CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Implementation of the ACB compression method improvements in the Java language
Student:	Bc. Jiří Bican
Supervisor:	Ing. Radomír Polách
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2016/17

Instructions

Study the ACB compression method, its implementations, and effective structures for its index. Focus on the order statistic tree and other binary search tree structures.
Choose and implement an index structure and create an effective ACB compression algorithm with good entropy coding (arithmetic coding or range coding) in the Java language. Analyse and implement various variants of this algorithm given by your supervisor. Test and compare these variants based on time and space complexity and compression ratios on various corpuses.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 9, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Implementation of the ACB compression method improvements in the Java language

Bc. Jiří Bican

Supervisor: Ing. Radomír Polách

10th January 2017

Acknowledgements

I would like to thank my supervisor, Ing. Radomír Polách, for the idea and his helpful advices regarding this thesis. I would also like to thank my family for their support during my study.

Declaration

In Prague on 10th January 2017 . . . .

Citation of this thesis

Bican, Jiří. *Implementation of the ACB compression method improvements in the Java language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato práce se zabývá rozborem kompresního algoritmu ACB. V práci je popsán jak samotný algoritmus, tak všechny jeho předchozí implementace a vylepšení. Předmětem práce bylo analyzovat datové struktury použitelné pro slovníkové indexování s ohledem na rychlost a kompresní poměr algoritmu, navrhnout možná slovníková vylepšení a to všechno implementovat v jazyce java, ve kterém algoritmus ACB zatím neexistoval. Implementační vylepšení jsou podložena experimentálním měřením.

Klíčová slova Algoritmus ACB, kontext, content, komprese, dekomprese, Java.

Abstract

The subject of this thesis is to analyse the compression algorithm ACB and implement it in Java language, creating probably the first ACB Java implementation. It covers the basic logic of the algorithm, as well as all previous implementations and improvements. The goal is to analyse possible data structure variants for indexing the dictionary with respect to time and compression ratio, and suggest possible dictionary improvements. All improvements are verified by experimental measurements.

Keywords ACB algorithm, contextm content, compression, decompression, Java.

Contents

List of Figures

List of Tables

Introduction

Motivation

Compression methods have been a very popular branch of computer science ever since the storing of large amounts of data became an issue. There have been many compression algorithms invented, described and implemented. Few of them were so efficient that they became standardized in many more sophisticated softwares because of their reliability, speed and compression ratio. Such algorithms are for example *LZW*, *PPM*, *Huffman coding*, *LZ77*, *LZ78* or *Shannon-Fano*. They can be divided into groups of statistical methods, dictionary-based methods and context-based methods. Those mentioned are well known and frequently used, but there exist countless other compression methods. These might not be popular either because they are inefficient or for some other restriction, such as absence of parallelization or that no convenient data structure was known.

ACB is an unexplored context-based dictionary compression method. It is called Associative Coder of Buyanovsky or in short ACB, published by George Buyanovsky in 1994 [1]. The article was written in Russian, and the compression method was called Associative coding. George Buyanovsky also created an implementation of this algorithm, but distributed it only as an MS-DOS executable. The ACB method might have huge potential that has not yet been discovered but with the arrival of modern technologies and approaches, this potential may yet be fulfilled.

ACB method might have huge potential that have not been discovered yet, but with the arrival of modern technologies and approaches, this potential might be fulfilled.

Main goals

The main goal of this thesis is to implement the ACB algorithm in Java language along with all known improvements and adjustments. The first task is to understand the basics of the algorithm, then do research into all existing implementations. The next challenge is to find and implement a data structure which will be efficient enough to serve as a dictionary with fast searches, fast updates and be able to grow in size without unnecessary effort. The next task is to find the best way of storing data into the output files, as the raw ACB input is far too space consuming.

Thesis organization

This chapter introduces and explains the motivation and main goals of the thesis.

Chapter 1 defines the basic terms used in the field of compression methods. It also describes the basic algorithm design of ACB, providing some examples for the better understanding of algorithm compression and decompression steps. It also describes the basic ideas behind common improvements.

Chapter 2 is devoted to describing currently existing implementations.

Chapter 3 analyses the algorithm introduced in Chapter 1 including dictionary structure analysis: in other words which data structures to use and how to modify them to bring about the best result. There is also a section describing ACB output representation and improvements to the dictionary.

Chapter 4 describes in detail how the Java version of ACB compression method is implemented, what issues were faced and how they were solved.

Chapter 5 shows the results of experiments and performance measurements.

Chapter 1

Algorithm

This chapter focuses on describing the design of the *Associative Coder of Buyanovsky (ACB)* compression method, the details of its dictionary structure and compression and decompression process in general. Further details about particular data structures and algorithm implementations made in this thesis are described in detail in Chapter 3. Most of this chapter is referencing to description published in "*Data Compression: The Complete Reference*" by *David Salomon*, [2].

1.1 Definitions

Definition 1.1.1 (Alphabet)
Alphabet is a finite set of symbols.

Definition 1.1.2 (Symbol)
Symbol is an atomic element of an alphabet, usually one byte.

Definition 1.1.3 (String)
String is a sequence of symbols from alphabet.

Definition 1.1.4 (Empty string)
Empty string is the string with zero length and is denoted as ϵ.

Definition 1.1.5 (Codeword)
Codeword is a sequence of bits.

Definition 1.1.6 (Code)
Code K is an ordered triplet $K = (S, C, f)$, where

- S is a finite set of string,
- C is a finite set of codewords,

- f is an injective mapping $S \to C^{+}, \forall s_1, s_2 \in S, s_1 \neq s_2 \Rightarrow f(s_1) \neq f(s_2)$.

By C^{+} we denote the set of all codewords with non-zero length containing only symbols from the set C.

Definition 1.1.7 (Uniquely decodable code)
Uniquely decodable code is defined as

$$\forall u_1, u_2 \in C, u_1 \neq u_2 \Rightarrow f^{-1}(u_1) \neq f^{-1}(u_2)$$

Also codeword $u \in C^{+}$ is uniquely decodable with respect to f, if there is at most one sequence $s \in S^{+}$ such that $f(s) = u$.

Definition 1.1.8 (Prefix code)
Prefix code is such a set C^{+} of codewords, where no codeword is a prefix of another codeword. These codes belong to uniquely decodable codes. Prefix codes are often used for their unique ability while decoding - reading from left to right.

Definition 1.1.9 (Compression)
Compression is defined as

$$f(s) = u; s \in S, u \in C$$

Definition 1.1.10 (Decompression)
Decompression is defined as

$$f^{-}1(u) = s; s \in S, u \in C$$

Definition 1.1.11 (Compression ratio)
Compression ratio is the ratio between the compressed

le size and the original file size. Its value is computed using Equation 1.1.

$$compressionratio = \frac{compressedsize}{originalsize} \tag{1.1}$$

Result of Equation 1.1 is a real number. If its value is lower than 1.0, the compressed file is smaller than the original file. If the value is greater than 1.0, the compressed file is actually greater than the original file, or in other words, compression of this file resulted in negative compression.

1.2 Basic overview

Following definitions are important for describing the basic behaviour of ACB compression method. There are ASCII characters used as an alphabet in these examples, but it could be any binary data in general. Because of that, we talk about text, but it could be any input binary sequence.

1.2.1 Structure

Every term t_i in the given text can be represented as a position in the text. Let any substring α in the left vicinity from this position is called **context** and any substring β in the right vicinity (including t_i) is called **content** of term t_i. Let the pair $I_i = (\alpha, \beta)$ be **context item** of term t_i. If context or content of context item is limited by a number of terms, then the context item is bound. Any subset of all possible k-bound context items of the text form k-bound **context item collection**.

Structure of context dictionary adds a complete order to the context item collection. The order can be described that the context item I is smaller then context item J if α_I is smaller using reverse comparison then α_J. Reverse comparison means right-to-left. Comparison of individual terms is interpretation-dependent.
Formally holds: $\alpha_I < \alpha_J \longrightarrow I < J$

Aside from context dictionary, ACB uses sliding window that can be also found in some dictionary methods such as LZ77. Sliding window is always located at the particular index of the text, left side from the index is called context and right part content. This is shown below.

context $\Leftarrow$ $\boxed{\text{...swiss m|iss is missing...}}$ $\Rightarrow$ content

At the beginning of encoding/decoding, the runtime context dictionary is empty. But if we consider sliding window index to be as shown above, the context dictionary would look like shown in Table 1.1.

Table 1.1: ACB dictionary

Context	Content	Context	Content
	swiss miss		swiss miss
s	wiss miss	swiss	miss
sw	iss miss	swi	ss miss
swi	ss miss	s	wiss miss
swis	s miss	swis	s miss
swiss	miss	swiss	miss
swiss	miss	sw	iss miss

1.2.2 Compression

The compression algorithm of the ACB compression method is explained in Algorithm 1. The algorithm runs in cycle from the beginning of the input file to the end. Position is represented as an index *idx*. In the body of the cycle, the first step is to find the best matching dictionary context to the current

Algorithm 1 ACB compression algorithm

```
1: procedure ACBCompress
2:     $idx \leftarrow 0$
3:     while $i < FileSize$ do
4:         $ctx \leftarrow$ best matching context for current context
5:         $cnt \leftarrow$ best matching content for current content
6:         $l \leftarrow$ common length of best matching content and current content
7:         $c \leftarrow$ first non matching symbol of current content
8:         Output($cnt - ctx, l, c$)
9:         UpdateDictionary($idx, l + 1$)
10:        $idx \leftarrow idx + l + 1$
11:    end while
12: end procedure
```

context (left from index position). The dictionary is sorted by contexts, so search is not linear, binary search can be used. As a second step, algorithm finds best matching dictionary content to the current content. Current content is right from the index and it is data to be encoded, while current context is data already encoded. The dictionary contents are unfortunately not sorted in any way and search have to be done in linear time. As dictionary grows in size, this becomes big disadvantage. As shown in Chapter 3, this is solved by searching in limited surrounding.

After both best matching context and content are found, this data are sent to output. Output is called the triplet and its particular form may vary, but in the original implementation it consists of three parts as shown in the Example 1.2. The first part is the distance between the best matching context and content in the dictionary. The second part is the length of common prefix of current content and the best dictionary content. The third part is the first non matching symbol of current content.

$$(cnt - ctx, l, c) \tag{1.2}$$

After the triplet is put to the output, the index has to be moved to the new position and the dictionary have to be updated. New index is counted by adding length + 1 for the extra encoded symbol. And dictionary is updated for the same amount of context pairs.

For better understanding of the ACB encoding method the simple example is presented.

Example 1.2.1
Compress text S using ACB compression algorithm.

$$S = mississippi$$

The compression starts at index 0 with empty dictionary.

```
|mississippi
```

Step 1

Empty dictionary makes searching part of the algorithm pointless, it could be skipped and the first output triplet is

$$(0, 0, m)$$

Encoded length of the text is 0 plus one character contained in the triplet. Updating the dictionary with this triplet extends it by $l + 1 = 1$ new lines, after an update it looks like presented below where ε represents an *empty*

```
 Context|Content
0       ε|m
```

context or an *empty string.* Now we move the pointer in the input data by l + 1 symbols, in this case by 1. After the

rst encoded symbol the sliding window looks like

```
m|ississippi
```

Step 2

Algorithm first searches for the best matching context. There is only one item in the dictionary and it's index is 0. The content-searching part leads to the same result, best matching content index is 0 too.

Common length between best matching content at index 0 and actual content of string S is 0. The first non matching symbol of the current content

```
S:  m|ississippi
0:  ε|m
```

is i. The next triplet sent to the encoding output is

$$(0, 0, i)$$

The dictionary updated by this triplet looks then like Notice that the vocab-

```
 Context|Content
0       ε|mi
1       m|i
```

ulary is sorted right-to-left by contexts every time it is updated. ε stands for the empty string is in lexicographical ordering lower than any other symbol so it is always at position 0. After this step, the sliding window looks like

```
mi|ssissippi
```

Step 3

Current content, which is mi, is greater than ε at index 0 in reverse order comparison, but lesser than m at index 1. None of them have any symbol in common so both can be considered as the best match. It is up to the implementation which one to pick, but the decision must be deterministic and same as for decoding, otherwise it will not produce the same data.

This example choose the greater index, 1 in this particular case. Search for the best matching content gives the same result as in the step 2, and so the next output triplet is

$$(0, 0, s)$$

Then the dictionary is updated and sliding window shifted.

```
  Context|Content
0       ε|mis
1      mi|s
2       m|is
```

```
mis|sissippi
```

Step 4

Last symbol of the current context is lexicographically bigger than the last context in the dictionary. According to the agreement from previous step that the algorithm always picks bigger index, it would normally pick context at index 3, but because there is not any, it picks content index 2.

Content comparison goes much easier here, as there is one common symbol 's' with content at index 1. That makes common length to be 1, after that goes first non matching symbol which is 'i' and resulting triplet is

$$(cnt - ctx, l, c) = (1 - 2, 1, i) = (-1, 1, i)$$

Updating dictionary adds there two new records, one for common length and one representing symbol in triplet. There should always be all the encoded symbols in the dictionary.

```
missi|ssippi
```

Step 5

Context index is 2 in this step. There is one common letter with context at index 1, but it still relies between 1 and 2 and bigger index is chosen. Content

```
  Context|Content
0       ε|missi
1      mi|ssi
2       m|issi
3     mis|si
4    miss|i
```

index is easily found to be 1 with three symbols in common. Next symbol is 'p'. Output triplet and updated dictionary, that grew by 4 symbols are following along with sliding window.

$$(2-1,3,p)=(1,3,p)$$

```
     Context|Content
0          ε|mississip
1         mi|ssissip
2      missi|ssip
3   mississi|p
4          m|ississip
5        mis|sissip
6     missis|sip
7       miss|issip
8    mississ|ip
```

mississip|pi

Step 6

The best matching context index is 5 and the best matching content is index 3. The common length of the contents is 1 symbol. The first non matching symbol is 'i'. The last output triplet, final dictionary and sliding window are

$$(3-5,1,i)=(-2,1,i)$$

mississippi|

After all the six steps we compressed the text

$$S = mississippi$$

into

$$(0,0,m),(0,0,i),(0,0,s),(-1,1,i),(1,3,p),(-2,1,i) \tag{1.3}$$

	Context	Content
0	ε	mississip
1	mi	ssissippi
2	missi	ssippi
3	mississi	ppi
4	m	ississippi
5	mississip	pi
6	mississipp	i
7	mis	sissippi
8	missis	sippi
9	miss	issippi
10	mississ	ippi

1.2.3 Decompression

The decompression algorithm of the ACB compression method is explained in Algorithm 2.

Algorithm 2 ACB decompression algorithm

```
procedure ACBDecompress
    $idx \leftarrow 0$
    while !EOF do
        $d \leftarrow$ read distance from input triplet
        $l \leftarrow$ read common length from input triplet
        $c \leftarrow$ read next symbol from input triplet
        $ctx \leftarrow$ best matching context for current context
        $cnt \leftarrow d + ctx$
        $s \leftarrow$Copy$(cnt, l)$ + $c$
        Output$(s)$
        UpdateDictionary$(idx, l + 1)$
        $idx \leftarrow idx + l + 1$
    end while
end procedure
```

Decompression algorithm reads data from previously compressed file and operates until there are some. It has stream of triplets as an input. Particular representations of triplets in binary data is further discussed in Chapter 3. Triplet consists of three data parts, d as distance, l as length of encoded data and c as another symbol.

$$(d, l, c)$$

One triplet is read in every step of the cycle and three variables are initialized. Then the best matching context to the current context have to be found. As a current context we consider already decoded data. So, speaking about sliding window as in compression, it is always at the end of the decoded data. Best matching context is found by the same method as in compression, reversed comparison. There is not any common content which means it has to be computed. Best common context *ctx* is known as well as distance *d* in the dictionary of the best matching content from this context, it is counted using the equation

$$cnt \leftarrow d + ctx \tag{1.4}$$

Then l symbols are taken from content at index *cnt* and they are copied to the output along with symbol c at the end. As a last step, the dictionary have to be updated. This works the same way as for compression, $l + 1$ new dictionary items are added while keeping it's context-content pairs sorted in reverse context comparison.

This process is demonstrated by the following example.

Example 1.2.2
Decompress input triplets using ACB decompression algorithm.

$$(0, 0, m), (0, 0, i), (0, 0, s), (-1, 1, i), (1, 3, p), (-2, 1, i)$$

As always, the algorithm starts at index 0 with empty dictionary, but also empty sliding window. Thing to notice is that the dictionary and the sliding window looks exactly the same after each step.

Step 1

The first triplet is read from the input stream. There is no context at the moment, but more importantly this triplet has common length l equal to zero, so searching part of the algorithm is skipped and symbol c = 'm' is sent to the output. The dictionary is updated by this symbol and sliding window is shifted, it looks like

```
   Context|Content
0        ε|m
```

```
m|
```

Step 2, Step 3

In case of second and third triplet, there is the same situation as in step 1, it has common length l equal to zero so no context comparison is made and symbols are directly send to output. The updated dictionary and sliding window looks like

```
mis|
  Context|Content
0       ε|mis
1      mi|s
2       m|is
```

Step 4

The next input triplet is

$$(d, l, c) = (-1, 1, i)$$

l greater than zero means there is encoded more than one character. Best context for current context `mis` is at index 2. This information is enough to find content that was used for encoding, we follow formula

$$cnt = d + ctx = -1 + 2 = 1$$

The best content is the content with index 1: `s`. As we have the best matching content, we can output $l = 1$ common symbols to the output and add the next non matching symbol $c =$ 'i'. After updating, the dictionary and sliding window looks like

```
  Context|Content
0       ε|missi
1      mi|ssi
2       m|issi
3     mis|si
4    miss|i
```

```
missi|
```

Step 5

This step and the last one are very similar to Step 4. For next triplet

$$(1, 3, p)$$

is the best matching context $ctx = 2$, distance $d = 1$ and using the equation 1.4 the content index is $cnt = 1$. Then $l = 3$ symbols are copied and the following symbol is $c =$ 'p', `ssip` is added to the output and resulting dictionary is

```
mississip|
```

Step 6

$$(-2, 1, i)$$

```
        Context|Content
0             ε|mississip
1            mi|ssissip
2         missi|ssip
3      mississi|p
4             m|ississip
5           mis|sissip
6        missis|sip
7          miss|issip
8       mississ|ip
```

The best matching context for this triplet is $ctx = 5$, distance $d =$ -2, using equation 1.4 we get $cnt = 3$ so we copy $l = 1$ to the output plus last symbol $c =$ 'i'.

After those six steps, the text S is decoded

$$S = mississippi$$

that is equal to the original input string in the Example 1.2.1.

1.3 Modifications

As well as basic compression and decompression method, there are some modifications to the ACB algorithm presented in Salomon's book [2]. For completeness they are stated in this section.

1.3.1 Variable output

As the compression or decompression algorithm process, more context-content pairs are added to the dictionary which is increasing the chance that dictionary content with as much as possible common symbols to the current content will be found. But if there is no such content found, only non matching symbol is sent to the output and zero distance and zero length are wasting space in the triplet. The triplet will actually increase the compression ration. Same thing could be seen at every beginning of the compression process, where new symbols are added to the dictionary.

There is proposed a modification in Salomon [2] to create two different output cases, one containing non matching symbols alone, and the second one only distance and common length. To distinguish between them, a bit flag is needed. So the triplet output looks like

$$(0, c) \quad or \quad (1, d, l)$$

1.3.2 Enable bigger common length than best content length

This modification says that if it is possible to copy more symbols from the best matching content after the dictionary update, it should be done. That is solved simply by overlapping buffers, best content comparison continues in common content if the temporary end is reached. By allowing comparison of the contents beyond the best matching content bounds, we can actually improve the performance, as there is no need to check the bounds.

This modification is explained on very simple example.

Example 1.3.1
How would the output look like using modification 1.3.2

$$S = sssss$$

Solution:

The compression always begins the same way, first symbol have to be added to the dictionary and is sent to the output.

$$(0, 0, s)$$

```
  Context|Content
0        ε|s
```

```
s|ssss
```

In the next step, both best matching context and content are equal as there is no other dictionary pair, $ctx = cnt = 0$. Distance is obviously $d = 0$ and common length would be 1, but because there is `'s'` following to both current content and best content, and best content would be updated by this symbol after this step anyway, it can be taken into account right now and make common length as long as best content and common content match.

The next and last triplet carries all the same remaining symbols in it.

$$(0, 3, s)$$

For decoding, the process of overlapping buffers have to be simulated. The trick is to continuously copy data between best content buffer and common content buffer.

1.3.3 Sorted contents

Last presented modification in Salomon's book [2] is sorting contents for common contexts, but it is not analysed here in detail. It is mentioned later in Section 2.5 that this modification brings no compression ratio advantage and runs longer. Basic idea behind it is that:

1. Contents for common context are sorted.
2. Among there contents the search of best matching one is done using the binary search.

Chapter 2

Related work

This chapter discuss existing implementations of ACB algorithm and describes their behaviour.

Due to the fact that ACB compression method is not very spread, there are not many implementations available on the internet. All those that could be found are stated here with their description and distinctions from each other.

2.1 George Buyanovsky

The original author of ACB compression and decompression method is George Buyanovsky. ACB stands for associative coder of Buyanovsky. This algorithm was presented in Russian popular magazine [1] in 1994. It also contained the source code, but it was distributed as a shareware MS-DOS executable. Part of it is written in assembler, the rest is written in C. Even though there was this source code available, the exact algoritmization and functionality is unknown. Another source code was also found at [3], but the license is unknown. Code itself is also very hard to understand, but it is obvious that an arithmetic coding is used for coding lengths and symbols, but not distances, as they are impossible to predict and could take any value from context vicinity. Another detail is that probabilities of lengths are initialized, so values 0 to 6 have higher probability than others, precisely

$$b[\,] = \{45, 13, 10, 7, 5, 4\}$$

Application was released with a readme file with description how to use the program and what is program capable of. Those capabilities are:

- The program can not only compress a single file, but is also able to create containers with multiple files or multi-volume archives.

- The program is capable of usage of the context files - here named "TAUGHT CHANNEL" - MODE. Using the "TAUGHT CHANNEL" creates a context file, which size is limited to 1.5 - 1.9 MB.

- The memory consumption during the compression and the decompression is limited to 16MB, and the maximum size of file is limited to 64MB.

- The program is capable of password protection and error protection.

2.2 Martin Děcký

Another implementation was made at Faculty of Mathematics and Physics at Charles University in Prague by Martin Děcký in 2006. It is distributed under the GNU GPL licence. He made it as a proof of concept project to demonstrate whether it works or not, the performance and compression ratio were not big issue. It is written in C++.

Dictionary is represented by standard library list, search is linear using standard library iterator and dictionary is never cleared. That leads to increasing time complexity for big input files and also big memory overhead. If the growing dictionary buffer reaches maximum allocated memory size, exception is thrown and compression/decompression is abandoned. Triplets are not coded in any way, they are outputted as a bit stream. But triplets takes multiple forms.

2.3 Lukáš Cerman

This implementation was made by Lukáš Cerman in his bachelor thesis [4] in 2003. It was implemented in Microsoft Visual Studio in C.

The dictionary is implemented in two dimensional array. First dimension's index is computed as a hash from the first two symbols of the context. It means hash function is used to access the other array dimension in constant time. The second dimension is sorted array of contexts. Binary search is used here with complexity of $O(\log n)$. The oldest item is removed if the dictionary happens to be full, it is checked before each insertion.

Triplets are encoded in two possible ways, Huffman coder is the default one, another option is range coder. Triplets takes these variations:

- If l >0, (l, d) is sent to output, (0, c) otherwise.

- If l >0, (l + |Σ| - 1, d) is sent to output, (c) otherwise.

- If l >0, (|Σ|, l - 1, d) is sent to output, (c) otherwise.

There are two stages of the compression. At first, input file is encoded by ACB producing raw triplets stored in separate temporary file. This phase could be modified by specifying used alphabet, there are two possibilities present, one which groups the upper case and the lower case symbols together, second which groups the vowels and the vowels with punctuations together. Then the temporary file is encoded using the Huffman or range encoding, according to the triplets in temporary file.

2.4 Filip Šimek

The implementation from Filip Šimek was distributed along with ExCom library [5] as part of his master thesis [6] under the GNU LGPL license, the same licensing as for the whole ExCom. It is written in C++.

The dictionary is represented as an array of pointers to the input file. Block data size, which is defined by the program parameter, determines upper memory consumption bound that dictionary could consume. After block of data of the selected size is read, dictionary is cleared and new block of data starts from scratch. It means dictionary content from one data block is not used for any other. This is small improvement from the previous versions where size of the dictionary was dependent of the input file.

Dictionary update is based on merge sort, all new context-content pairs (represented by pointer to the file block) are added to the end of the dictionary at once, then it is sorted using linear probing. This is an improvement, in previous versions every new context-content pair was inserted separately leading to multiple shifting of the dictionary.

No innovation was made to the triplets, they are present as mentioned in Section 1.2.1. No coding is used and instead of distance d, the content index is part of the triplet. That means first part of the triplet occupies same amount of triplet as the data block length specified by execution parameter. The triplet looks like

$$(cnt, l, c)$$

Modification 1.3.2 is not used, because best content search is limited to already encoded data and length l specified as program parameter.

Even though best content is not used for distance calculation, it is searched anyway. The implementation also contains methods to store the distance and the length values into a file for later histogram vizualization.

2.5 Michal Valach

The latest and most advanced implementation of ACB compression method is one made by Michal Valach as his Master's thesis [7] at Czech Technical

University in Prague in 2011. This implementation extended and improved previous ACB method made by Filip Šimek [6] mentioned 2.4 in ExCom library [5]. As well as the ExCom library, this algorithm was published under the GNU LGPL license and is written in C++.

Michal Valach extends previous work done on ACB compression method by vast research, analysis and improvements for both time and compression ratio. His work is significant improvement at the field of compression in ExCom library.

Dictionary is represented by indexing to the input file. It is backed by various data structures under the interface SuffixStructure, particular subclasses are SuffixTree, SuffixArray and SuffixArray2.

The SuffixTree is implemented using STD library map, which is backed by the Red-Black tree. The Red-Black tree has the access time of O(log (|Σ|) and the content is sorted, which allows to use its method upper bound for searching for the closest higher key.

The SuffixArray is implementation of suffix array data structure based on the STD library vectors. It is implemented as a two-dimensional vector array, the first dimension is indexed using the first symbol of the context. The second dimension is a suffx array for the suffixes that starts with particular symbol. For searching, binary search is used with the complexity of O(log n). Dictionary update is done by insertion to vector, leading to it's shift of O(n) complexity.

The SuffixArray2 is backed by the B+ Tree and due to the measurements, it is used as a default implementation for the dictionary. Implementation of rotations is not present in this B+ Tree. The rotations are procedures to restructure the tree to be consistent of maximal depth equal to $\log_B n$, happening when some node was filled but there are still free spots elsewhere in the same depth. Main usage is to save memory, but at the cost of speed. The SuffixArray2 also contains the implementation of sorted contents as described in Section 1.3.3.

For triplet representation, all variations listed below were implemented and tested.

- The whole triplet (l, d, c) is always sent to the output,
- The triplet (1, l, d) or (0, c) is sent to the output,
- The triplet (1, l, d, c) or (0, c) is sent to the output,
- The triplet (l + |Σ| - 1, d) or (c) is sent to the output,
- The triplet (l, d, c) or (l, c) is sent to the output.

The values 1 and 0 are bit flags, l is the common length, d is the distance, c is the next symbol, and $|\Sigma|$ is the size of the alphabet.

In terms of triplet coding, Adaptive Huffman Coding and Adaptive Arithmetic Coding is implemented.

Research pointed out that B+ Trees are fastest dictionary data structure, (1, l, d, c)/(0, c) and (l, d, c)/(l, c) are most efficient triplet representations and Adaptive Arithmetic Coding better method to code emerging triplets to the final output file. On the other hand, modification of sorted contents, mentioned in [2] and in Section 1.3.3, brings no compression or time improvement.

Chapter 3

Analysis

This chapter focuses on analysis of all important tasks that have to be examined and discussed for successful and effective ACB compression method implementation. It's structure is divided into three main sections.

The first section examines possible ways to store and represent the dictionary described in Section 1.2.1. It lists the suitable data structures taking into account frequently performed actions, which is search for best matching context, traversing it's surrounding to find the best matching content, and also to be able to update the dictionary with new pairs without the need of shifting a lot of data.

The second section covers anything related to triplets. The way triplets might be represented, ways to store triplets into output bit stream, what data the triplets should contain and how to determine information needed from that. The possible ways of coding triplets are discussed too.

3.1 Dictionary structure

The particular dictionary structure was specified in section 1.2.1. It consists of few main key parts that were mentioned, but requirements to them were not listed yet.

Dictionary pairs. The dictionary is made of context-content pairs. Context-content pairs are required in order to store coupled parts of the text that were once divided by floating window into context and content parts. The context-content relation must be evident from any type of data structure used. And all modifications, especially variable output from section 1.3.1, must be possible.

Search for the best matching context. The first function in the algorithm, that the dictionary should provide, is searching for the best context.

Search is done in the whole dictionary, so it would have at least linear complexity if no support is provided. But the search have to be as quick as possible. As the comparison with the current context is made by right-to-left bit traverse, it makes sense to sort the dictionary.

Search for the best matching content. This function is called only in compression process. It searches the dictionary for the best matching content to the current one. This time, it uses normal left-to-right comparison. By the means of the algorithm, it searches contents as close to the best matching content as possible, to keep the distance low. The neighbour dictionary pairs should be accessed as quickly as possible.

Dictionary ordering. Context-content pairs are stored in specific order to provide context search operation logarithmic time complexity. The ordering is made by right-to-left comparison, comparing contexts only, as the search for the best matching context is made at first. It might make sense to sort contexts and contents separately, while preserving relation between context and content for each pair. But then, the distance between best context and content will not be low.

Dictionary update. After the triplet is created, sliding window is shifted by common length of contents. All skipped positions in text compose new context-content pairs and have to be added to the dictionary. The dictionary must remain sorted.

Requirement for all those steps is to be as fast as possible. Ideally constant time for both the data access and data update. The problem is that if we want constant time updates, we cannot have constant time access and vice versa.

The delete operation is not present in compression nor decompression algorithm, so it do not have to be implemented at all.

The aim is to analyse structures that have the most balanced data access and data update times. In the middle of both requirements happened to be logarithmic complexity for both operations.

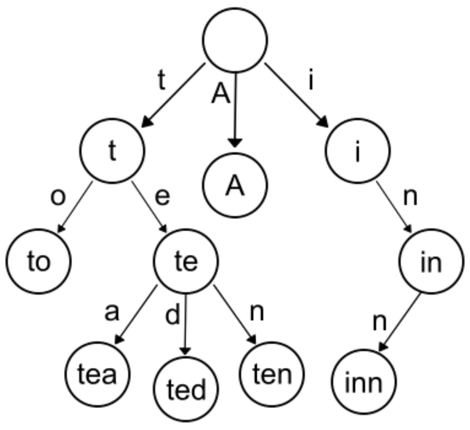
3.1.1 Suffix Structures

Suffix, also called postfix, is lunguistic term describing the ending of a word. The search for contexts is done by backward comparison, in other words, the search for the longest common postfix is made. There are few structures that are designed to do it effectively.

3.1.1.1 Suffix Trie

Suffix trie was presented in 1960 by Edward Fredkin [8]. An example containing words *A, to, tea, ted, ten, i, in* and *inn* is shown in Figure 3.1[1]. Each edge represents a symbol, each node value is created by concatenation of all edges (symbols) from root and represents a word.

Figure 3.1: Trie structure example.



This trie example could be called a prefix tree as it does not only contains all named strings, but also all their prefixes. Unlike suffix trie, that is similar structure with difference that on top of all prefixes of all strings, it also contains all prefixes of all suffixes of all words it contains. So, for example if the suffix trie contains word *mississippi* it also contains *ississippi, ssissippi, sissippi* and so on until ε, where ε is the root of the trie. Every single node of a suffix trie is represented by data structure containing one symbol of the edge leading to it, list of children of this node and pointers to siblings.

3.1.1.2 Suffix Tree

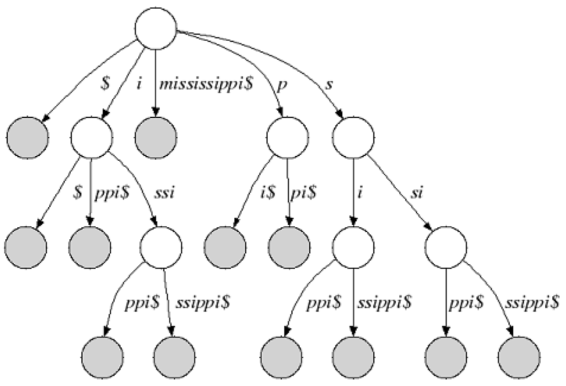
Suffix tree is structure very similar to suffix trie presented in previous section. It was first introduced by Peter Wiener in 1973 [9]. There were some improvements presented later, but most important one was the first on-line construction algorithm in linear time by Ukkonen [10] in 1995.

In comparison, suffix trie and suffix tree almost looks identical, the most noticeable difference is that suffix tree has the non branching nodes that are not leaves are compacted. The example for string *mississippi* is shown in Figure 3.2[2]. Each node represents a sub-string of a suffix with minimal length

[1]Source: https://upload.wikimedia.org/wikipedia/commons/b/be/Trie_example.svg
[2]Source: https://seqan.readthedocs.io/en/seqan-v2.0.2/_images/streeNoSentinel.png

of 1. To get the whole suffix of a node, all parents of the current node must be concatenated.

Figure 3.2: Suffix Tree example.



Suffix links are required for the construction of the suffix tree. The suffix links are links between nodes with lengths l and $l + 1$, where the shorter one is the suffix of the longer one. The link is directed from longer one to shorter one.

A node in a suffix tree is represented as a data structure that contains start of a suffix in data, length of the text in node, or suffix end, list of children of the node and pointers to siblings of the node. The siblings are in both the suffix trie and tree used for linear traversal of the nodes after the best matching context is found, and traverse have to be done to find best matching content in proximity.

For the ACB dictionary purpose, both *suffix trie* and *suffix tree* are almost identical structures. The construction algorithm presented by Ukkonen [10] is designed to create the suffix trees from the source string in ordinary left-to-right order. Algorithm is noted as on-line, meaning that if there is already a suffix tree for the string S and new symbol c is added, resulting string Sc is created easily, without need to rebuild the structure from scratch. The problem with this algorithm is that for the ACB dictionary usage, new symbol c needs to be added to the beginning of the string S to form cS. Ukkonen's algorithm needs to be reworked to expand suffixes from the start, not from the end. This approach is presented by Michal Valach [7] in his master thesis. The complexitis are presented in Table 3.1, but the algorithm itself is not analysed in detail here.

3.1.1.3 Suffix Array

Suffix array, as the name says, consists of an array containing pointers to the original text, to suffixes in lexicographical order. This means a huge advantage

Table 3.1: Operation complexity of Suffix Tree

Property	Complexity
Space	$O(n)$
Search	$O(m)$
Insert (average)	$O(n)$
Insert (worst)	$O(n \exp 2)$

over the *suffix tree* in memory consumption. Another plus is speed of accessing suffixes, but at a cost of updating speed. This is because the backing data structure is array and updating an array is always expensive, when a new suffix is inserted, all the consequent suffixes have to be moved. An example of the suffix array is shown in Table 3.2 where the original text *mississippi* was indexed from number 1 and then when *suffix array* is appropriately sorted.

Table 3.2: Suffixes and suffix array

SA		SA	
1	mississippi	12	$
2	ississippi	11	i
3	ssissippi	8	ippi
4	sissippi	5	issippi
5	issippi	2	ississippi
6	ssippi	1	mississippi
7	sippi	10	pi
8	ippi	9	ppi
9	ppi	7	sippi
10	pi	4	sissippi
11	i	6	ssippi
12	$	3	ssissippi

In 1993, Gene Myers and Udi Manber [11] presented a *suffix array* as a potential substitution of *suffix trees* because of their memory demands. Along with the algorithm, they introduced inefficient constructing algorithms, which were later improved up to the linear sorting time algorithm by Pang Ko and Srinivas Aluru [12] in 2003. There are more algorithms, but all of them operates over the static text, which is inappropriate for ACB dictionary. After every triplet creation, new elements are added to the dictionary, thus it is continuously growing. But there exists one exception, it is an algorithm that is able to update the suffix array after the source string is prolonged.

Data structure used for the *suffix array* would be *ArrayList* with *Integer* as inner data type for indexing. It has constant time access and search for the best matching context is done by binary search in *O(log n)*. The search for the best matching content is done easily too, traversing neighbour array fields is simply handled by a *for* loop. Memory consumed equals to *n times* data type used for indexing, *Integer* with 4 byte size in our case. It was already said that the best problem is dictionary update. For every suffix index inserted, all following elements have to be shifted, making it *O(n)* worst case complexity. Only possible mitigation is to add all newly created dictionary pairs, represented by suffix indices to the original text, to the end of the *suffix array*, and sort them altogether.

Table 3.3: Operation complexity of Suffix Array

Property	Complexity
Space	$O(n)$
Search	$O(n \log n)$
Traversal	$O(1)$
Insert	$O(n)$

3.1.2 Search Tree Structures

There comes a lot of good ideas from Suffix Structures analysis, but they are not ideal. *Suffix Arrays* were good for searches and traversing neighbours and *Suffix Trees* were good for fast updating. Now let have a look at tree structure possibilities and analyse if they might be a fit.

Tree is a hierarchical data structure consisting of *nodes* and *links* between nodes. It has a *root node* and *subtrees* of children with parent, represented as linked nodes. Each node is data structure consisting of value and links to child nodes, where no link reference is duplicated and none of them points to the root. It means no link cycle is possible. For the basic tree terminology, see example in Figure 3.3[3].

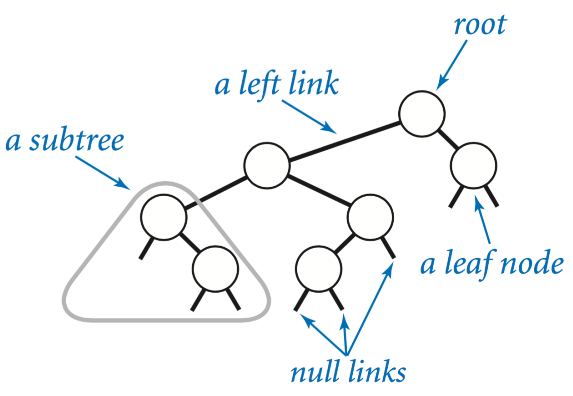
If the node data structure is enhanced by key, forming *¡Key, Value¿* pair, and keys are comparable, this tree is referred as a *Search Tree*. If any node has no more than two child node, this tree is referred as a *Binary Tree*.

3.1.2.1 Binary Search Tree

To define binary search trees, we may use recursive definition. It is either empty (null) or a node containing links to two disjoint binary trees. We refer to the node at the top as the *root* of the tree, the node referenced by its left link as the *left subtree*, and the node referenced by its right link as the *right subtree* [13]. Nodes whose links are both null are called *leaf* nodes. The *height* of a tree is the maximum number of links on any path from the root node to a leaf node. An example of the binary tree is shown in Figure 3.3.

[3]Source: http://introcs.cs.princeton.edu/java/44st/images/binary-tree.png

Figure 3.3: Binary tree with terminology.



A *binary search tree* is a binary tree that contains a key–value pair in each node and for which the keys are in *symmetric order*: The key in a node is larger than the key of every node in its left subtree and smaller than the key of every node in its right subtree. It allows all ACB dictionary operations to operate as described next.

Searching a binary search tree for a specific key can be programmed recursively or iteratively. If the tree is empty, terminate the search as unsuccessful. If the search key is equal to the key in the node, terminate the search as successful (by returning the value associated with the key). If the search key is smaller than the key in the node, search (recursively) in the left subtree. If the search key is greater than the key in the node, search (recursively) in the right subtree.

Insertion logic is similar to searching for a key, but the implementation is trickier. The key to understanding it is to realize that only one link must be changed to point to the new node, and that link is precisely the link that would be found to be null in an unsuccessful search for that key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Delete operation will not be discussed as it is not present in ACB dictionary.

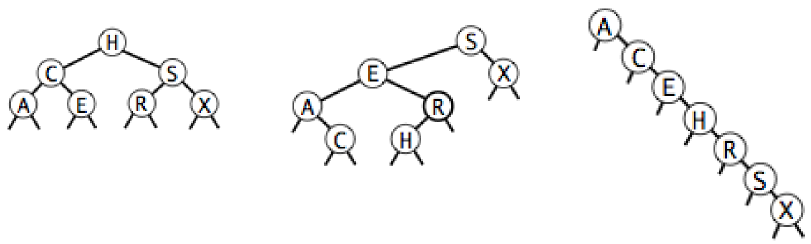
Traversing a binary search tree is simple, the most basic tree-processing function is known as *tree traversal*: given a (reference to) a tree, we want to systematically process every node in the tree. For linked lists, we accomplish this task by following the single link to move from one node to the next. For trees, however, we have decisions to make, because there are two links to follow. To process every node in a binary search tree, recursively process every node in the left subtree, then process the node at the root, and then process every node in the right subtree. This approach is known as inorder tree traversal because it processes the nodes in a binary search tree in key-sorted order.

But for ACB dictionary purpose, this function is useless, as it proccess whole key set and not nodes by siblings of the best matching context. There is no way of traversing nodes by siblings if they have no reference to the nearest higher and lesser value, but only to the left and right subtree. For example, closest higher value of biggest key in left subtree of a root is the smallest value in the right subtree and there is no link between them.

The running times of algorithms on binary search trees are ultimately dependent on the shape of the trees, and the shape of the trees is dependent on the order in which the keys are inserted, because inserted key is appended to null child link of a proper leaf node. This leads to possibilities of a best case scenario and a worst case scenario of inserting order, it is shown in Figure 3.4[4].

Figure 3.4: Binary tree possible shapes.

(a) Best case. (b) Average case. (c) Worst case.



In the *best case*, the tree is perfectly balanced (each node has exactly two non-null children), with about $\log n$ links between the root and each leaf node. In such a tree, the cost of every put operation and get request is proportional to the height of the tree, $\log n$ or less.

In *average case*, a classic mathematical derivation shows that the expected number of key compares is $2 \log n$ for a random put or get in a tree built from n randomly ordered keys.

In the *worst case*, each node (except one) has exactly one null link, so the binary search tree is essentially a linked list with an extra wasted link, where put operations and get requests take linear time. Unfortunately this worst case is not rare in practice. It arises, for example, when we insert the keys in order.

[4]Source: http://algs4.cs.princeton.edu/32bst/images/

The running times of algorithms on binary search trees depend on the shapes of the trees 3.4, which, in turn, depends on the order in which keys are inserted. And in ACB dictionary, the order cannot be affected.

Table 3.4: Complexity of Binary Search Tree

Property	Average	Worst case
Space	$\Theta(n)$	$O(n)$
Search	$\Theta(\log n)$	$O(n)$
Insert	$\Theta(\log n)$	$O(n)$

There are two possibilities to minimize operation time, both are trying to keep tree as close as possible to best-case scenario. First is by enhancing the tree to optimal binary search tree, presented by Gaston Gonnet in 2013 [14], which is a search tree where the average cost of looking up an item (the expected search cost) is minimized by statistical optimisations to position particular items as they are accessed, frequently accessed close to the top and vice versa. It is not usable for ACB dictionary. Second option is to balance tree after each dictionary update, which is discussed in the next section.

3.1.2.2 Self-balancing Search Trees

A *self-balancing binary search tree* or *height-balanced binary search tree* is a binary search tree that attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible at all times, automatically.

In previous section was mentioned that most operations on a binary search tree take time proportional to the height of the tree, so it is desirable to keep the height small. Self-balancing binary trees solve this problem by performing transformations on the tree at key times, in order to reduce the height. Although a certain overhead is involved, it is justified in the long run by ensuring fast execution of later operations.

A binary tree with height h can contain at most $2^0+2^1+...+2^h = 2^{h+1}-1$ nodes. The height of the tree is expressed by Equation 3.1 for a tree with n nodes and height h. The height must always be at most the $\lceil \log_2 n \rceil$ and at the least $\lfloor \log_2 n \rfloor$.

$$\begin{aligned} n &\le 2^{h+1}-1 \\ h &\ge \lceil \log_2(n+1)-1 \rceil \\ h &\ge \lfloor \log_2 2 \rfloor \end{aligned} \tag{3.1}$$

Knowing this, the time complexities for balancing binary search trees are shown in Table 3.5.

The feature of balance is achieved differently for different balanced binary search trees implementations, this thesis states two most common ones.

Table 3.5: Time complexity for self-balancing trees

Operation	Time complexity
Lookup	$O(\log n)$
Insertion	$O(\log n)$
In-order traversal	$O(n)$

An **AVL tree** was first balanced data structure invented [15] by Georgy Adelson-Velsky and Evgenii Landis in 1962, where "AVL" stands for **A**delson-**V**elsky **L**andis. If the maximum and minimum height differs more than one, rebalancing is executed to restore the balance property. Balancing is maintained by tree rotations, and the need to balance is recognized by *balance factor* which is also characteristic property of AVL trees. Balance factor for a node acquires values {-1, 0, 1} and means a height difference of its two child subtrees. Balance factors can be kept up-to-date by knowing the previous balance factors before insertion and the change in height – it is not necessary to know the absolute height. For holding the AVL balance information, two bits per node are sufficient [15].

A **Red-Black tree** is a binary search tree with one extra bit of storage per node: its color, which can be either *red* or *black*. By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced. Each node of the tree now contains the fields *color*, *key*, *left* and *right* link.

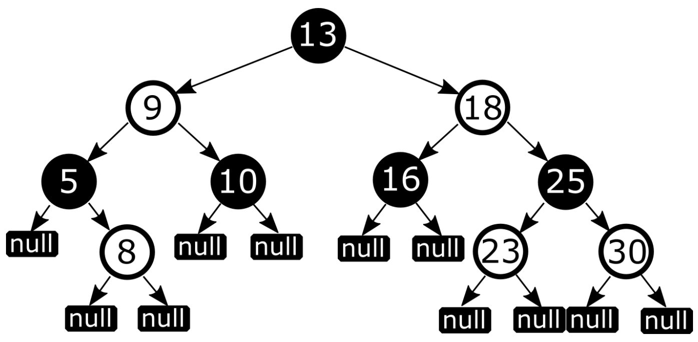
A binary search tree is a red-black tree if it satisfies the following *red-black properties*:

1. Every node is either red or black.
2. Each null pointer is considered to be black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

An example of a red-black tree is shown in Figure 3.5.

As mentioned before, self-balancing binary search trees keeps the balance by rotations. The rotation is made by reconnecting links between nodes, thus reforming a tree. Four basic rotations can be distinguished and the elementary moves are demonstrated in Figure 3.6.

Figure 3.5: Red-Black tree example.



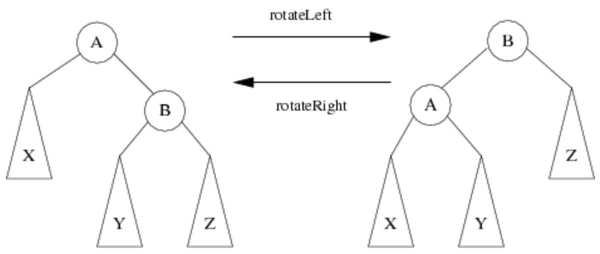
Left rotation. During left rotation, the node n is replaced by its right sub-tree and the node n becomes left sub-tree. The left sub-tree of the node $n.right$ after rotation is right sub-tree node n.

Right rotation. During right rotation, the node n is replaced by its left sub-tree and node n becomes its right sub-tree. The right sub-tree of the node $n.left$ after rotation is left sub-tree node n.

Left-right rotation. Left-right rotation is composition of two rotation operations, left rotation around node $n.left$ and right rotation around node n.

Right-left rotation. Right-left rotation is opposite operation to left-right one. It rotates right around $n.right$ and rotates left around n.

Figure 3.6: Elementary tree rotations.



Both AVL trees and red–black trees are similar tree structures with similar behaviour. The operations to balance the trees are different; both AVL trees and red-black require $O(1)$ rotations in the worst case, while both also require $O(logn)$ other updates (to colors or heights) in the worst case. AVL trees require storing 2 bits of information in each node, while red-black trees require just one bit per node. The bigger difference between the two data structures is their height limit. The AVL tree is more strict about height and requires more frequent adjustments, taking longer to update but is faster in case of search.

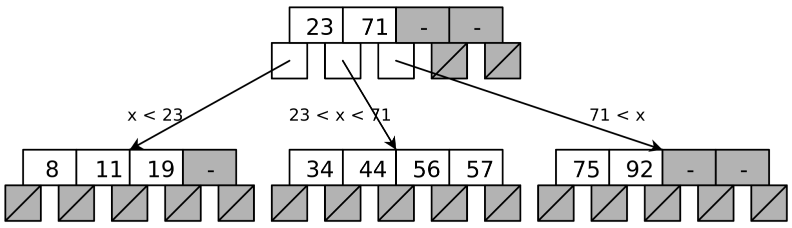
Because update is more frequent operation in ACB dictionary, it makes more sense to use red-black trees over AVL trees.

3.1.2.3 B-Trees

The *B Tree* was firstly introduced by Bayer and McCreight in 1970 [16]. The B Tree is a general form of the Binary search tree, where the node has more than two children, but the particular terminology and definitions are not uniform. The definition by Donal Knuth [17] that B-tree of order m is a tree which satisfies the following properties:

- Every node has at most m children.
- Every non-leaf node (except root) has at least $\lceil \frac{m}{2} \rceil$ children.
- The root has at least two children if it is not a leaf node.
- A non-leaf node with k children contains *k-1* keys.
- All leaves appear in the same level.

Figure 3.7: Example of a B Tree.



The example of a b-tree is shown in Figure 3.7[5]. Each internal node keys separates keys $(k_1 ... k_m)$ between two children meaning that all keys in the left child of the key k_1 are lower than the key k_1, and all the keys in the right child are greater then the key k_1.

The *search* in the B Tree is identical to search in the binary search tree, except the fact, that instead of deciding between two nodes, we need to decide between m nodes. This search in the tree node can be done using the linear search if m is low, but the binary search is faster.

The *insertion* in the B Tree consists of two steps. The first step is to locate the node into which the new value should be inserted. In the B Tree, the insertion always starts from a leaf node and can end up in two states. The node is not full, so the value is simply added to the proper position. Or, the node is full and has to be split into two new nodes with evenly distributed

[5]Source: https://upload.wikimedia.org/wikipedia/commons/6/68/B-tree_example.svg

numbers. As a new separation key is taken a median of those numbers. It is then inserted to its parent and the algorithm is repeated until it fins its place. If it climbs up to the root and it is full too, new root is created containing one key and two children.

The complexities of the basic operations in the B Tree can be seen in Table 3.6. The logarithms are not of base 2, but base m which is the size of the node.

Table 3.6: B Tree complexities.

Operation	Average & Worst
Space	$O(n)$
Search	$O(\log_m n)$
Insert	$O(\log_m n)$

A modification of the B Tree called the B+ Tree is very similar data structure. The main difference between the B Tree and the B+ Tree is that in the B+ Tree the data are all stored in the leaves, and the inner nodes only contain keys for selecting lower nodes. The B+ Tree leaf nodes can also be linked using pointers to their left and right siblings. Both these modifications means that after the desired data are found, it is possible to linearly continue to the left or to the right for other values without the need of traversing the tree. This is the main reason why the B+ Tree is much better choice for implementing the ACB dictionary then the simple B Tree.

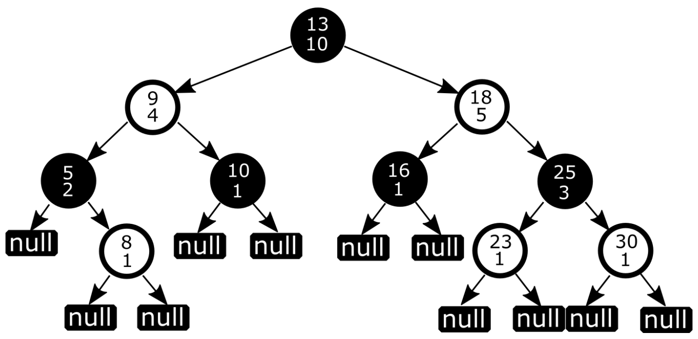
3.1.3 Dynamic order statistics

The ACB dictionary consists of few frequently repeated operations introduced at the beginning of Chapter 3.1. While compressing, best content search is done in $O \log n$ thanks to binary search trees, and the search for best matching content is done by traversing the siblings. The length might be counted as number of siblings traversed, positive or negative, depending on direction. In case of decompression, best content search is similar, but the length is given to the algorithm, and need to traverse that far by siblings is not very effective option. Rather than that, some sort of statistical order information have to be added to the tree.

This property is called *dynamic order statistics* and the tree with this property is called *order statistic tree*. The order statistic tree is a variant of the binary search tree (or more generally, a B-tree) that supports two additional operations beyond insertion, lookup and deletion. Both operations can be performed in $O(\log n)$ worst case time when a self-balancing tree is used as the base data structure [13].

1. *Select*(i) - find the i^{th} smallest element stored in the tree (sometimes called $SearchByRank(x)$).
2. *Rank*(x) - find the rank of element x in the tree, i.e. its index in the sorted list of elements of the tree.

Figure 3.8: An augmented red-black tree to support order statistics



A data structure that can support fast order-statistic operations is shown in Figure 3.8. An order-statistic tree T is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree fields *key[x]*, *color[x]*, *p[x]*, *left[x]*, and *right[x]* in a node x, we have another field *size[x]*. This field contains the number of (internal) nodes in the subtree rooted at x (including x itself), that is, the size of the subtree. In the Figure 3.8 it is illustrated as $\binom{key[x]}{size[x]}$.

$$size[x] = size[left[x]] + size[right[x]] + 1$$

3.1.3.1 Retrieving an element with a given rank

The procedure SELECT(x, i) returns a pointer to the node containing the i^{th} smallest key in the subtree rooted at x. To find the i^{th} smallest key in an order-statistic tree T, we call SELECT(root[T], i).

The idea behind SELECT is following. The value of SIZE[LEFT[X]] is the number of nodes that come before x in an inorder tree walk of the subtree rooted at x. Thus, SIZE[left[x]] + 1 is the rank of x within the subtree rooted at x.

In line 2 of SELECT, we compute r, the rank of node x within the subtree rooted at x. If $i = r$, then node x is the i^{th} smallest element, so we return x in line 4. If $i < r$, then the i^{th} smallest element is in x's left subtree, so we recurse on left$[x]$ in line 6. If $i > r$, then the i^{th} smallest element is in x's right subtree. Since there are r elements in the subtree rooted at x that come before x's right subtree in an inorder tree walk, the i^{th} smallest element in the

Algorithm 3 Select

```
 1: procedure SELECT(x, i)
 2:     $r \leftarrow$ size[left[x]] + 1
 3:     if i = r then
 4:         return r
 5:     else
 6:         if i < r then
 7:             return SELECT(left[x], i)
 8:         else
 9:             return SELECT(right[x], i - r)
10:         end if
11:     end if
12: end procedure
```

subtree rooted at x is the $(i-r)^{th}$ smallest element in the subtree rooted at right[x]. This element is determined recursively in line 9.

To see how SELECT operates, consider a search for the 6^{th} smallest element in the order-statistic tree of Figure 3.8. We begin with x as the root, whose key is 13, and with i = 6. Since the size of 13's left subtree is 4, its rank is 5. Thus, we know that the node with rank 6 is the 6 - 5 = 1^{st} smallest element in 13's right subtree. After the recursive call, x is the node with key 18, and i = 1. Since the size of 18's left subtree is 1, its rank within its subtree is 2, which means it is the second smallest element. Thus, a pointer to the left subtree node with key 16 is returned by the procedure.

Because each recursive call goes down one level in the order-statistic tree, the total time for SELECT is at worst proportional to the height of the tree. Since the tree is a red-black tree, its height is $O(\log n)$, where n is the number of nodes. The running time of SELECT is $O(\log n)$ for a dynamic set of n elements.

3.1.3.2 Determining the rank of an element

Given a pointer to a node x in an order-statistic tree T, the procedure RANK returns the position of x in the linear order determined by an inorder tree walk of T.

The procedure works as follows. The rank of x can be viewed as the number of nodes preceding x in an inorder tree walk, plus 1 for x itself. The following invariant is maintained: at the top of the while loop of lines 4-9, r is the rank of key[x] in the subtree rooted at node y. We maintain this invariant as follows. In line 2, we set r to be the rank of key[x] within the subtree rooted at x. Setting y $\leftarrow$ x in line 3 makes the invariant true the first time the test in line 4 executes. In each iteration of the while loop, we consider the subtree

Algorithm 4 Rank

```
1: procedure RANK(T, x)
2:     $r \leftarrow$ size[left[x]] + 1
3:     $y \leftarrow x$
4:     while $y \neq root[T]$ do
5:         if y = right[p[y]] then
6:             $r \leftarrow r + size[left[p[y]]] + 1]$
7:         end if
8:         $y \leftarrow p[y]$
9:     end while
10:    return r
11: end procedure
```

rooted at p[y]. We have already counted the number of nodes in the subtree rooted at node y that precede x in an inorder walk, so we must add the nodes in the subtree rooted at y's sibling that precede x in an inorder walk, plus 1 for p[y] if it, too, precedes x. If y is a left child, then neither p[y] nor any node in p[y]'s right subtree precedes x, so we leave r alone. Otherwise, y is a right child and all the nodes in p[y]'s left subtree precede x, as does p[y] itself. Thus, in line 6, we add size[left[y]] + 1 to the current value of r. Setting y $\leftarrow$ p[y] makes the invariant true for the next iteration. When y = *root[T]*, the procedure returns the value of r, which is now the rank of key[x].

Since each iteration of the while loop takes $O(1)$ time, and y goes up one level in the tree with each iteration, the running time of Rank is at worst proportional to the height of the tree, $O(\log n)$ on an n-node order-statistic tree.

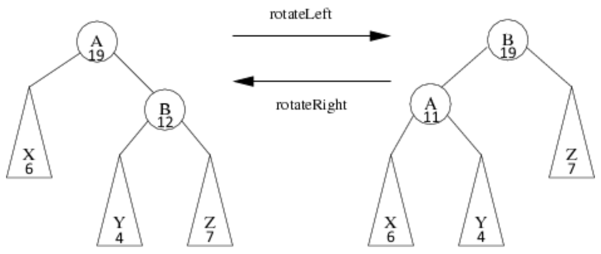
3.1.3.3 Maintaining subtree sizes

Given the size field in each node, Select and Rank can quickly compute order-statistic information. But unless these fields can be efficiently maintained by the basic modifying operations on red-black trees, our work will have been for naught. We shall now show that subtree sizes can be maintained for insertion without affecting the asymptotic running times of either operation.

We noted in Section 3.1.2.2 that insertion into a red-black tree consists of two phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and ultimately performing rotations to maintain the red-black properties.

The two size fields that need to be updated are the ones incident on the link around which the rotation is performed. The updates are local, requiring

Figure 3.9: Tree rotations with size property.



only the size information stored in A, B, and the roots of the subtrees shown as triangles x, y and z in Figure 3.9.

To maintain the subtree sizes in the first phase, we simply increment size[x] for each node x on the path traversed from the root down toward the leaves. The new node added gets a size of 1. Since there are $O(\log n)$ nodes on the traversed path, the additional cost of maintaining the size fields is $O(\log n)$. In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: it invalidates only the two size fields in the nodes incident on the link around which the rotation is performed.

Since at most two rotations are performed during insertion into a red-black tree, only $O(1)$ additional time is spent updating size fields in the second phase. Thus, the total time for insertion into an n-node order-statistic tree is $O(\log n)$, asymptotically the same as for an ordinary red-black tree.

3.2 Triplet representation and coding

Triplets are the output of the ACB compression algorithm and they, by default, consists of three parts, that is why they are called triplets. There exists more variations and usage of triplets, this is summarized in section 3.2.1.

Storing triplets into output is not very efficient if no further coding is done. Various approaches should be analyzed to decide which to chose and how to handle it in section 3.2.2.1.

3.2.1 Triplet variations

Few possible variations were already presented in Chapter 2. To remind, they were:

- Default triplet (l, d, c),
- the triplet (1, l, d) or (0, c) presented by Salomon [2],

- the triplet (1, s, d, l, 0, c) or (1, s, d, l, 1) or (0, 0, c) variation by Děcký
- the triplet (l + |Σ| - 1, d) or (c) presented by Cerman,
- the triplet (|Σ|, l - 1, d) or (c) presented by Cerman too,
- the triplet (1, l, d, c) or (0, c) as improvement to triplet by Salomon,
- the triplet (l, c) or (l, d, c) presented by Valach.

where zeros and ones are bit flags, d is the distance, l is the common length, c is a new symbol, s is the sign of the distance d, and $|\Sigma|$ is the size of the alphabet.

In the thesis done by Valach [7] it is proved that the last two mentioned triplet variations are the best ones in terms of final compression ratio.

3.2.2 Triplet coding

The coding of triplets, as shown later, needs bounded values to work properly. A maximum possible value is assigned to each part of the triplet to restrict its bit size. So if it is defined that any part of a triplet occupies b bits, this triplet part can contain 2^b different values. This changes the behaviour of ACB dictionary and compression or decompression method as follows:

Bit flag. The bit flag doesn't change anything. It is just showing what different parts does the triplet have. Bit flag should be stored into 1 bit.

Length. The length shows how many common symbols did actual content and best matching content have. During the comparing part of the algorithm, maximal possible length have to be taken into account. Even if there are more common symbol, the comparison have to stop and maximal value is stored. This value is always zero or positive, unsigned data types might be used internally in the triplet representation before it is serialized into bit output.

Distance. The distance is also bounded into maximal value. When searching for the best matching content, it cannot be done in the whole dictionary, as such distance value would not have any specific boundary. Instead, the vicinity of context with index i is searched in the interval $[i-2^{d-1}, i+2^{d-1})$ where d is the number of bits assigned to this triplet property. The distance takes both positive and negative values so caution should be taken when serializing and deserializing triplets. Serialization is done by AND operation with bit mask of value $2^d - 1$ and deserialization by inverting all leading zeros if it should be a negative number. In asymptotic complexity, the search for the best matching content is $O(1)$, because both the distance and the length have upper limit.

Symbol. Bit size of a symbol triplet property depends on the size of an alphabet. Commonly used alphabets for compression methods are 256 symbols, thus 8 bits, 1 byte.

Encoding of the triplets, above stated also as a de/serialization, have to be done using an *uniquely decodable code.* It has to be uniquely decodable, so that the decoder decodes the input stream without any guessing. It satisfies many codes, but the most effective appeared to be the adaptive arithmetic coding [7]. About the requirements and demands over the adaptive arithmetic coding is the next section 3.2.2.1. But arithmetic coding is not the only such code, there exists similar Range coding, covered in section 3.2.2.2.

3.2.2.1 Arithmetic coding

The most important advantage of arithmetic coding is its flexibility: it can be used in conjunction with any model that can provide a sequence of event probabilities. This advantage is significant because large compression gains can be obtained only through the use of sophisticated models of the input data. Models used for arithmetic coding may be adaptive, and in fact a number of independent models may be used in succession in coding a single file.

The other important advantage of arithmetic coding is its optimality. Arithmetic coding is optimal in theory and very nearly optimal in practice, in the sense of encoding using minimal average code length. When the probability of some single symbol is close to 1, arithmetic coding does give considerably better compression than other methods. So the bigger probability of symbols, the more efficient the arithmetic coding is. And in case of some triplet components, the probability is pretty high.

A disadvantage of arithmetic coding is that it does not in general produce a prefix code, excluding parallel coding with multiple processors. A minor disadvantage is the need to indicate the end of the file.

The algorithm for encoding a file using arithmetic coding works conceptually as follows:

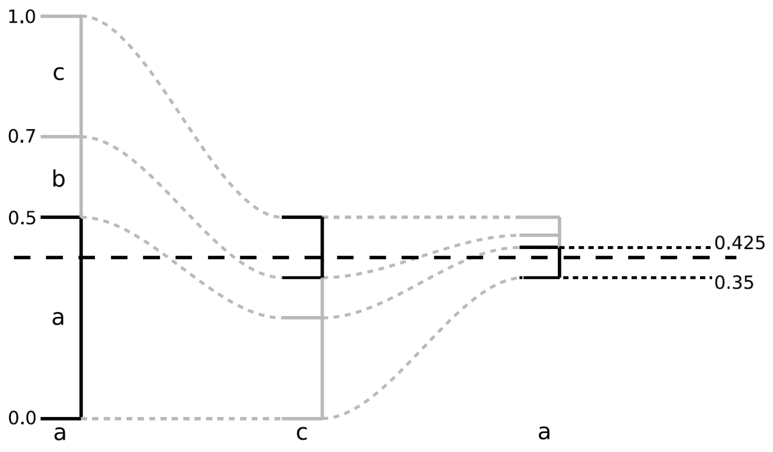
1. We begin with a "current interval" [L, H) initialized to [0; 1).
2. For each symbol of the input, we perform two steps:
   a) We subdivide the current interval into subintervals, one for each possible alphabet symbol. The size of a symbol's subinterval is proportional to the estimated probability that the symbol will be the next symbol in the input.
   b) We select the subinterval corresponding to the symbol that actually occurs next in the input, and make it the new current interval.

3. We output enough bits to distinguish the final current interval from all other possible final intervals.

The length of the final subinterval is clearly equal to the product of the probabilities of the individual symbols. Also, some mechanism to indicate the end of the input is needed, either a special end-of-file symbol coded just once, or some external indication of the file's length. An example of probabilities and interval division is shown in Figure 3.10.

Aside from other compression algorithms, where some code is created after every encoded symbol, arithmetic coding produces just one decimal number between zero, included, and one, excluded. Division by the number according to probabilities is made in every step. It is not possible to divide a number stored in bits huge amount of times without a precision lost, this is solved by shifting bounds of the bits, replacing least significant bytes with most significant ones, underflowing, rescaling etc. More about this algorithm can be found in [2].

Figure 3.10: Probabilities of arithmetic coding "aca"



The adaptive arithmetic coding works that every symbol in the arithmetic coding starts with the same non-zero probability. This starting probability could be adjusted, if any assumption for the probable values could be made. During the encoding/decoding these probabilities are updated after each symbol is encoded/decoded.

As arithmetic coding depends only on the symbol probabilities and the position of the selected probability, it allows to use multiple probability distributions in one arithmetic code. In other words, it is possible to use multiple alphabets in one code, if there is a deterministic algorithm to select, which alphabet is the correct one at any time during the decompression. Or, another possibility is to have separate arithmetic coder with its own frequencies and final interval number for every alphabet. Those alphabets are in our case the triplet components.

3.2.2.2 Range coding

The range encoder is an entropy coder similar to the arithmetic coder. Compared to an arithmetic coder the files are almost the same size (less than 0.01% in most cases) but the operation speed is faster [18]. It is so similar to arithmetic coder that I would say it is identical. Small differences are stated in Martin's paper [18], that is such range encoders have the tendency to perform renormalization a byte at a time, rather than one bit at a time (as is usually the case). In other words, range encoders tend to use bytes as encoding digits, rather than bits. While this does reduce the amount of compression that can be achieved by a very small amount, it is faster than when performing renormalization for each bit. In fact, it does not matter whether range or arithmetic coder is used.

3.3 Algorithm improvements

This section deals with possible new features and improvements of the ACB compression method and dictionary management. Section 3.3.1 is featuring the possibilities and potential trouble of use of the second best content and the longest common prefix of the 1^{st} and 2^{nd} best one.

3.3.1 Second best content

As a reminder, the original coding process presented by G. Buyanovski [1] will be repeated.

Lets have a string and compression method in progress. Actual position of the algorithm in the string is index i. Then the dictionary is searched for the best matching context, which is the part of the string *before* the index i, by right-to-left comparison. The best matching context is found. The context's neighbourhood is then searched for the best matching content, which is the part of the string *after* the index i. When it is (or is not) found, the length variable has value between zero and the maximum value defined by maximal bit size of a resulting triplet property, no matter if the found context would have even more symbols in common. This all is then handled to any triplet coder. The decompression works in the same manner, but it is not that important for this case. Even if the best matching content would fit longer length, it could not to preserve triplet coding functionality.

Features search for the second best content tries to solve this potential issue. It takes an adventage of a fact that the dictionary is, for any given index i, at the very same state for both compression and decompression. Otherwise the indices for context and contents would not find the same context-content pairs. If the content of the dictionary and every context-content pair are the same, search for the second best content would be the same too. And when the content is second best one, there is a decent probability that they share many symbols in common. This number, called *longest common prefix* (LCP) of two best contents, can be substracted from found length, increasing maximum boundary by the same value and potentially offering better compression ratio. Cases when the maximal allowed length is not enough and content shares even more symbols might not be frequent, but on contrary, this improvement do not affect compression ratio in any negative way, and speed probably not too, because all near contents are searched anyway.

Good way to demonstate it is to provide an example. Because this modification do not alter context behaviour, they are omitted.

Example 3.3.1
Construct a triplet from provided text and dictionary state. Then deconstruct the text from the triplet.

$$\texttt{...|}\underline{\texttt{alfaa}}\texttt{b}$$

	Context|Content
0	$\texttt{...|alfx}$
1	$\texttt{...|}\underline{\texttt{alfaa}}\texttt{xy}_{1^{st}}$
2	$\texttt{...|}\underline{\texttt{alfaa}}\texttt{xz}_{2^{nd}}$
3	$\texttt{...|alfab}$
4	$\texttt{...|ao}$

Search for the best matching content is done iteratively. At 0, the common length is 3. At 1, it increases to 5. The content at next index has the same length, so in normal compression method, it would not matter which one to take, but because now the second best matching content is important too, let the content at index 2 mark as the second best one. Third and fourth index length is shorter so do nothing. LCP(alfaaxy, alfaaxz) = 6, but length = 5. Length output should be substitution of LCP and original length, 5 - 6 = -1 in this case, but length can not be negative. So zero is outputted instead, forming triplet

$$(d, 0, b)$$

Decompression now. The input is previous triplet and dictionary looks the same.

At distance d from the context, there is a dictionary pair at index 2, which was previously selected as the best matching content. Search among contents have to be done to reveal the second best one. It is at index 1, sharing 6 symbols in common. LCP is 6, length from triplet is 0 so 6 symbols should be copied to the output, forming

$$\texttt{...alfaa}\boxed{\texttt{x}}\texttt{b|}$$

And that is definitively not what was inputted. The mistaken part was marking the second best matching content. Allowing it to have LCP with the best one greater than common length is wrong. To prevent this, the best matching content should be lexicographically smallest possible with maximal found length, and the second best matching content should be lexicographically smaller, disallowing LCP overflow and even length to be zero. This helps a lot to other triplet variations where the coder behaviour depends on zero or non-zero length property. In fact, when there is no content with common symbol, the search for the content during the decompression should not be done at all.

So once again the example 3.3.1. The dictionary looks like

	Context	Content
0	...	$\underline{\texttt{alf}}\texttt{x}_{2nd}$
1	...	$\underline{\texttt{alfaa}}\texttt{xy}_{1st}$
2	...	alfaaxz
3	...	alfab
4	...	ao

with best matching content at index 1. See, $\texttt{alfax}\underline{\texttt{y}} < \texttt{alfax}\underline{\texttt{z}}$. The second best matching content is chosen at index 0, having LCP equal to 3. Common length is 5, making 5 - 3 = 2. Outputted triplet is

$$(d, 2, b)$$

When decompressing, the same roles have to be followed. The best content, second best content and LCP are the same. Final length is 3 + 2 = 5 and that is exactly what it should be, output is

```
...alfaab|
```

Important remarks:

- Search for the 2nd best content is not needed when no symbols in common are found between best and actual content.

- The best matching content have to be lexicographically smallest among all contents fulfilling common symbols found.
- Second best content have to be lexicographically smaller than the first one.
- Length cannot be 0 if there are symbols in common between best and actual content.
- Contents are searched during decompression when the triplet length property is not zero to find 2^{nd} best content and to count longest common prefix.

Chapter 4

Implementation

This chapter deals with the way ACB comression method is implemented and what problems had to be challenged to achieve fully functional program.

Basic overview of development environment is presented in Section 4.1. External libraries and technologies used are in Section 4.2. Following Section 4.3 presents program architecture and design. The dictionary behaviour along with chosen backing data structures is presented in Section 4.4, after which the Section 4.5 focused on final triplet coding is located.

4.1 Overview

The program is written in Java language, version 8. There are no other ACB implementation is java, those mentioned in Chapter 2 are mainly in C or C++. It had to be built from scratch, including complete architecture design.

As a repository and version control system was used Git[6]. Build process system was chosen to be Maven[7] and whole development work was done in IDE IntelliJ IDEA[8] from JetBrains.

4.2 External libraries

There are many various libraries well optimised providing reliable and well documented functionality. It is good to use them rather than "inventing a wheel" again. Because this program consists of just one complex algorithm, no massive frameworks, databases or other business-heavy libraries are needed, I could get by general ones to outsource only simple thing. Their list is following.

[6] http://git-scm.com/
[7] http://maven.apache.org/
[8] http://www.jetbrains.com/idea

Commons CLI. [9] The Apache Commons CLI library provides an API for parsing *command line options* passed to programs. It's also able to print help messages detailing the options available for a command line tool. Commons CLI supports different types of options and is very well customizable.

Apache Log4j2. [10] Apache Log4j is a Java-based logging utility among many others *java logging frameworks*. Apache Log4j 2 is an upgrade to Log4j that provides significant improvements over its predecessor, Log4j 1.x, and provides many of the improvements available in Logback while fixing some inherent problems in Logback's architecture. The framework was rewritten from scratch and has been inspired by existing logging solutions, including Log4j 1 and java.util.logging. It has very manu new features and improvements, for the full list, see project site.

JUnit. [11] JUnit is a simple, open source framework to write and run repeatable *tests*. It is an instance of the xUnit architecture for unit testing frameworks. JUnit features include assertions for testing expected results, test fixtures for sharing common test data and test runners for running tests. JUnit was originally written by Erich Gamma and Kent Beck.

4.3 Architecture

This ACB implementation was probably first of a kind in java. Completely new design was made to work fast, fulfil all the ACB compression method requirenments and to offer potentially extensible environment for further improvements. Architecture research and design was made mainly in topics showed by following sections.

4.3.1 Functional parts provider

At the beginning, the issue was to how to decouple execution logic from initialization logic, so the algorithm runs solely with configured components and is not delayed by initialization during runtime.

Solution was in form of data provider, that takes parsed command line arguments, initialize proper instance constructors and offers them under unified interface methods. Of course that every functional module, such as dictionary or triplet coder, have to share the same class interface. The compression algorithm can then treat all the instances the same way and the data provider gives the right instance every time it is asked.

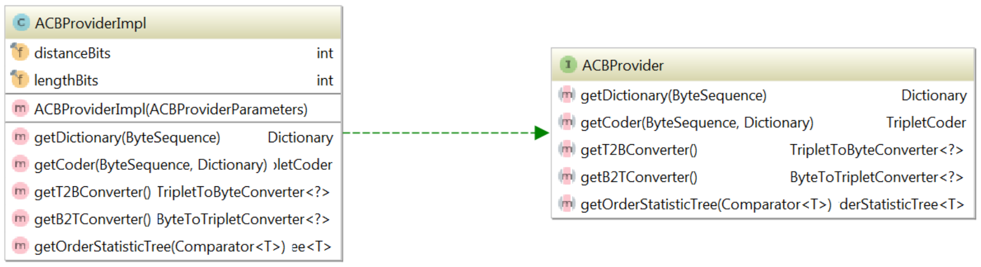
[9] http://commons.apache.org/cli/

[10] http://logging.apache.org/log4j/2.x/

[11] http://junit.org/

The interface ACBProvider provides methods to get all the important components of the compression method, those are dictionary, coder, converter between bytes and triplets, and the backing dictionary structure. Class ACBProviderImpl has to be instantiated by constructor, by which all the parsed information from user input are handled. In constructor, it creates instances of constructor method references of all those important parts. Then, when particular get method is called, it forwards its parameters to constructor reference and return the intended class instance. UML diagram is shown at figure 4.1.

Figure 4.1: UML diagram of functional parts provider.


ACBProviderImpl
distanceBits int
lengthBits int
ACBProviderImpl(ACBProviderParameters)
getDictionary(ByteSequence) Dictionary
getCoder(ByteSequence, Dictionary) ›letCoder
getT2BConverter() TripletToByteConverter<?>
getB2TConverter() ByteToTripletConverter<?>
getOrderStatisticTree(Comparator<T>) ʒe<T>
ACBProvider
getDictionary(ByteSequence) Dictionary
getCoder(ByteSequence, Dictionary) TripletCoder
getT2BConverter() TripletToByteConverter<?>
getB2TConverter() ByteToTripletConverter<?>
getOrderStatisticTree(Comparator<T>) derStatisticTree<T>

4.3.2 Data stream

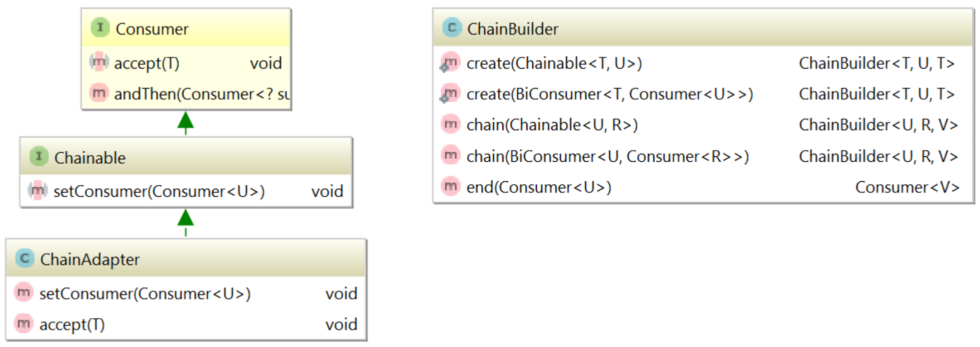
Issue here was to design program data flow such that it is as fast as possible, all the components should start their process immediately, while keeping the communication between parts general and extensible. Inspiration was taken from Java Stream API that becomes available in Java 8. It first creates whole streamline, sometimes also called a pipe, with the input logic at the beginning and output logic, called terminator, at the end.

The object composition is chained using ChainBuilder forming a chain of processes in which the data flow freely. There is a contract that every method that wants to be part of a chain have to have two parameters, as input parameter of any type that it can start the work with, and the second parameter is a Consumer that is able to consume the output, forwarding it to following chain segment. This ensures every part of the chain to be fully responsible over the handling with object input and to output the result whenever it want and mainly how many times it wants, Consumer is always there forwarding everything.

The flow od data, or the chain, is illustrated below. Brief information describes the purpose of the process, arrow indicates direction and text above arrow tell what type of data flows there.

Compression: Read from file $\xRightarrow{byte[]}$ parse if needed $\xRightarrow{byte[]}$ ACB compression $\xRightarrow{triplet}$ triplet coding $\xRightarrow{byte[]}$ Write to file.

Figure 4.2: UML diagram of chain manager.


Consumer
accept(T) void
andThen(Consumer<? su
Chainable
setConsumer(Consumer<U>) void
ChainAdapter
setConsumer(Consumer<U>) void
accept(T) void
ChainBuilder
create(Chainable<T, U>) ChainBuilder<T, U, T>
create(BiConsumer<T, Consumer<U>>) ChainBuilder<T, U, T>
chain(Chainable<U, R>) ChainBuilder<U, R, V>
chain(BiConsumer<U, Consumer<R>>) ChainBuilder<U, R, V>
end(Consumer<U>) Consumer<V>

Decompression: Read from file $\xrightarrow{byte[]}$ triplet decoding $\xrightarrow{triplet}$ ACB decompression $\xrightarrow{byte[]}$ concatenate if needed $\xrightarrow{byte[]}$ Write to file.

UML diagram of ACB chain manages is shown in Figure 4.2.

4.3.3 Input Output

In previous section was said that the chain is created and terminated by IO logic. For this purpose, the class ACBFileIO was created. It has no interface. It contains methods to access files in various ways for both opening and closing. The list of methods is:

ACBFileIO. Constructor, it has one argument variant with integer, the number indicated a block size to which is opened file parsed and forwarded, and no argument variant, which initialize the class so the whole file is forwarded as byte array.

openParallel. This method opens a file in mode allowing paralelism, it contains ExecutorService that handles creation of threads. Those threads hosts next processes, namely ACB compression. Parallelization was not completes, because triplet coder - adaprive arithmetic coding - do not allow processing in parallel mode.

openParse. Opens file in regular mode, parsing it if specified by constructor parameter. Parsed blocks are forwarded to next chain segment.

saveParse. The opposite process, concatenating and saving bytes into file.

openObject and saveObject Unused methods, they were saving final ArrayList of Bytes by java serialization, not by concatenating inner parts, causing memory overhead. It is deprecated, openArray should be used.

openArray. Method used for decompression, opens array of bytes.

saveArray. Terminal method of compression, saves stream of bytes.

For opening and closing purposes, bytes are manipulated as java ByteBuffer for the better support of java IO api.

4.4 Dictionary

ACB dictionary was introduced in Section 1.2.1 and closely analyzed in Section 3.1. Now lets have a look at the implementation facts and issues.

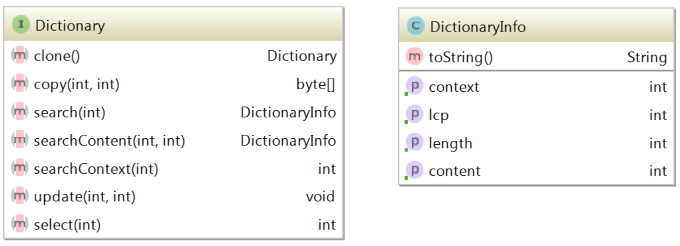
The dictionary consists of context-content pairs. This functionality, along with searches and updates, is managed by backing data structure. Dictionary class is wrapping this tree data structure, enhancing it and providing some dictionary-related logic.

The *sliding window* is represented by byte array wrapping object. It was inspired by String and wrapped byte array inside. Interface ByteSequence is similar to CharSequence, providing uniform access to bytes. ByteArray is the same as String, immutable array of bytes. If is used at a time of compression, because whole input sentence could be read and is known at the start. And if a programmer wants to build long Strings efficiently, he uses StringBuilder, the same is used is case of decompression - byte array is slowly build by copying symbols from dictionary or adding symbols from triplets. ByteBuilder was used with exactly same inner logic as StringBuilder, including dynamically growing arrays of bytes.

The dictionary does not contain any part of the original text, only mentioned sliding window. All context-content pair values are represented as indices to this sliding window. This index is of type integer, as integer's value is large enough to index any type of text, nevertheless block partitioning is probably done during the input part.

The dictionary interface provides following operations. To be more obvious, UML diagram is presented in Figure 4.3.

Figure 4.3: Dictionary interface.



Dictionary	
clone()	Dictionary
copy(int, int)	byte[]
search(int)	DictionaryInfo
searchContent(int, int)	DictionaryInfo
searchContext(int)	int
update(int, int)	void
select(int)	int

DictionaryInfo	
toString()	String
context	int
lcp	int
length	int
content	int

clone. Cloning dictionary is useful only for testing, cloning every state of a dictionary for later comparison with dictionary states created by decompression.

copy. This method have two integer attributes specifying how many symbols to copy from which index. It is used only for decompression.

search. This is the most used method. It gets index number from its call parameter, calls method searchContext and with obtained best matching context, it calls searchContent returning its return value.

searchContext. Delegates call to rank method of a backing ordered statistical tree.

searchContent. Searches the defined proximity of given context for the best matching content. This method is overriden in inherited dictionary structure to the base one, handling differently the second best matching contents, mentioned in Section 3.3.1.

update. Called after every triplet was handled, it updates backing dictionary tree by values that have been encoded/decoded.

select. Delegates call to select method of a backing ordered statistical tree. It is used only for best matching content handling, as the select result is needed to know where the search for second best matching content was originally made.

4.4.1 Dictionary order

ACB dictionary has unusual ordering for the context part. It orders by right-to-left comparison, for example *za* is smaller than *az*. There were two options how to handle this type of ordering.

First one was to wrap every index into data structure extending Comparable<T> where T is that wrapper. Compare method would have access to the dictionary's sliding window because the reference would have been given to the wrapper object, that is why it was created. Then compare method would access it every time it is called, traversing backward until it finds inequality. This variant was memory consuming, because every index stored had to be wrapped by object referencing to the text.

The second variant which I have chosen is by creating custom Comparator<Integer> with access to the sliding window and handling backward traversing. It means backing tree structures does not have comparable keys, but comparator is given to them instead. This variant looked to me more convenient, but primitive integer indices has to be also substituted by Integer object for the possibility to be part of the comparator and backing tree, as java do not allow primitives to do so.

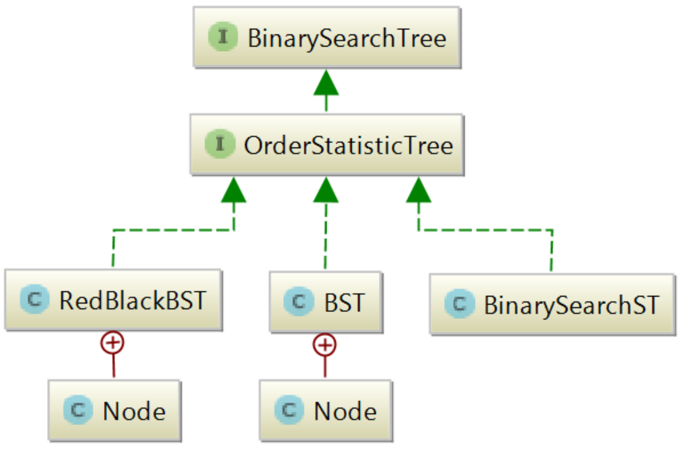
Another issue was that in some cases, when two contexts were too much similar, it took a lot of time to decide which one was smaller or greater, resulting into almost no impact as they were probably siblings anyway. It makes sence to bound those comparisons by some constant. When two contexts are similar up to some point, they are called identical and are stored to the dictionary in deterministic order. I choose this constant to be 10 and when reached, I compared them based on the context length, because every context has different length.

4.4.2 Ordered statistical trees

We know from analysis that self-balanced tree structures are best for backing the dictionary. It have to be adjusted by dynamic order statistics to provide full functionality for the dictionary. The search trees are defined by set of nodes containing keys and corresponding values, usually written as Tree<Key, Value>. The Key is crucial for us, it points to the original text. Its value is exactly the index at which was the sliding window positioned at the time of processing it. The content lies right from index, context lies left from index. By key, the dictionary is able to access both. So what use Value for? **Value is not needed and shell be removed from any backing tree to save some memory.**

There are three backing dictionary structures implemented in the program. Their inheritance relationships are showed in Figure 4.4. All three have been taken from [19] and slightly refactored not to contain values, just keys, and not to require keys to be comparable, but rather use comparator.

Figure 4.4: Backing trees relationship.



4.4.2.1 BinarySearchST

This is actually not a tree, but *binary search symbol table.* The underlying data structure is array, with the keys kept in order. For the insert operation, all larger keys are moved to make room for inserted value. Search is done by binary search in sorted array. This implementation is apparently not effective, it was included to see and compare trees and arrays.

4.4.2.2 BST

This implementation uses an unbalanced *binary search tree.* There is an inner private class to define nodes in BST. Each node contains a key, a value, a left link, a right link, and a node count. The left link points to a BST for items with smaller keys, and the right link points to a BST for items with larger keys. The instance variable N gives the node count in the subtree rooted at the node.

4.4.2.3 RedBlackBST

This implementation uses a *left-leaning red-black binary search tree.* It is also called 2-3 search tree, because it takes the advantage of B Trees, but is implemented as a red-black tree. The basic idea behind red-black binary search trees is to encode 2-3 trees by starting with standard BSTs (which are made up of 2-nodes) and adding extra information to encode 3-nodes. We think of the links as being of two different types: red links, which bind together two 2-nodes to represent 3-nodes, and black links, which bind together the 2-3 tree. Specifically, we represent 3-nodes as two 2-nodes connected by a single red link that leans left. We refer to BSTs that represent 2-3 trees in this way as red-black BSTs. Given any 2-3 tree, we can immediately derive a corresponding red-black BST, just by converting each node as specified. Conversely, if we draw the red links horizontally in a red-black BST, all of the null links are the same distance from the root, and if we then collapse together the nodes connected by red links, the result is a 2-3 tree. Node and correspondence between trees is shown in Figure 4.5[12] [19].

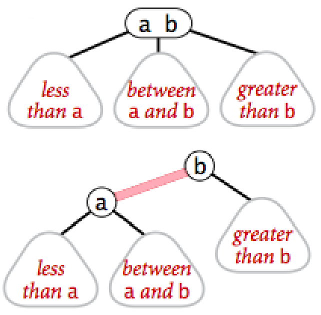
4.5 Coder

Triplet coder and decoder are different. They are represented by classes ByteToTripletConverter and TripletToByteConverter. They serves as a wrapper to any coder, handling communication with preceding and following chain segments. They create coder for every element of triplet, allowing adaptive arithmetic coding to have separated frequency table for each alphabet thus

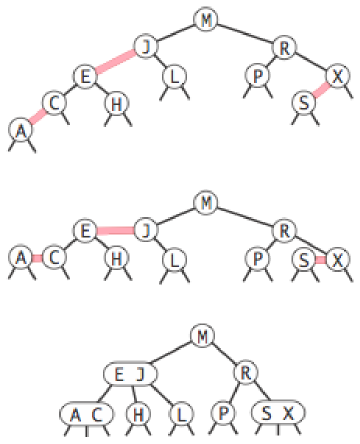
[12]Source: http://algs4.cs.princeton.edu/33balanced/images/redblack-encoding.png

Figure 4.5: 2-3 tree correspondence to red-black tree.

(a) 3-node



(b) Transition between trees.



achieving better compression. Two variants of triplet coding were implemented.

4.5.1 Adaptive Arithmetic Coding

Opensource project Nayuki [20] was used and adjusted for arithmetic coding purpose.

Two pairs of command-line programs fully demonstrate how this software package can be used to encode and decode data using arithmetic coding. One pair of programs is the classes ArithmeticCompress and ArithmeticDecompress, which uses a static frequency table. The other pair of programs is the classes AdaptiveArithmeticCompress and AdaptiveArithmeticDecompress, which uses an adaptive/dynamic frequency table and was used for ACB triplet coding.

The classes ArithmeticCoderBase, ArithmeticEncoder, and ArithmeticDecoder implement the basic algorithms for encoding and decoding an arithmetic-coded stream. The frequency table can be changed after encoding or decoding each symbol, as long as the encoder and decoder have the same table at the same position in the symbol stream. At any time, the encoder must not attempt to encode a symbol that has a zero frequency.

Objects with the interface FrequencyTable keep track of the frequency of each symbol, and provide cumulative frequencies too. The cumulative frequencies are the essential data that drives arithmetic coding.

The classes BitInputStream and BitOutputStream are bit-oriented I/O streams, analogous to the standard bytewise I/O streams. However, since they use an underlying bytewise I/O stream, the bit stream's total length is always a multiple of 8 bits [20].

4.5.2 Bit Array Composing

Different approach was to implement simple bit array composer and decomposer in classes BitArrayComposer and BitArrayDecomposer. It just takes the triplet value and stores it into byte array, buffering and aligning bits. It was implemented for comparison with adaptive arithmetic coding, to see how much more efficient it is than using no coding at all. It's functionality is supported by BitStreamInputStream and BitStreamOutputStream classes that implements an enhanced I/O stream which allows to read a stream of bit fields ranging in size from 1 bit (a true bit stream) to 32 bits (a stream of integers). The size of the current bitfield can be changed at any point while reading the stream.

4.6 Tests

Unit tests are implemented using JUnit test framework. The tests are written so that all the parameters and the whole compression and decompression chain is tested. It may be mor eclear to call them functional tests rather that unit tests, as thez do not test small units, but mainly whole consistency, data flow and completeness.

Various mocked objects and ACB data structure wrappers were used to control the flow and the processes correctly.

Chapter 5

Measurements

This chapter shows results and outcomes of implementations presented in previous chapters. Experiments and performance measurements are covered, showing what king of implementation is better. Special attention was given to new improvement presented in Section 3.3.1 and also to comparison of new java implementation and C++ ones presented in Chapter 2.

Memory management was tested and observed by and observed by JVM Debugger Memory View for IntelliJ IDEA plugin, time was measured by internal JVM clock. Correctness of compression and decompression was tested by comparing original file and file after compression and decompression for all the combinations of program modules.

Testing platform was Intel®Core^{TM}i5-430M, 2,26 GHz, 3 MB L3 cache, quadro core, with RAM 4 GB DDR3, Windows 7 64-bit architecture, Java 64-bit 1.8.0_102.

5.1 Used corpuses

Compression corpuses are used for even comparison over manifold set of data. Their advantage is that they contains various commonly used data sets that tests algorithms from every side of usage.

Only Calgary corpus was used in this measurements, but let mention more of them for completeness.

5.1.1 Calgary Corpus

The Calgary Corpus was founded by Ian Witten, Tim Bell and John Cleary at University of Calgary in 1987. It contains 18 files of 9 different types with complete size 3,266,560 bytes. A Table 5.1 contains all of those files descriptions.

Table 5.1: Calgary corpus description

File name	Size [B]	Description
bib	111,261	bibliographic references
book1	768,771	english text
book2	610,856	english text
geo	102,400	geophysical data
news	377,109	news articles
obj1	21,504	executable code
obj2	246,814	executable code
paper1	53,161	english text
paper2	82,199	english text
pic	513,216	bitmap black and white picture
progc	39,611	C source code
progl	71,646	Lisp source code
progp	49,379	Pascal source code
trans	93,695	transcript of a terminal session

5.1.2 Cantenbury Corpus

The Canterbury Corpus was published by Ross Arnold and Tim Bell in 1997. The aim was to replace outdated Calgary Corpus and to provide more relevant testing for new compression algorithms. The files were selected based on their ability to provide representative performance results. It contains 11 files with size of 2,810,784 bytes showed in Table 5.2.

Table 5.2: Cantenbury corpus description

File name	Size [B]	Description
alice29.txt	152,089	English text
asyoulik.txt	125,179	Shakespeare
cp.html	24,603	HTML source code
elds.c	11,150	C source code
grammar.lsp	3,721	LISP source code
kennedy.xls	1029,744	Excel spreadsheet
lcet10.txt	426,754	technical writing
plrabn12.txt	481,861	poetry
ptt5	513,216	CCITT test set
sum	38,240	SPARC Executable
xargs.1	4,227	GNU manual page

5.1.3 Prague Corpus

Prague corpus was created as a part of master thesis by Jakub Řezníček in 2010. This corpus contains 30 files of the total size 58,265,600 bytes, being the largest from those three corpuses. Files are also described in Table 5.3.

Table 5.3: Prague corpus description

File name	Size [B]	Description
abbot	349,055	binary file
age	137,216	spreedsheet
bovary	2,202,291	german text
collapse	2,871	JavaScript source code
compress	111,646	HTML source code
corilis	1,262,483	graphics
cyprus	555,986	XML data
drkonqi	111,056	binary file
emission	2,498,560	database file
firewrks	1,440,054	audio file
flower	10,287,665	audio file
gtkprint	37,560	binary file
handler	11,873	Java source code
higrowth	129,536	spreedsheet
hungary	3,705,107	XML file
libc06	48,120	binary file
lusiadas	625,664	portuguese text
lzfindmt	22,922	C source code
mailflder	43,732	Python source code
mirror	90,968	binary file
modern	388,909	swedish text
nightsht	14,751,763	graphics
render	15,984	C++ source code
thunder	3,172,048	audio file
ultima	1,073,079	english text
usstate	8,251	Java source code
venus	13,432,142	graphics
wnvcrdt	328,550	database file
w01vett	1,381,141	database file
xmlevent	7,542	PHP source code

5.2 Dictionary structure experiments

This section presents tests and measurements over all three implemented data structures presented in Section 4.4.2. Those are binary search symbol table, binary search tree without self-balancing property and finally self-balancing 2-3 tree (or red-black tree). The Table 5.4 presents it.

BinarySearchST is implemented by array that have to shift the data larger than inserted key every time the dictionary is updated. It explains the longest time and compression ratio.

BST, as binary search tree, is quite more efficient, but in few cases, when the inserting output is unfortunate, it reporting strong behaviour changes.

RedBlackBST is the most stable one for its balancing property and optimised operations.

5.3 Dictionary memory consumption

Context based dictionary compression methods have big requirement for memory allocation during its runtime. In this field, array-based structures have advantage over the tree based, because trees needs internal node data structure

Table 5.4: Time complexity of dictionary structures

	Compression time [μ s]		
File	BinarySearchST	BST	RedBlackBST
bib	449,055	360,416	266,125
book1	4,014,775	2,516,040	1,890,435
book2	3,769,551	2,100,647	1,850,437
geo	326,017	220,143	180,670
news	1,619,001	1,001,431	776,641
obj1	62,554	52,789	44,658
obj2	140,966	125,920	99,647
paper1	111,257	70,102	55,786
paper2	250,123	190,019	149,600
pic	10,440,054	6,450,179	3,982,702
progc	87,593	67,233	50,833
progl	187,400	177,926	135,074
progp	130,707	89,904	80,414
trans	222,410	201,405	130,956

to store keys, links and other properties. Table 5.5 shows how memory demanding were those three dictionary backing structures.

In general, the heap memory used by a Java object in Hotspot consists of:

- an object header, consisting of a few bytes of ”housekeeping” information,
- memory for primitive fields, according to their size,
- memory for reference fields (4 bytes each),
- padding: potentially a few ”wasted” unused bytes after the object data, to make every object start at an address that is a convenient multiple of bytes and reduce the number of bits required to represent a pointer to an object.

A single-dimension array is a single object. As expected, the array has the usual object header. However, this object head is 12 bytes to accommodate a four-byte array length. Then comes the actual array data which, as expected, consists of the number of elements multiplied by the number of bytes required for one element, depending on its type. The memory usage for one element is 4 bytes for an object reference. List of the memory usage of primitive types is not a case for ACB dictionary so it is not discussed here.

Table 5.5: Memory allocation of dictionary structures

		Memory allocation [B]		
File name	Size [B]	BinarySearchST	BST	RedBlackBST
bib	111 261	445208	1335256	1780200
book1	768 771	3075248	9225372	12300360
book2	610 856	2443584	7330392	9773720
geo	102 400	409760	1228920	1638424
news	377 109	1508600	4525432	6033768
obj1	21 504	86176	258168	344088
obj2	246 814	987416	2961888	3949048
paper1	53 161	212808	638056	850600
paper2	82 199	328960	986512	1315208
pic	513 216	2053024	6158712	8211480
progc	39 611	158608	475456	633800
progl	71 646	286744	859872	1146360
progp	49 379	197680	592672	790088
trans	93 695	374944	1124464	1499144

5.4 Triplet component experiments

This section experiments with adjustable triplet components, which are distance and length. Alphabet size representing symbol is not customizable, because it does not make sense.

Distance means how far is search for best matching contents done after the best matching context is found. Increasing distance parameter insignificantly increases search time, but significantly increases probability at which the better content is found, reducing compression ratio.

Length stands for the maximum length of symbols shared in common between actual text content and best matching content, saving both time and compression ratio.

Table showing relationship between compression ratio and distance bits is showed in Table 5.6 and relationship between compression ratio and length bits is showed in Table 5.7.

5.5 Triplet coding

There were implemented all known triplet forms to test how they do differ in compression ratio. They were named after their discoverers or implementators. Implemented triplet forms are:

- Simple - default triplet (l, d, c),

Table 5.6: Relationship between compression ratio and distance property

	Distance property [b]				
File	3	5	7	9	11
bib	0.289	0.286	0.282	0.276	0.297
book1	0.358	0.339	0.360	0.344	0.323
book2	0.323	0.305	0.301	0.300	0.286
geo	0.641	0.596	0.604	0.586	0.573
news	0.341	0.351	0.345	0.332	0.347
obj1	0.487	0.460	0.517	0.500	0.518
obj2	0.357	0.352	0.344	0.334	0.332
paper1	0.344	0.350	0.327	0.353	0.347
paper2	0.356	0.340	0.337	0.323	0.338
pic	0.108	0.103	0.114	0.111	0.108
progc	0.336	0.333	0.331	0.335	0.330
progl	0.228	0.240	0.232	0.218	0.232
progp	0.224	0.219	0.221	0.219	0.221
trans	0.203	0.187	0.191	0.196	0.204

Table 5.7: Relationship between compression ratio and length property

	Length property [b]				
File	3	5	7	9	11
bib	0.329	0.301	0.300	0.301	0.318
book1	0.373	0.373	0.369	0.377	0.382
book2	0.341	0.327	0.320	0.326	0.322
geo	0.657	0.633	0.662	0.636	0.651
news	0.385	0.367	0.359	0.365	0.372
obj1	0.539	0.509	0.500	0.524	0.547
obj2	0.405	0.359	0.354	0.358	0.357
paper1	0.382	0.363	0.359	0.360	0.368
paper2	0.367	0.363	0.356	0.366	0.371
pic	0.149	0.124	0.116	0.108	0.110
progc	0.379	0.345	0.343	0.362	0.362
progl	0.285	0.249	0.239	0.241	0.258
progp	0.277	0.236	0.231	0.236	0.243
trans	0.273	0.216	0.206	0.208	0.211

- Salomon - the triplet (1, l, d) or (0, c),
- Salomon2 - the triplet (1, l, d, c) or (0, c),
- Valach - the triplet (l, c) or (l, d, c),
- LCP - longest common prefix is used.

It is showed that for some data, the LCP improvements really works, but it can not be generalized. Data where there are more repetitive strings, the LCP improvement seems to work well, though.

Table 5.8: Forms of triplet representations

		Compression ratio			
File	Simple	Salomon	Salomon2	Valach	LCP
bib	0.314	0.312	0.298	0.304	0.308
book1	0.387	0.386	0.381	0.376	0.374
book2	0.342	0.341	0.340	0.329	0.320
geo	0.699	0.691	0.682	0.652	0.656
news	0.377	0.374	0.372	0.370	0.364
obj1	0.532	0.529	0.523	0.522	0.519
obj2	0.381	0.371	0.366	0.362	0.350
paper1	0.376	0.373	0.367	0.367	0.367
paper2	0.381	0.380	0.377	0.376	0.373
pic	0.126	0.123	0.116	0.115	0.115
progc	0.370	0.368	0.362	0.358	0.352
progl	0.251	0.254	0.252	0.248	0.248
progp	0.254	0.254	0.248	0.237	0.236
trans	0.218	0.216	0.216	0.210	0.205

Conclusion

This thesis deals with not very well examined context based compression method named ACB after its inventor George Buzanovsky. The algorithm was described in detail, followed by list of contributors and their implementations of this method. After, depth analysis of possible data structure improvements was made, along with other initiatives and observations. The outcome of this thesis is functional implementation of the ACB algorithm in java language, opening potential possibilities of future improvements. All relevant elements were implemented and they underwent technical measurements such as compression efficiency, memory consumption and time requirements.

Future work

This java implementation does still have some weak spots, such as linked leaves in dictionary backing structure or more sophisticated adaptive arithmetic coding. But brand new implementation in java opens a lot of options, the most significant ones are easy algorithm parallelization using java threads. It was already slightly designed, but not finished nor tested.

Bibliography

[1] Buyanovsky, G. *Associative coding.* Monitor, 10 - 22 pp.

[2] Salomon, D. *Data Compression: The Complete Reference.* Springer-Verlag, fourth edition.

[3] Large text compression benchmark. Available from: mattmahoney.net/dc/text.html#2185

[4] Cerman, L. Acb compression algorithm.

[5] Excom library. Available from: http://www.stringology.org/projects/ExCom/

[6] Šimek, F. Data compression library.

[7] Valach, M. Effcient implementation of ACB compression algorithm for ExCom library.

[8] Fredkin, E. *Trie memory.* ACM Computer Survey, 490 - 499 pp.

[9] Wiener, P. *Linear pattern matching algorithms.* Switching and Automata Theory, 1 - 11 pp.

[10] Ukkonen, E. *On-line construction of suffix trees*, volume 14. Algorithmica, 249 - 260 pp.

[11] Gene Myers, U. M. *Suffix arrays: a new method for on-line string searches*, volume 22. SIAM Journal Computing, 935 - 948 pp.

[12] Pan Ko, S. A. *Space efficient linear time construction of suffix arrays*, volume 2676. In Combinatiorial Pattern Matching (CPM 03).LNCS, 203 - 210 pp.

[13] Cormen, T.; Leiserson, C.; Rives, R.; et al. *Introduction to Algorithms*. MIT Press and McGraw-Hill, third edition, ISBN 0-262-03384-4.

[14] Gonnet, G. *Optimal Binary Search Trees*. Scientific Computation.

[15] Sadgewick, R. *Algorithms*. Addison-Wesley, ISBN 0-201-06672-6, 199 pp.

[16] Bayer, R.; McCreight, E. *Organization and Maintenance of Large Ordered Indices*. Boeing Scientific Research Laboratories.

[17] Knuth, D. *Sorting and Searching*, volume 3. Addison-Wesley, second edition, ISBN 0-201-89685-0.

[18] Martin, G. N. N. *An algorithm for removing redundancy from a digitized message*. Video & Data Recording Conference.

[19] Sadgewick, R.; Wayne, K. *Algorithms*. Addison-Wesley Professional, fourth edition, ISBN 032157351X 9780321573513. Available from: `http://algs4.cs.princeton.edu`

[20] Nayuki. Reference arithmetic coding. Available from: `https://www.nayuki.io/page/reference-arithmetic-coding`

Appendix A

Acronyms

ACB	Asociative Coder of Buyanovsky
API	Application Programming Interface
AVL	Adelson-Velsky and Landis
BST	Binary Search Tree
EOF	End Of File
IDE	Integrated Development Environment
IO	Input/Output
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LCP	Longest Common Prefix
ST	Symbol Table
VCS	Version Control System

Appendix B

Contents of enclosed CD

```
readme.txt ...................... the file with CD contents description
license.txt ......................... the file with licence specification.
src ................................... the directory of source codes
dist ........................................ distributed jar file
text ...................................... the thesis text directory
    src ................ the directory of LaTeX source codes of the thesis
    thesis.pdf ......................... the thesis text in PDF format
```

Appendix C

User manual

This appendix summarizes usage of implemented ACB compression method by console parameters.

```
usage: acb.jar input output [options]
input - input file or directory
-af,   –arith-freq ⟨freq⟩      ⟨freq⟩ is comma separated array of integers defining
                               init values
                               of arithmetic coding frequency table
                               (default is 45,13,10,7,5,4,1...)
-bs,   –bit-stream-array       no coding is used for triplets (default is adaptive arithmetic coding)
-d,    –distance ⟨N⟩           N bits used for distance triplet element (default is 6)
                               maximal context-content distance is 2(̂N - 1)
-de,   –decompress             decompress input (default is to compress)
-ds,   –dict-struct ⟨struct⟩   ⟨struct⟩ represents data structure used
                               in dictionary (default is red_black)
                               values = [RED_BLACK, BST, ST]
-h,    –help                   print this help
-l,    –length ⟨N⟩             N bits used for length triplet element (default is 4)
                               maximal length is 2(̂N)-1
-log,  –log-level ⟨level⟩      sets logging level of the application
-m,    –measure                measured program process data printed to file ⟨out⟩
                               or to standard output if no file specified
-tc,   –triplet-coder ⟨coder⟩  ⟨coder⟩ represents triplet coding strategy (default is simple)
                               values = [SALOMON, SALOMON2, SIMPLE, VALACH, LCP]
```