4}$$

and space complexity of the algorithm is:

$$\mathcal{O}(|R| + m^3 + \sigma m^2)\ . \tag{3.5}$$

Chapter 4

Implementation

Algorithms for ALZW compression and decompression together with pattern matching algorithms described in Chapter 3 were implemented into two command line applications *alzw* and *alzwq*. Another two implemented applications are called *sam2fasta* and *sam2seq*. These command line tools are for converting given genomic sequences in SAM format to sequence alignments in FASTA format (see Appendix B and C for the format descriptions) or to sequences of letters *a, c, g, t* and *n* respectively. The SAM format is used by 1000 Genomes project for storing individual sequences. Sequences from this project's database were used for testing. All applications were implemented in C++ 11. This chapter describes all interesting implementation details and trade-offs.

4.1 alzw

This application is a command line tool for compression and decompression of given genomic sequences using algorithm ALZW described in Chapter 2. In case of compression, at least one pairwise alignment in FASTA format is expected. All pairwise alignments passed to the input as command line arguments must be aligned to the same sequence, otherwise the behavior is undefined. The compressed set of sequences is put in the standard output. In case of decompression, the reference sequence together with the set of compressed sequences are passed to the input as command line arguments. The decompressed sequences are put in files. All names of the output files are concatenations of the original alignment file names and *".fa"* suffix. The output sequences are in FASTA format (see Appendix B). The application is used as follows:

```
alzw [OPTIONS] [RSEQ] [ALZW] [A1 [A2 [...]]]

    RSEQ  reference sequence file in FASTA format (used
          only in case of decompression)
```

```
        ALZW  ALZW compressed file (used only in case of
              decompression)
        A#    sequence alignment file in FASTA format (used
              only in case of compression)

    OPTIONS:

        -d     decompression
        -s num synchronization period [200] (valid only in
               case of compression)
        -a     adaptive synchronization (valid only in case
               of compression)
        -h     show help
```

Compression example:

```
alzw *.afasta > compressed.alzw
```

Decompression example:

```
alzw -d rseq.fa compressed.alzw
```

4.1.1 ALZW file format

ALZW compressed files consist of two parts (see Figure 4.1) – a file table and a sequence of codewords. Figure 4.2 depicts structure of the file table. The first element of the table is 32 bit unsigned integer representing number of records stored in the table. The records follows. Each record is a NULL terminated string representing path to the original alignment file. The sequence of codewords represents all the compressed alignments. Width of each codeword depends on state of the encoder during compression (size of the dictionary, values of the delta-encoded numbers). There are no separators between individual compressed sequences. During decompression, a new sequence is started on each successful completion of an alignment.

4.1.2 Optimizations

Early experiments with implementation of the encoding algorithm showed very high memory consumption. The experiments were performed with several randomly chosen sequences of human chromosome 20 from the database of 1000 Genomes project. Two optimizations were employed in order to decrease memory consumption. Detailed measurements of memory consumption with and without these optimizations will be presented in Chapter 5.

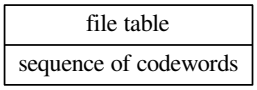

Figure 4.1: ALZW file format.

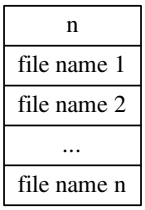

Figure 4.2: Structure of the file table used in ALZW file format.

The first optimization is based on an observation that adding a phrase into the ALZW dictionary often creates a sequence of nodes with consecutive numbers. Such sequence can be collapsed into a single node. Unlike regular nodes, the collapsed nodes must also store information about the collapsed transitions, i.e. number of transitions and the transition symbols. Figure 4.3 shows a subset of the dictionary presented in Figure 2.2. Sequences starting in nodes 6 and 12 were collapsed into two single nodes.

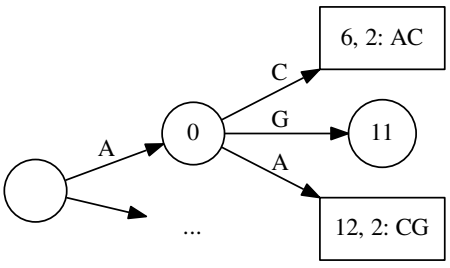

Figure 4.3: Example of an ALZW dictionary with collapsed nodes.

Unfortunately, this optimization complicates codeword look ups during decompression. Without this optimization, a codeword look up can be done in constant time using a hashtable-based index of the dictionary nodes. The same method can be used even with this optimization but instead of pointers to individual nodes, pointer-offset pairs would be used as values of the table. This method has a big disadvantage – it would almost nullify the effect of this optimization since all codewords, even the collapsed ones, must be indexed. A better index is needed in order to solve this problem. The index must support efficient insertions, deletions and queries for non-existing node identifiers. A node with the closest lower number must be always returned for such query. This property ensures support of look ups for codewords contained within collapsed nodes. Balanced binary search trees were chosen as ideal candidates. The final implementation of the node index uses RB tree as it requires less operations on insertion and deletion in comparison with AVL tree.

Using RB tree based index of codewords also increases decoding time complexity to $\mathcal{O}(nr \log(nr))$, where n is number of codewords and r is length of a given reference sequence. The factor of $\log(nr)$ is derived from the upper bound on size of the dictionary, which is $\mathcal{O}(nr)$. The encoding time complexity remains unchanged since the index is not needed for encoding.

The second memory optimization is trying to address the fact that SNPs are present on different positions within individual input sequences. This issue causes the compression algorithm to loose its "synchronization" when compressing multiple sequences. The algorithm starts creating phrases at different offsets for each compressed sequence and therefore it creates different sets of phrases for these sequences. The goal is to reuse as many phrases between all compressed sequences as possible. The optimization creates explicit *synchronization points* within a given set of input sequences. When a synchronization point is reached during compression, a new phrase is started.

Two variants of this optimization were tested. The first (static) variant creates a synchronization point every n bases, where n is an input parameter. The second variant is adaptive. It requires one additional pass over the input sequences before their compression in order to find positions of all SNPs. Synchronization points are put right behind these positions. In case a sequence of such synchronization points would create a phrase shorter than n bases, where n is again an input parameter, only the first synchronization point from the sequence is used. In both variants, offsets of synchronization points are computed relatively to the beginning of a given reference sequence to take possible differences between individual input alignments into account. See Figure 4.4 for illustration of the difference in compression with and without synchronization points.

The last interesting optimization adds a new special codeword into the dictionary. The codeword is used as an indicator of codeword width increments. It is used every time when adding a new node into the dictionary would increase width of codewords. This allows to read a sequence of code-

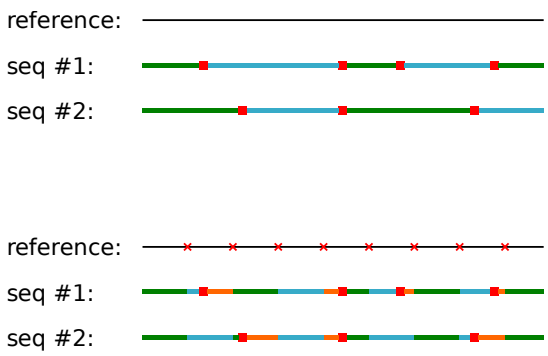

Figure 4.4: Illustration of the difference in compression with and without explicit synchronization points. Synchronization points are denoted by red crosses, SNPs by red squares, different phrases by colored lines.

words without decoding it. The optimization is not needed for compression or decompression, it is required for implementation of the algorithm presented in Section 3.4. Negative effect on compression ratio is negligible for long sequences.

4.2 alzwq

This command line tool provides a simple interface for searching of given patterns in ALZW compressed files. All algorithms described in Chapter 3 were implemented. Selection of a particular algorithm can be done using a command line argument. Queries are read from the standard input, one query per line. Empty line quits the application. An ALZW compressed file together with a corresponding reference sequence must be passed as command line arguments. The application is used as follows:

```
alzwq [OPTIONS] RSEQ ALZW

    RSEQ  reference sequence file in FASTA format
    ALZW  ALZW compressed file

OPTIONS:

    -h     show help
    -a alg searching algorithm [lm], valid options are:
               lm  Lahoda-Melichar
```

```
dfa deterministic finite automaton
bmh Boyer-Moore-Horspool
s   simple search (naive algorithm)
```

Loading compressed sequences:

```
alzwq rseq.fa compressed.alzw
```

Entering query:

```
AGAGGGACTGGGTTCCATGGGACTGCAGGAGAGCAAGGAGGCCACTGTGGCCGAAATGGA
```

Query output:

```
match (seq: 1, offset: 57046274)
match (seq: 2, offset: 57044658)
match (seq: 5, offset: 57043848)
match (seq: 6, offset: 57055067)
match (seq: 7, offset: 57048277)
match (seq: 8, offset: 57044878)
match (seq: 9, offset: 57054258)
match (seq: 10, offset: 57047313)
...
```

4.2.1 Optimizations

In order to speed up searching of multiple patterns, the whole process has been split into two phases – preprocessing and searching. The ALZW dictionary is constructed only once in the preprocessing phase. It is used then for all queries read from the standard input. This optimization also creates space for a second one – hash-based node indexing. Since the dictionary nodes are indexed using RB tree (see Section 4.1), we can improve codeword look up times using a hash-based index. The index will be limited only on the codewords present in the compressed file, which is usually a ralatively small subset of all codewords present in the dictionary. The hash-based index together with the dictionary construction require only one additional pass over the compressed file before execution of the first query.

4.3 sam2fasta, sam2seq

Both of these applications are command line conversion tools for sequences in binary SAM format (see Appendix C). The first one produces sequence

alignments in FASTA format used by the *alzw* application. The second one produces sequences of characters *a, c, g, t* and *n* used by *rlz* application. Both applications require a single reference sequence file in FASTA format and at least one sequence file in binary SAM format to be passed as command line arguments. Output file names are concatenations of the original file names and *".afasta"* suffix in case of the first application and *".seq"* suffix in case of the second one. The applications are used as follows:

```
sam2fasta/sam2seq [OPTIONS] RSEQ FILE1 [FILE2 [...]]

    RSEQ  reference sequence file in FASTA format
    FILE# sequence file in binary SAM format

OPTIONS:

    -h     show help
```

Usage examples:

```
sam2fasta rseq.fa *.bam
sam2seq rseq.fa *.bam
```

4.4 Compilation

All applications were built and tested in Fedora 21 x86_64, a GNU/Linux distribution. They can be compiled using GCC C++ 11 compiler version 4.9.2. In order to make the compilation simple, a *Makefile* is attached. The *samtools* library (version 0.1.19) used in *sam2fasta* and *sam2seq* applications is included as a part of the project and compiled together with the applications. The following extra libraries are required:

- Zlib version 1.2.3 or higher,
- POSIX threads.

Chapter 5

Experimental evaluation

This chapter presents results of experimental evaluation of the ALZW compression and decompression algorithms presented in Chapter 2 and the pattern matching algorithms presented in Chapter 3. Twenty randomly chosen sequences of human chromosome 20 from the database of 1000 Genomes Project were used for this evaluation.

Memory consumption was monitored using GNU *time* utility with "-v" parameter. Compression ratio was computed relatively to size of corresponding sequences in FASTA format. Two types of compression ratio are presented. The first type includes only compressed sequences, i.e. without the used reference sequence. The second type is overall compression ratio which includes also the reference sequence as it is needed for decompression. The reference sequence was taken in its plain form (it is simple to reconstruct the FASTA format again) and compressed using GZip.

All experiments were performed in Fedora 21 x86_64 Linux environment on a machine with Intel Core i7-2620M CPU @ 2.70 GHz and 8 GB main memory. Each experiment was repeated twice in order to minimize time distortions caused by IO cache of the host operating system. Detailed data from the measurements can be found in Appendix A.

5.1 Compression and decompression

Figure 5.1 depicts relation between compression time, decompression time and number of sequences. The sequences were compressed with static synchronization period of 200 bases. As expected from the time complexity analysis presented in Section 2.5, the relation between compression time and number of sequences is linear. The decompression time on the other hand does not show any signs of the $\mathcal{O}(\log n)$ RB index used. Moreover, the decompression is even faster than compression.

Effect of the RB index is more obvious when comparing memory consumption (Figure 5.2). The memory consumption also seems to be linear with

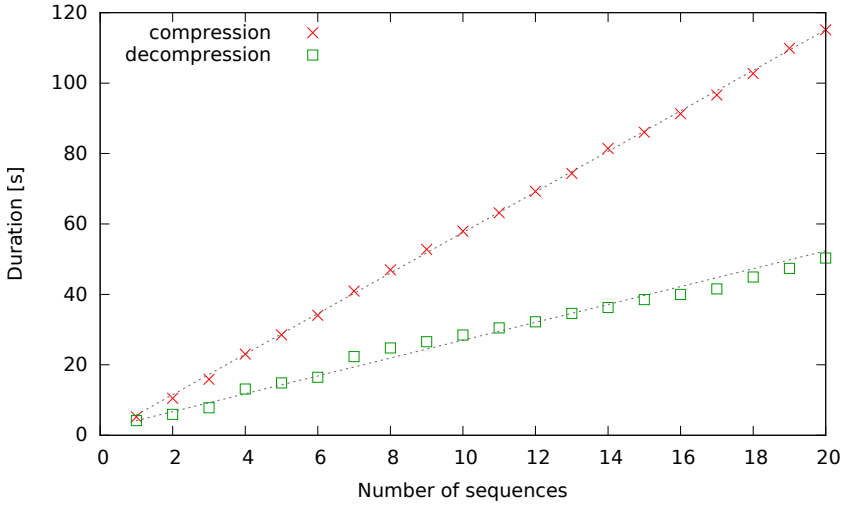

Figure 5.1: Relation between compression and decompression times and number of sequences.

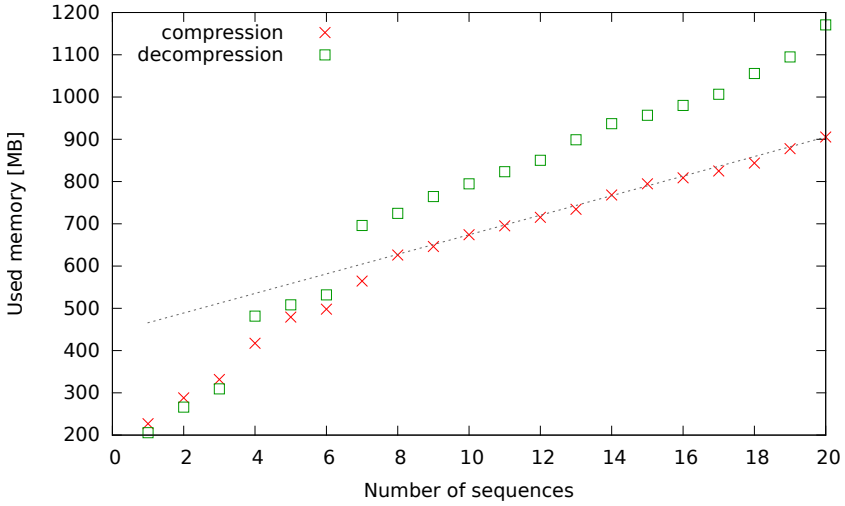

Figure 5.2: Relation between memory consumption of compression and decompression and number of sequences.

number of compressed sequences; however, it is more sensitive on content of the sequences (see the unevenness between 1 and 8 sequences).

Compression ratio (without the reference sequence) compared to encoding in FASTA format is shown in Figure 5.3. The algorithm achieves compression ratio around 5 % for sequences of human chromosome 20 and static synchronization period of 200 bases. The overall compression ratio (i.e. including the

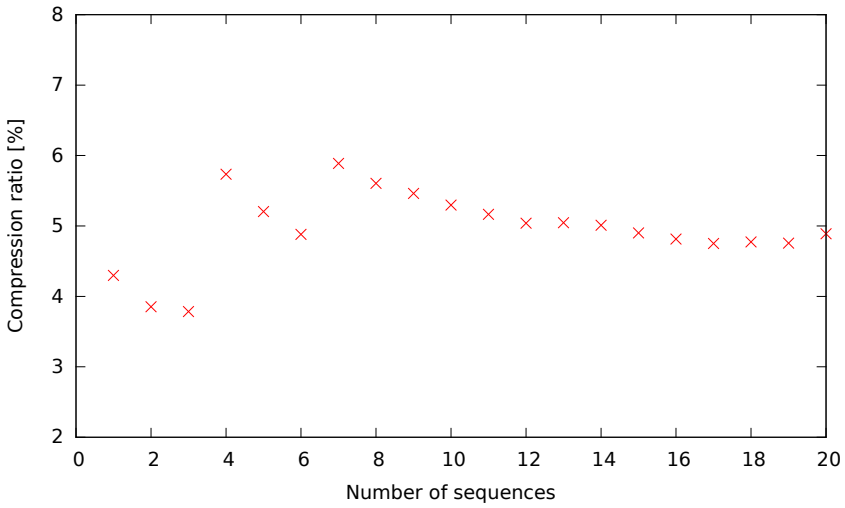

Figure 5.3: Relation between compression ratio and number of sequences.

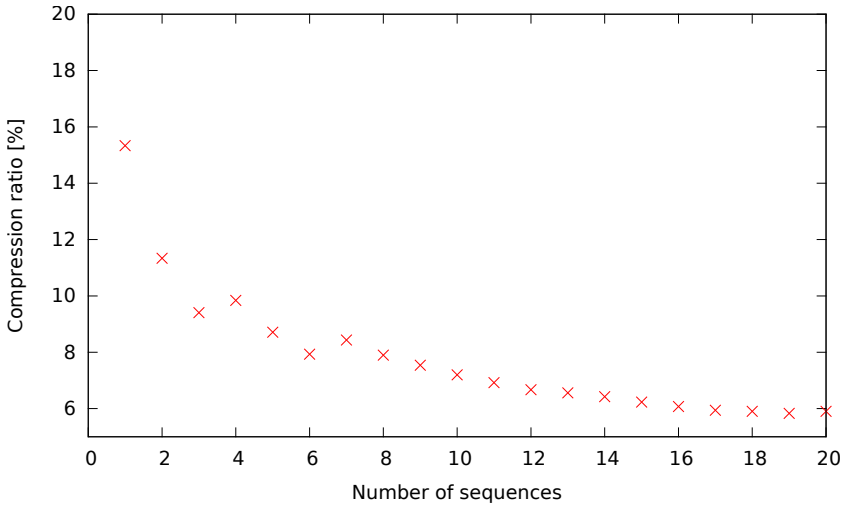

Figure 5.4: Relation between overall compression ratio and number of sequences.

reference sequence) is shown in Figure 5.4. It is clear that compressing more sequences minimizes the negative effect of storing the reference sequence.

5.2 Optimizations

Comparison of memory consumption with and without the node collapsing optimization described in Section 4.1 is shown in Figure 5.5. Due to memory

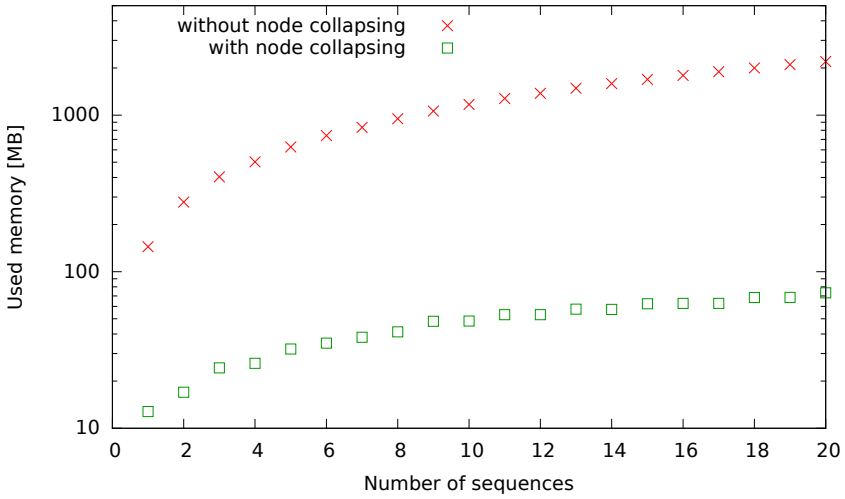

Figure 5.5: Comparison of memory consumption with and without node collapsing.

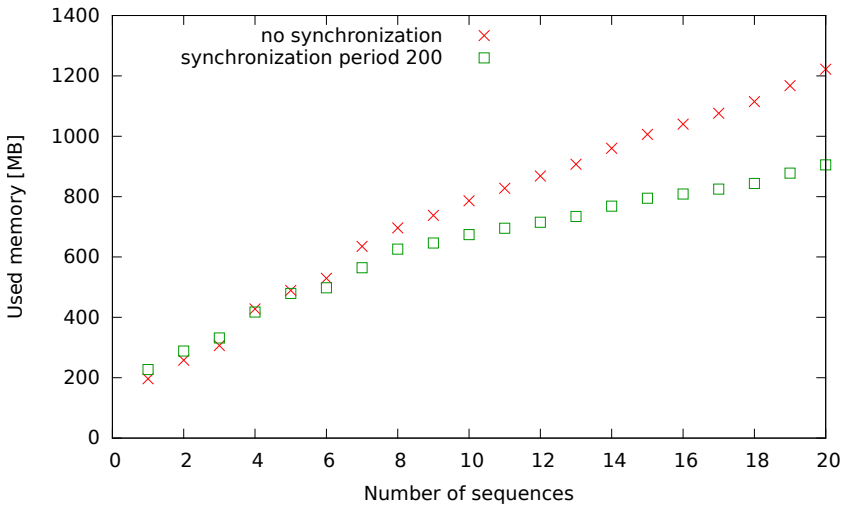

Figure 5.6: Memory consumption with and without explicit synchronization.

limitation, only short prefixes of chromosome 20 (5 % of the original length) were used for this test. The memory savings are enormous – the optimized variant uses more than 10 times less memory than the variant without node collapsing.

Figure 5.6 depicts the difference in memory consumption with and without the synchronization optimization as described in Section 4.1. Significance of the optimization is apparent only when compressing more than five sequences,

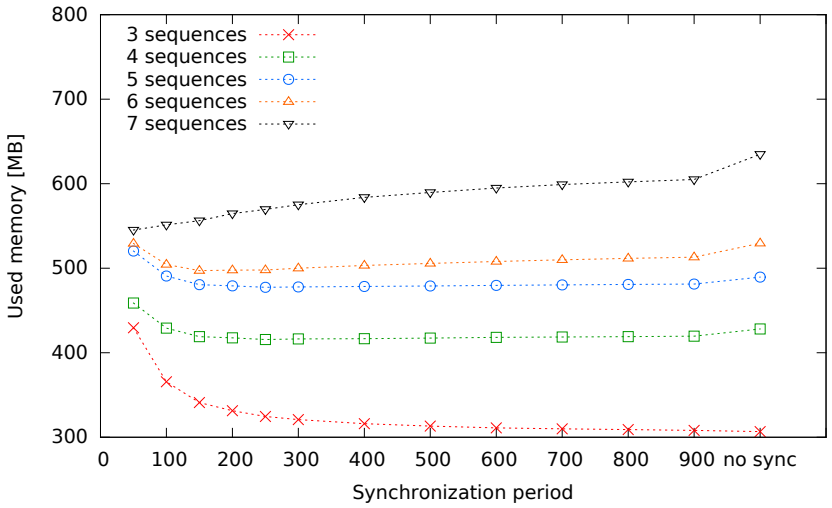

Figure 5.7: Effect of synchronization period on memory consumption.

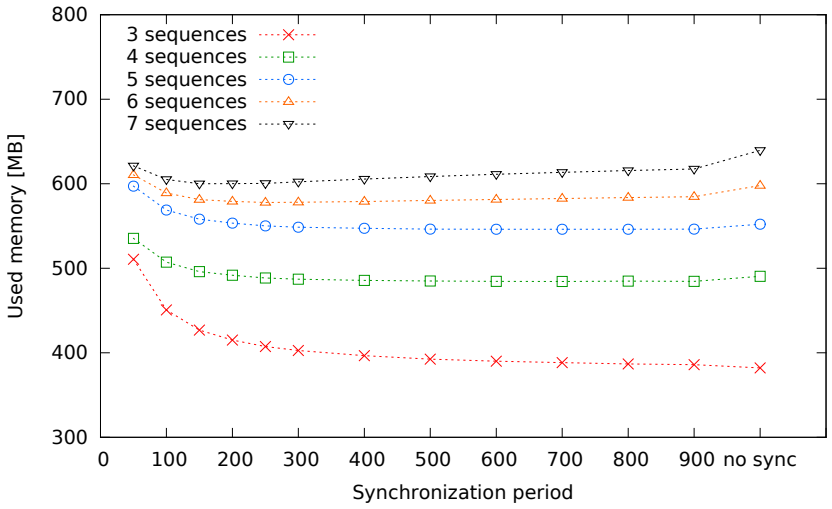

Figure 5.8: Effect of synchronization period on memory consumption (sequences were taken in reverse order).

otherwise the effect is questionable. The memory consumption is even higher with the optimization in case of one, two or three sequences.

Importance of the synchronization period parameter choice is depicted in Figure 5.7. Shortening the synchronization period for small number of compressed sequences increases memory requirements. But starting from four sequences, there is always a synchronization period value with the lowest possible memory consumption. The relation between memory consumption and

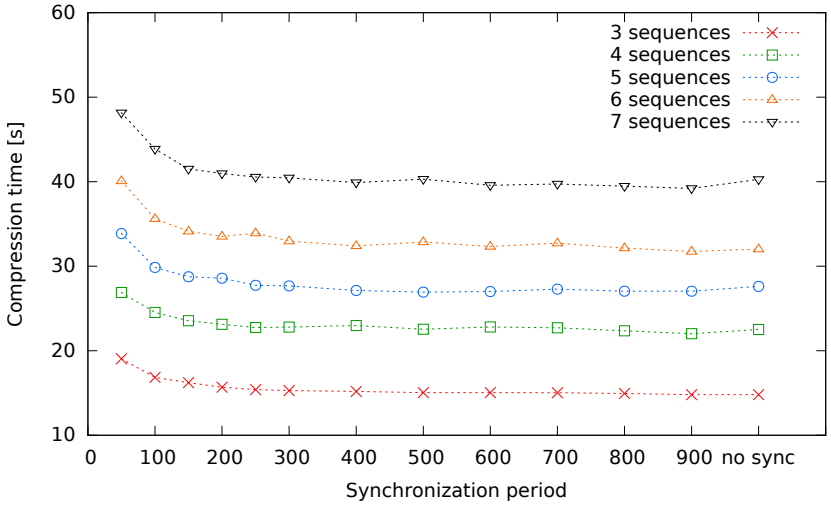

Figure 5.9: Effect of synchronization period on compression time.

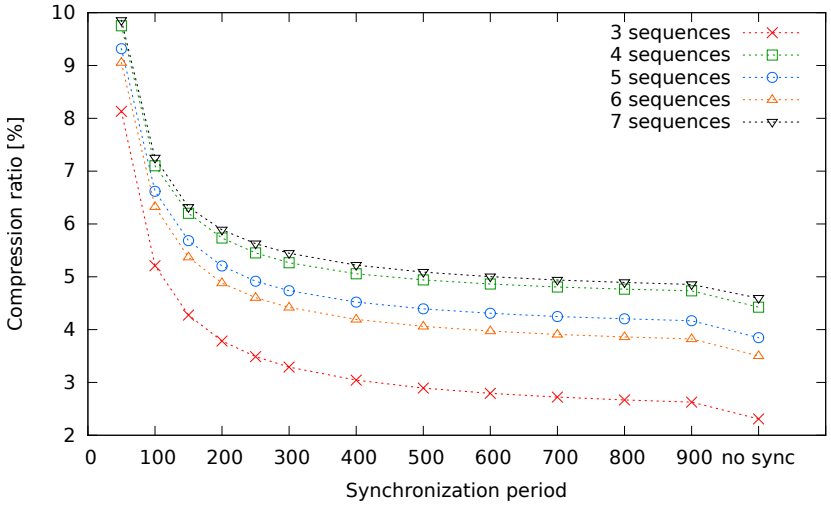

Figure 5.10: Effect of synchronization period on compression ratio.

synchronization period is even strictly increasing for seven sequences. Unfortunately, there is no simple universal rule for choosing the synchronization period parameter. Effect of a particular parameter value is strongly related to given sequences. Even changing the order in which given sequences are compressed may affect behavior of the algorithm. For example, compare Figure 5.7 and 5.8. The same set of sequences was compressed in case of Figure 5.8. The difference is only in order of their compression – they were taken in reverse order in case of Figure 5.8.

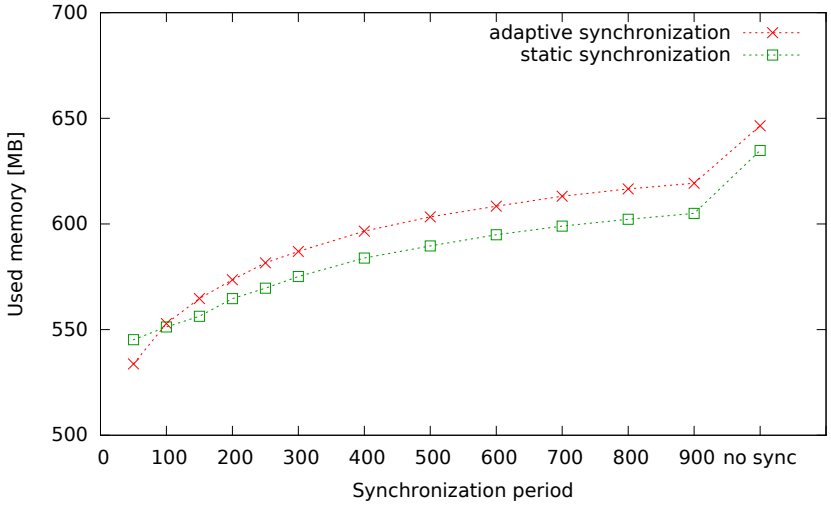

Figure 5.11: Comparison of memory consumption with static and adaptive synchronization.

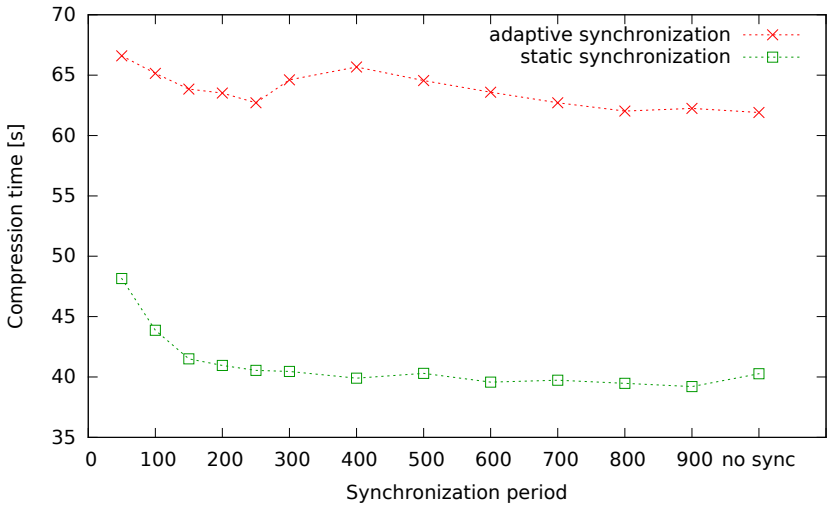

Figure 5.12: Comparison of compression time with static and adaptive synchronization.

Decreasing the synchronization period parameter value has also negative effect on compression time and compression ratio (see Figure 5.9 and 5.10). Considering all the positive and negative effects of the synchronization optimization, a good compromise for compressing sequences of human chromosome 20 would be setting the parameter value between 100 and 300.

Figure 5.11 depicts the difference between static and adaptive synchroniza-

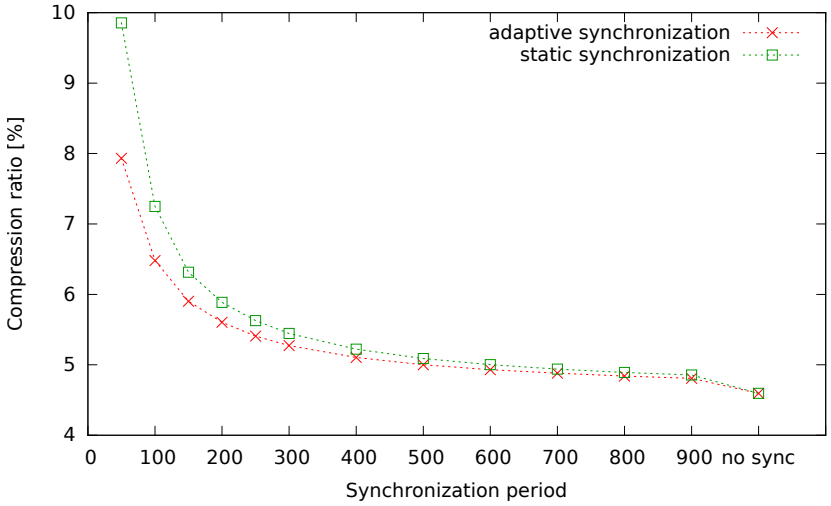

Figure 5.13: Comparison of compression ratio with static and adaptive synchronization.

tion in terms of memory consumption. In case of the adaptive synchronization, the synchronization period parameter denotes the minimum phrase length (i.e. a synchronization point is ignored if length of the current phrase is lower than a given threshold). As implied by the figure, the adaptive synchronization has slightly higher memory requirements than the static version. It is caused by *synchronization map* – a structure storing precomputed synchronization points. This preprocessing step requires one additional pass over a given set of input sequences which also negatively affects compression time as can be seen in Figure 5.12. The only improvement provided by the adaptive synchronization is in compression ratio (see Figure 5.13). It is more significant with shorter synchronization periods. Increasing the synchronization period value gradually diminishes the difference until it vanishes completely.

5.3 Searching

The pattern matching algorithms presented in Chapter 3 are compared in terms of search times in Figure 5.14. Searching was performed on a set of two sequences of human chromosome 20 compressed using ALZW with static synchronization period of 100 bases. Searched patterns were randomly generated. Bad results of the Lahoda-Melichar algorithm for long patterns are caused mainly due to its expensive preprocessing (see Figure 5.15). The algorithm is able to outperform the other three only for short patterns and large datasets as shown in Figure 5.16. This test was performed on a set of twenty sequences of human chromosome 20 compressed using ALZW with

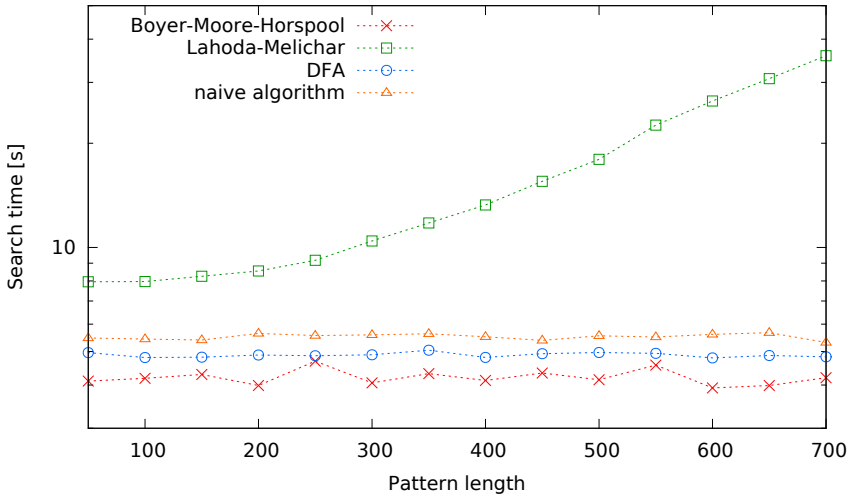

Figure 5.14: Relation between search time and length of a given pattern for the algorithms supported by *alzwq*.

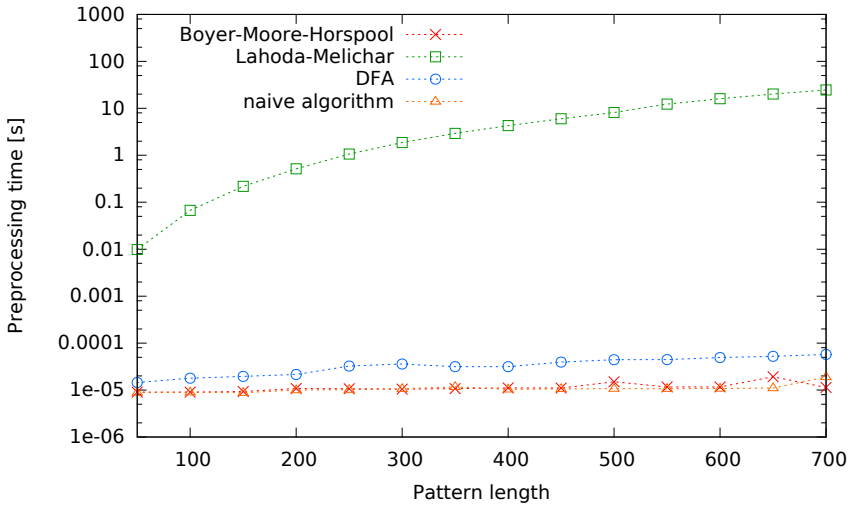

Figure 5.15: Comparison of preprocessing times for the algorithms supported by *alzwq*.

static synchronization period of 100 bases.

Memory consumption of these algorithms is compared in Figure 5.17 and 5.18. The Lahoda-Melichar algorithm exhibits high memory overhead for all pattern lengths. Overhead of the other three algorithms is negligible.

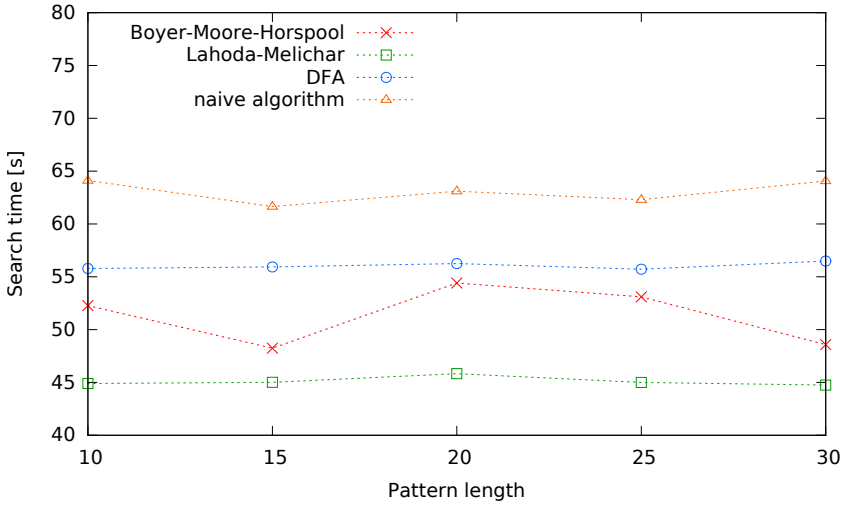

Figure 5.16: Relation between search time and length of a given pattern for the algorithms supported by *alzwq*.

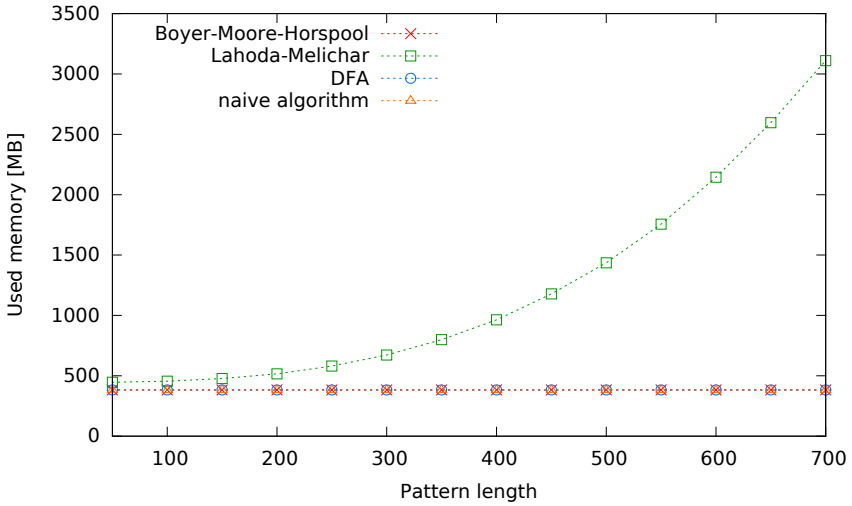

Figure 5.17: Comparison of memory consumption of the algorithms supported by *alzwq*.

5.4 Comparison with other compression methods

Table 5.1 shows compression efficiency of the ALZW algorithm, similarly targeted RLZ and general purpose GZip. Compression efficiency is expressed in *bits per base* (bpb). The first line of the table contains 0-order entropy of the original set of sequences. Using compression ratio would be inappropriate in

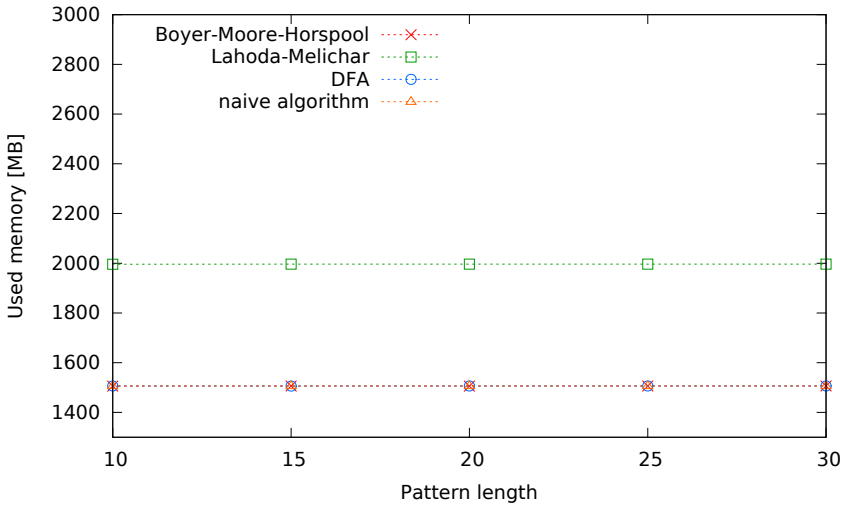

Figure 5.18: Comparison of memory consumption of the algorithms supported by *alzwq*.

Table 5.1: Comparison of different compression methods in terms of compression efficiency.

	Entropy [bpb]	
	S. cerevisiae	Human chromosome 20
original	2.12	2.21
alzw	1.34	**0.36**
rlz	**0.29**	0.38
GZip	2.4	2.29

this case since each algorithm works with different input format. Two datasets were compressed – 39 genomes of Saccharomyces cerevisiae and 21 sequences of human chromosome 20. In both cases, ALZW provides better compression ratio than GZip and it even outperforms RLZ in case of the human chromosome 20 dataset. (Both datasets were compressed without synchronization in order to get the best compression ratio from the ALZW algorithm.)

Times of compression and decompression are compared in Table 5.2. ALZW provides the best compression times from the set of the three algorithms. In case of decompression, the situation is different. Both RLZ and GZip are faster. Memory requirements of the three algorithms in case of compression and decompression are presented in Table 5.3.

Table 5.2: Comparison of different compression methods in terms of compression and decompression times.

	S. cerevisiae		Human chromosome 20	
	Comp. [s]	Decomp. [s]	Comp. [s]	Decomp. [s]
alzw	**40**	13	**112**	55
rlz	63	7	311	39
GZip	100	**5**	237	**11**

Table 5.3: Comparison of different compression methods in terms of memory consumption.

	S. cerevisiae		Human chromosome 20	
	Comp. [MB]	Decomp. [MB]	Comp. [MB]	Decomp. [MB]
alzw	124	155	1222	1510
rlz	42.5	7.8	780	26
GZip	**2.9**	**2.8**	**2.9**	**2.9**

Conclusion

A new compression algorithm called ALZW was developed and implemented as a part of this thesis. The algorithm is designed for compression of sets of highly similar strings and it was tested on two datasets – 21 sequences of human chromosome 20 and 39 sequences of Saccharomyces cerevisiae – and compared with two other compression algorithms – similarly targeted RLZ and general purpose GZip. Despite having quite big memory requirements, the algorithm gives the best compression ratio for the human chromosome 20 dataset and offers the best compression times out of the tested algorithms. The decompression is not as fast as in case of RLZ and GZip but it is still faster than the compression.

Several pattern matching algorithms were discussed in context of the implemented compression algorithm. They were implemented and tested against each other for different lengths of searched patterns. The LM algorithm [19] was good only for short patterns and big datasets. The BMH algorithm [20] provided the best results in all other cases.

Even though the algorithm provides good results (at least in case of compression of human chromosome 20), there is still a lot of space for further optimizations. For example, in encoding of insertions and encoder synchronization. Unlike RLZ, the algorithm also currently does not support efficient random access into compressed files.

Bibliography

[1] Kuruppu, S.; Puglisi, S. J.; Zobel, J. Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval. In *String Processing and Information Retrieval*, *Lecture Notes in Computer Science*, volume 6393, edited by E. Chavez; S. Lonardi, Springer Berlin Heidelberg, 2010, ISBN 978-3-642-16320-3, pp. 201–206.

[2] Kuruppu, S.; Puglisi, S. J.; Zobel, J. Optimized Relative Lempel-Ziv Compression of Genomes. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference*, *ACSC '11*, volume 113, Darlinghurst, Australia: Australian Computer Society, Inc., 2011, ISBN 978-1-920682-93-4, pp. 91–98.

[3] Grabowski, S.; Deorowicz, S. Engineering Relative Compression of Genomes. *CoRR*, volume abs/1103.2351, 2011.

[4] Procházka, P.; Holub, J. Compressing Similar Biological Sequences Using FM-Index. In *Data Compression Conference (DCC), 2014*, IEEE, 2014, pp. 312–321.

[5] Chen, X.; Kwong, S.; Li, M. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proceedings of the fourth annual international conference on Computational molecular biology*, ACM, 2000, p. 107.

[6] Grumbach, S.; Tahi, F. A new challenge for compression algorithms: genetic sequences. *Information Processing & Management*, volume 30, no. 6, 1994: pp. 875–886.

[7] Kuruppu, S.; Beresford-Smith, B.; Conway, T.; et al. Iterative Dictionary Construction for Compression of Large DNA Data Sets. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, volume 9, no. 1, January 2012: pp. 137–149, ISSN 1545-5963.

[8] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, volume 10, February 1966: pp. 707–710.

[9] Welch, T. A. A Technique for High-Performance Data Compression. *Computer*, volume 17, no. 6, June 1984: pp. 8–19, ISSN 0018-9162.

[10] Yokoo, H. Improved variations relating the Ziv-Lempel and Welch-type algorithms for sequential data compression. *Information Theory, IEEE Transactions on*, volume 38, no. 1, January 1992: pp. 73–81, ISSN 0018-9448.

[11] Salomon, D. *Data Compression – The Complete Reference*. Springer New York, 2004, ISBN 978-0-387-40697-8.

[12] Needleman, S. B.; Wunsch, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, volume 48, no. 3, 1970: pp. 443–453, ISSN 0022-2836.

[13] Smith, T.; Waterman, M. Identification of common molecular subsequences. *Journal of Molecular Biology*, volume 147, no. 1, 1981: pp. 195–197, ISSN 0022-2836.

[14] Huang, W.; Umbach, D. M.; Li, L. Accurate anchoring alignment of divergent sequences. *Bioinformatics*, volume 22, no. 1, 2006: pp. 29–34.

[15] Thompson, J. D.; Higgins, D. G.; Gibson, T. J. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, volume 22, no. 22, 1994: pp. 4673–4680.

[16] Altschul, S. F.; Gish, W.; Miller, W.; et al. Basic local alignment search tool. *Journal of Molecular Biology*, volume 215, no. 3, 1990: pp. 403–410, ISSN 0022-2836.

[17] Shen, S.-Y.; Yang, J.; Yao, A.; et al. Super pairwise alignment (SPA): an efficient approach to global alignment for homologous sequences. *Journal of Computational Biology*, volume 9, no. 3, 2002: pp. 477–486.

[18] Elias, P. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, volume 21, no. 2, March 1975: pp. 194–203, ISSN 0018-9448.

[19] Lahoda, J.; Melichar, B. General pattern matching on regular collage system. In *Proceedings of the Prague Stringology Conference '05*, edited by J. Holub; M. Šimánek, Czech Technical University in Prague, Prague Stringology Club, August 2005, ISBN 80-01-03307-4, pp. 153–162.

[20] Horspool, R. N. Practical fast searching in strings. *Software: Practice and Experience*, volume 10, no. 6, 1980: pp. 501–506, ISSN 1097-024X.

[21] Baeza-Yates, R. A.; Régnier, M. Average running time of the Boyer-Moore-Horspool algorithm. *Theoretical Computer Science*, volume 92, no. 1, 1992: pp. 19–31, ISSN 0304-3975.

[22] Aho, A. V.; Hopcroft, J. E.; Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1974, ISBN 978-0201000290.

[23] Kida, T.; Shibata, Y.; Takeda, M.; et al. A unifying framework for compressed pattern matching. In *String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware*, IEEE, 1999, ISBN 0-7695-0268-7, pp. 89–96.

[24] FASTA format. National Center for Biotechnology Information. April 2015. Available from: `http://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml`

[25] The SAM/BAM Format Specification Working Group. *Sequence Alignment/Map Format Specification*. SAMv1 edition, March 2015. Available from: `http://samtools.github.io/hts-specs/SAMv1.pdf`

Appendix A

Data from experimental evaluation

Table A.1: Relation between compression and decompression times, memory consumption and number of sequences.

Number of sequences	Duration [s]		Used memory [MB]	
	compression	decompression	compression	decompression
1	5.232357	4.166167	226.980	205.652
2	10.489771	5.926878	288.184	266.124
3	15.886271	7.801413	331.548	309.372
4	23.050428	13.118687	417.416	481.652
5	28.507743	14.859205	479.152	508.120
6	34.070694	16.471080	497.872	531.796
7	40.928335	22.313617	564.436	696.124
8	46.985495	24.783185	626.048	724.748
9	52.769546	26.544997	646.356	764.504
10	57.971574	28.469935	674.036	794.468
11	63.161655	30.462997	695.148	823.260
12	69.278990	32.220120	715.276	850.248
13	74.357532	34.603957	734.304	898.756
14	81.456574	36.273671	768.248	936.952
15	86.039249	38.510106	794.712	956.876
16	91.300733	40.003135	808.668	979.996
17	96.620971	41.557204	825.020	1006.524
18	102.712915	44.927742	843.692	1055.540
19	109.864520	47.385145	877.816	1094.764
20	115.126815	50.319774	905.400	1170.628

Table A.2: Relation between compression ratio and number of sequences.

Number of sequences	Compression ratio [s]	Overall compression ratio [s]
1	4.297037	15.332279
2	3.853007	11.335698
3	3.785122	9.406061
4	5.734309	9.842657
5	5.204215	8.713066
6	4.880849	7.932703
7	5.887501	8.433080
8	5.605229	7.898346
9	5.461553	7.539098
10	5.295952	7.199168
11	5.164602	6.919747
12	5.039149	6.668613
13	5.045340	6.557809
14	5.009516	6.423370
15	4.901444	6.233497
16	4.813814	6.072516
17	4.751562	5.943676
18	4.773793	5.901936
19	4.756824	5.829354
20	4.890039	5.905171

Table A.3: Comparison of memory consumption with and without node collapsing.

Number of sequences	Used memory [MB]	
	with node collapsing	without node collapsing
1	12.784	144.284
2	16.988	278.512
3	24.316	403.140
4	26.008	503.128
5	32.080	625.992
6	34.988	740.344
7	38.112	834.920
8	41.304	947.724
9	48.252	1061.344
10	48.384	1169.484
11	53.112	1277.568
12	53.128	1378.596
13	57.628	1483.000
14	57.452	1589.240
15	62.372	1691.136
16	62.716	1792.288
17	62.748	1894.248
18	68.368	1999.176
19	68.528	2099.684
20	73.312	2200.476

Table A.4: Memory consumption with and without explicit synchronization.

Number of sequences	Used memory [MB]	
	without synchronization	static sync. period 200
1	196.436	226.980
2	258.056	288.184
3	306.732	331.548
4	427.800	417.416
5	489.348	479.152
6	529.544	497.872
7	635.120	564.436
8	696.440	626.048
9	737.808	646.356
10	786.216	674.036
11	827.776	695.148
12	868.408	715.276
13	907.052	734.304
14	960.112	768.248
15	1006.216	794.712
16	1040.156	808.668
17	1076.432	825.020
18	1114.840	843.692
19	1167.544	877.816
20	1222.112	905.400

Table A.5: Effect of synchronization period on memory consumption.

Sync. period	Used memory [MB] number of sequences:				
	3	4	5	6	7
50	429.568	458.800	520.380	529.316	545.232
100	365.676	429.088	490.764	504.388	551.236
150	341.120	418.980	480.480	497.096	556.312
200	331.440	417.568	479.128	497.684	564.684
250	324.632	415.592	477.460	497.832	569.624
300	320.868	416.356	477.864	499.976	575.116
400	316.076	416.688	478.476	503.336	583.872
500	313.208	417.400	478.928	505.664	589.584
600	311.100	418.164	479.628	507.940	594.912
700	309.892	418.560	480.296	509.960	598.968
800	309.044	419.096	480.868	511.628	602.192
900	308.036	419.584	481.248	513.116	604.976
–	306.612	428.076	489.548	529.612	634.780

Table A.6: Effect of synchronization period on memory consumption (sequences were taken in reverse order).

Sync. period	Used memory [MB] number of sequences: 3	4	5	6	7
50	510.600	535.436	597.132	610.568	621.208
100	450.652	507.264	568.912	589.048	605.088
150	426.836	496.032	558.084	581.220	600.068
200	415.000	491.740	553.348	579.052	600.152
250	407.340	488.560	550.084	577.816	600.372
300	402.652	487.064	548.648	578.088	602.240
400	396.524	485.628	547.244	579.004	605.532
500	392.540	484.876	546.312	580.216	608.592
600	390.044	484.532	546.136	581.416	611.220
700	388.332	484.416	546.120	582.556	613.524
800	386.860	484.744	546.128	583.676	615.672
900	386.008	484.568	546.332	584.684	617.508
–	382.064	490.432	552.168	597.808	639.448

Table A.7: Effect of synchronization period on compression time.

Sync. period	Compression time [s] number of sequences: 3	4	5	6	7
50	19.045675	26.883740	33.870677	40.082402	48.159467
100	16.849937	24.532062	29.864893	35.606864	43.870621
150	16.222682	23.548965	28.753680	34.128656	41.507258
200	15.675598	23.129700	28.577988	33.527908	40.958817
250	15.402325	22.748495	27.740377	33.921067	40.552652
300	15.288032	22.798483	27.671684	32.972081	40.454660
400	15.178841	22.986306	27.123770	32.400232	39.897847
500	15.047755	22.542585	26.932533	32.865916	40.300925
600	15.040655	22.810644	27.009167	32.322766	39.574394
700	15.029614	22.715422	27.288084	32.725876	39.731454
800	14.938725	22.364840	27.045138	32.141055	39.478856
900	14.800013	22.019250	27.040915	31.741205	39.205088
–	14.798581	22.517642	27.614049	32.016678	40.274213

Table A.8: Effect of synchronization period on compression ratio.

Sync. period	Compression ratio [%] number of sequences: 3	4	5	6	7
50	8.131776	9.752009	9.318991	9.054633	9.855125
100	5.209596	7.102150	6.618178	6.323924	7.249267
150	4.274965	6.201733	5.684608	5.369399	6.315548
200	3.785122	5.734309	5.204215	4.880849	5.887501
250	3.488918	5.452650	4.914818	4.604609	5.628303
300	3.290917	5.264798	4.736331	4.419380	5.445148
400	3.042789	5.057867	4.519974	4.193297	5.221519
500	2.893280	4.941537	4.393638	4.060433	5.089726
600	2.793370	4.863389	4.309052	3.971723	5.001723
700	2.721706	4.807506	4.248456	3.908024	4.938669
800	2.668004	4.766477	4.203936	3.861275	4.892238
900	2.627481	4.733169	4.167957	3.823605	4.854923
–	2.309201	4.425506	3.848132	3.502348	4.594432

Table A.9: Comparison of static and adaptive synchronization.

Sync. period	Comp. time [s]		Used memory [MB]		Compression ratio [%]	
	static sync.	adaptive sync.	static sync.	adaptive sync.	static sync.	adaptive sync.
50	48.159467	66.600608	545.232	533.760	9.855125	7.932399
100	43.870621	65.142806	551.236	552.860	7.249267	6.480435
150	41.507258	63.843076	556.312	564.688	6.315548	5.902468
200	40.958817	63.513746	564.684	573.580	5.887501	5.604386
250	40.552652	62.707696	569.624	581.564	5.628303	5.408246
300	40.454660	64.608661	575.116	586.980	5.445148	5.272608
400	39.897847	65.672198	583.872	596.600	5.221519	5.104122
500	40.300925	64.551148	589.584	603.372	5.089726	5.000363
600	39.574394	63.584640	594.912	608.408	5.001723	4.930036
700	39.731454	62.707087	598.968	613.124	4.938669	4.880510
800	39.478856	62.027102	602.192	616.612	4.892238	4.840387
900	39.205088	62.241852	604.976	619.264	4.854923	4.807121
–	40.274213	61.910243	634.780	646.460	4.594432	4.594432

Table A.10: Relation between overall search time, preprocessing time and length of a given pattern for the algorithms supported by *alzwq*.

Pattern length	Overall search time [s]				Preprocessing time [ms]			
	BMH	LM	DFA	naive	BMH	LM	DFA	naive
20 sequences of human chromosome 20 (static sync. period of 100 bases):								
10	52.270	44.894	55.789	64.108	0.009	0.182	0.011	0.009
15	48.239	45.009	55.935	61.650	0.009	0.459	0.012	0.009
20	54.404	45.833	56.251	63.098	0.009	0.866	0.012	0.009
25	53.102	44.997	55.715	62.300	0.009	1.515	0.012	0.009
30	48.581	44.746	56.486	64.064	0.009	2.475	0.012	0.009
2 sequences of human chromosome 20 (static sync. period of 100 bases):								
50	4.109	7.953	4.963	5.470	0.009	9.881	0.014	0.009
100	4.182	7.970	4.804	5.433	0.009	67.190	0.018	0.009
150	4.290	8.252	4.814	5.400	0.009	218.646	0.020	0.009
200	3.988	8.546	4.885	5.642	0.011	514.817	0.021	0.010
250	4.689	9.176	4.861	5.562	0.011	1064.238	0.033	0.010
300	4.059	10.438	4.892	5.585	0.010	1869.187	0.036	0.011
350	4.317	11.772	5.042	5.627	0.011	2932.223	0.031	0.012
400	4.123	13.272	4.806	5.517	0.011	4276.924	0.032	0.010
450	4.332	15.533	4.930	5.385	0.011	6014.270	0.039	0.011
500	4.145	17.971	4.969	5.550	0.015	8155.124	0.044	0.011
550	4.569	22.567	4.939	5.507	0.012	12321.230	0.044	0.011
600	3.923	26.504	4.792	5.607	0.012	16017.207	0.049	0.011
650	3.988	30.753	4.869	5.666	0.019	20127.441	0.052	0.011
700	4.200	35.832	4.826	5.313	0.011	24634.207	0.057	0.019

Table A.11: Comparison of memory consumption of the algorithms supported by *alzwq*.

Pattern length	Used memory [MB]			
	BMH	LM	DFA	naive
20 sequences of human chromosome 20 (static sync. period of 100 bases):				
10	1505.940	1996.088	1506.172	1506.236
15	1506.172	1996.236	1505.940	1506.108
20	1505.940	1996.228	1506.108	1506.024
25	1506.032	1996.368	1506.000	1506.040
30	1506.108	1996.464	1506.072	1506.024
2 sequences of human chromosome 20 (static sync. period of 100 bases):				
50	382.752	446.884	382.680	382.672
100	382.644	455.496	382.676	382.672
150	382.672	477.316	382.712	382.756
200	382.760	516.448	382.772	382.680
250	382.672	580.360	382.680	382.648
300	382.592	672.048	382.684	382.692
350	382.644	799.100	382.600	382.644
400	382.588	963.056	382.648	382.588
450	382.688	1177.924	382.768	382.680
500	382.640	1436.168	382.668	382.820
550	382.648	1756.364	382.908	382.760
600	382.768	2144.400	382.704	382.592
650	382.772	2596.220	382.668	382.592
700	382.640	3110.676	382.800	382.772

Appendix B

FASTA format

FASTA format, as defined in [24], is a text-based format for storing nucleotide and protein sequences. One file may contain multiple sequences. Each sequence consists of a description line and the sequence itself. The sequence is represented by the standard IUPAC codes. According to recommendation, all lines should be at most 80 characters long. Empty lines are not allowed. Lower-case letters in a sequence are treated as upper-case. Formal specification of the format in EBNF follows:

```
fasta       = { record } ;
record      = description , "\n" , sequence ;
description = ">" , { letter } ;
letter      = any ASCII character except "\n" ;
sequence    = { segment , "\n" } ;
segment     = { base } ;
base        = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
            | "I" | "K" | "L" | "M" | "N" | "P" | "Q" | "R"
            | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
            | "*" | "-" ;
```

There is no difference between regular FASTA format and aligned FASTA format except the fact it contains at least two sequences. The sequences must be aligned to a same length using `"-"` symbols (insertions and deletions).

Appendix C

SAM format

SAM stands for Sequence Alignment/Map and it is a format for storing large alignments of genomic sequences. The format is used by 1000 Genomes project. It contains short reads generated by a sequencer and aligned to a reference sequence. There is also a lot of metadata (mapping quality, etc.). Complete description of the format is beyond the scope of this text, see [25] for more details. There are two variants of the format – text-based and binary. Both of them can be read using *samtools* C library.

Unfortunately, a complete sequence or an alignment cannot be read easily. The format can be read only in form of the short aligned reads. Each position of the complete alignment may be covered by multiple reads. Moreover, the reads may contain different symbols for a particular position. An easy solution was employed in order to resolve this problem in *sam2fasta* and *sam2seq* tools. All reads covering a particular position are compared according to their mapping quality and symbol from a read with the best mapping quality is used (see Figure C.1). This might not be an ideal solution from the biological point of view, but it is applicable for testing the compression algorithm.

```
                                                          quality
ref:       A A C G A C G A C T T G A C T A A G G A T G T G A A A

read #1:         G G A G A C T T G T                          5 %
read #2:           C G A A T T                               70 %
read #3:                 C T T G A C A                      50 %

seq:       ? ? ? G G C G A A T T G A C A ? ? ? ? ? ? ? ? ? ? ? ?
```

Figure C.1: Example of a SAM alignment coverage.

Appendix D

Acronyms

ALZW Alignment-based LZW compression algorithm

AVL Adelson-Velsky and Landis tree

BMH Boyer-Moore-Horspool algorithm

DFA Deterministic Finite Automaton

EBNF Extended Backus-Naur Form

FA Finite Automaton

HSS Highly Similar Strings

LM Lahoda-Melichar algorithm

LZ77 Lempel-Ziv 1977 compression algorithm

LZAP Lempel-Ziv All Prefixes compression algorithm

LZMW Lempel-Ziv-Miller-Wegman compression algorithm

LZY Lempel-Ziv-Yokoo compression algorithm

LZW Lempel-Ziv-Welch compression algorithm

RB Red-Black tree

RLZ Relative Lempel-Ziv compression algorithm

SNP Single Nucleotide Polymorphism

Appendix E

Contents of enclosed DVD

- `alzw.testing.env.ova` .. VirtualBox image with the testing environment
- `readme.txt` ..... the file with DVD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `alzw` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format