CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Effective implementation of state-of-the-art variants of the ACB compression method for the ExCom library
Student:	Bc. Adam Léhar
Supervisor:	Ing. Radomír Polách
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Research the current ACB compression method implementation in the ExCom library [1].
Research structures for indexing context and content.
Choose appropriate structures for indexing context and content and implement them with consideration to ACB compression method requirements.
Create effective implementation of the ACB compression method using these structures.
Experiment with ACB compression method variants given by the supervisor.
Test and compare new and previous implementations of ACB compression methods for speed and compression ratio.

References

[1] ExCom. Retrieved from http://www.stringology.org/projects/ExCom/

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 19, 2016

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Master's thesis

Effective implementation of state-of-the-art variants of the ACB compression method for the ExCom library

Bc. Adam Léhar

Supervisor: Ing. Radomír Polách

6th May 2016

Acknowledgements

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

Declaration

In Pardubice on 6th May 2016 . . . .

Citation of this thesis

Léhar, Adam. *Effective implementation of state-of-the-art variants of the ACB compression method for the ExCom library.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Tato práce se zabývá analýzou algoritmu ACB a návrhem nových struktur pro indexování contextu a contentu s ohledem na vylepšení kompresního poměru. V této práci jsou navržena tři vylepšení, která jsou implementována a ověřena experimentálním měřením. Jedno z těchto vylepšení, které vedlo ke zlepšení kompresního poměru, je přidáno do knihovny kompresních algoritmů ExCom.

Klíčová slova algoritmus ACB, komprese, dekomprese, kompresní poměr, knihovna ExCom

Abstract

This thesis deals with analysis the ACB algorithm and design new structures for indexing context and content to improve the compression ratio. In this thesis are design three improvements that are implemented and verified by experimental measurement. One of these improvements, which improved the compression ratio, is included to the compression library ExCom.

Keywords ACB algorithm, compression, decompression, compression ratio, ExCom library

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Motivation

In modern technologies and computers is technology moving forward every day. Not only processing power is growing. Amount of generated and processed data is growing too. These data have to be stored somewhere.

In present days a huge amount of data is processed and stored in cloud. This puts a large memory demands on the data center, where these data is stored. Storing huge amount of data is expensive, it is advantageous to minimize amount of data to store.

One way to solve the problem with growing an amount of data that have to be store is decrease their amount without loss any information. One of the options can be lossless compression.

There are a lot of compression algorithms with difference principles. Statistical methods are Huffman coding, arithmetic coding, etc. Dictionary based methods are LZ77, LZ78, etc. Context methods are PPM, DCA, ACB, etc. All these compression methods has same characteristic. Size of output data is smaller than size of input data. After inverse transformation is output data identical with original data.

In the future an amount of data will be growing faster than nowadays. Data compression will be significant part of data storing.

Main goals

The main goal this master thesis is research structures for indexing context and content. The compression ratio can be improvement.

This thesis follow master thesis Efficient implementation of ACB compression algorithm for ExCom library [11] published by Michal Valach in 2011.

Outcome this thesis will be analysis of improvement original algorithm ACB published by George Buyanovsky. Main goal is decrease compression ratio. This improvements will be implemented in C++ and integrated to compression library ExCom [10]. Experimental measurement compare results previous and new implemented version of algorithms in ExCom library.

ExCom library

Compression library ExCom [10] was developed by Filip Šimek in his master thesis Data compression library [9]. This compression library was extended by Jakub Řezníček in his master thesis Corpus for comparing compression methods and an extension of a excom library [7]. Published licence of this library is GNU LGPL version 3.

Compression library ExCom (Extensible Compression Library) is gcc library of collected data compression algorithms. The idea behind is to have one source of data compression algorithms together with benchmarking environment.

Main benefit this library is identically interface for experiments and having access to the other algorithms implemented in this library. In the case different implementations of algorithms with different interface is problem to compare results measured on this implementations.

The ExCom library is well designed for extensions by new implemented algorithms. The library is implemented in C++ programming language for its speed reasons and it is prepared to be used in multi-threaded programs. The library has a built-in mechanism used for IO operations available for many types of streams and time measurement mechanism with microseconds precision.

Thesis organization

Introduction explain motivation why is important to use the lossless compression. There are described main goals this master thesis and introduced to compression library ExCom.

Chapter 1 describes compression algorithm ACB. This chapter explains main idea this algorithm, show compression and decompression process on practical examples.

Chapter 2 describes previous implementation of ACB algorithm with their advantages and disadvantages.

Chapter 3 analyse compression algorithm ACB. This chapter describes data structures for store context-content pairs. End of this chapter describes new variants of algorithm which will be implemented in this thesis.

Chapter 4 describes algorithm implementation include new variants of algorithm. This chapter describes data model and integration new implemented module to compression library ExCom.

Chapter 5 show results of experimental measurements. This chapter compare results new implemented variants of ACB algorithm with previous implementation of ACB algorithm in ExCom library.

Conclusion evaluate outcome of this master thesis.

Chapter 1

Algorithm ACB

1.1 Definitions

Definition 1.1 (Alphabet)
Alphabet is a finite set of symbols.

Definition 1.2 (Symbol)
A symbol is an atomic element of an alphabet.

Definition 1.3 (String)
String is a sequence of symbols from alphabet.

Definition 1.4 (Empty string)
Empty string is the string with zero length. Empty string is denoted as ϵ.

Definition 1.5 (Code word)
Codeword is a sequence of bits.

Definition 1.6 (Code)
Code K is a triple $K = (S,\ C,\ f)$, where:

- *S is a finite set of source units,*
- *C is a finite set of codewords (code units),*
- *f is an injective mapping $S \mapsto C^+$.*

$$\forall a_1, a_2 \in S, a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$$

Definition 1.7 (Compression)
Data compression is a process which tries to find a different representation of original data intending to reduce the number of bits needed to represent the data.

Definition 1.8 (Decompression)
Data decompression is a process complementary to data compression which applies transformation, inverse to that used in data compression, to compressed data.

Definition 1.9 (Lossless compression method)
Lossless compression method is a compression method whose output after decompression of compressed data is always exactly the same as the original data because compressed data retains all necessary information to restore the original data.

Definition 1.10 (Lossy compression)
Lossy compression method is a compression method whose output after decompression of compressed data may not be exactly the same as the original data because some information carried by the original data is claimed unimportant and is lost during compression.

Definition 1.11 (Compression ratio)
Compression ratio is a ratio between compressed data size and original (uncompressed) input data size obtained from the following equation.

$$compression\ ratio = \frac{compressed\ data\ size}{uncompressed\ data\ size}$$

Definition 1.12 (Context)
Every character c_i in the input text can be represented as a position in the text. Any substring α in the left vicinity from position in this text is called context.

Definition 1.13 (Content)
Every character c_i in the input text can be represented as a position in the text. Any substring β in the right vicinity (including c_i) is called content.

Definition 1.14 (Longest common prefix)
The length of the longest common prefix of two strings $A[0..m)$ and $B[0..n)$ is the largest integer $l \leq min\{m, n\}$ such that $A[0..l) = B[0..l)$.

Definition 1.15 (Corpus)
A corpus is a set of various files, used for benchmarking compression methods.

Definition 1.16 (Grandparent of node)
Grandparent of node is the parent of its parent, if it exists.

Definition 1.17 (Uncle of node)
If the parent of node is a left child of its parent, then the uncle is the right child of the grandparent. If the parent of node is a right child of its parent, then the uncle is the left child of the grandparent.

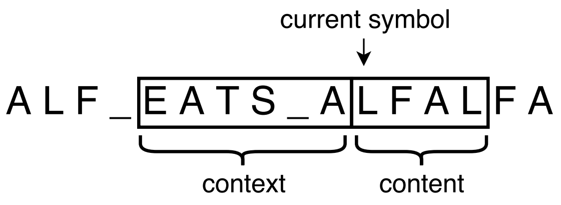

Figure 1.1: Sliding window used by ACB algorithm

1.2 Description

Compression algorithm ACB is context based compression method. This algorithm is based on properties of natural languages. In natural language is occurrence of next symbol dependent on context of text. The same words will probably have similar context.

ACB algorithm used adaptive dictionary. Both of encoder and decoder build the same dictionary dependent on actually processed character. Each item in the dictionary is represented by context-content pair. This algorithm used sliding window, show at figure 1.2, similar as algorithm LZ77 [13]. Left section of sliding window which contains processed text is called context. Right section of sliding window which contains raw text is called content. All items in the dictionary are lexicographically sorted by context. Characters in context are compared from right to left. Characters in content are lexicographically compared from left to right.

During every shift by one character of sliding window is created new context-content pair which is inserted to the dictionary.

Compression output and decompression input are triplets of numbers. These numbers represented difference between best content and best context, number of characters copy from the best content and value of next encoded character.

1.3 Compression

Compression starts with empty dictionary, in every algorithm iteration is inserted at least one context-content pair to the dictionary. Sliding windows starts with empty context. Context is all raw text.

Every algorithm iteration contains these four steps:

1. Find the best matching context for current context.

2. Find the best matching content for current content.

Algorithm 1.1 ACB compression

```
1: $i \leftarrow 0$
2: while $i < sizeOfFile$ do
3:     $c_x \leftarrow$ position of the best matching context for current context
4:     $c_n \leftarrow$ position of the best matching content for current content
5:     $l \leftarrow$ common length of the best matching content and current content
6:     $b \leftarrow$ first non matching symbol of current content
7:     OUTPUT $(c_n - c_x, l, b)$
8:     for $j \leftarrow 0$ to $l$ do
9:         DictionaryInsert$(i + j)$
10:    end for
11:    $i \leftarrow i + l + 1$
12: end while
```

3. Generate output triplet (d,l,b).

4. Insert new context-content pairs to the dictionary.

First step is finding the best matching context for current context in sliding window in all dictionary. Contexts are lexicographically compared character by character from right to left. When the best context is found, its position in sequence ordered by context is calculated.

Second step is finding the best matching content for current content in sliding window. Surrounding around the best context is scanning. Size of this surrounding is defined by number of bits using to encode difference between best content and best context. When the best content is found, its position in sequence ordered by context is calculated.

Third step is generating output triplet of numbers. First number represented difference of positions between best content and best context. Second number represented number of characters from left which are identically in the best content and current content in sliding window. Third number represented value of next character in sliding window content which did not copy from the best content.

Fourth step is shifting sliding window about number of character copy from the best content increasing by one. Every shift sliding window by one character generate context-content pair which is inserted to the dictionary.

Algorithm finish when current content in sliding window is empty string.

1.3.1 Compression example

For example I took the text **alf eats alfalfa**.

Step 1 First step starts on the beginning of the test. The dictionary is empty. The best context and the best content are not exist because the dictionary is empty. Difference between best content and best context will encode by 0. Number of copy characters from the best content is 0, next character is **a**. New context-content pair will insert to the dictionary.

|alf␣eats␣alfalfa

(0,0,a)

```
0  ϵ|a
```

Step 2 Second step starts shifting sliding window by one character. In the dictionary is one context-content pair. The best context and the best content are at index 0. Number of identical characters in the best content and current content in sliding window is 0. Because none characters from the best content are copy, it is not significant to know difference between best content and best context. This different will encode by 0. New context-content pair will insert to position 1 in the dictionary.

a|lf␣eats␣alfalfa

(0,0,l)

```
0   ϵ|al
1   a|l
```

Step 3 In third step the dictionary contains two context-content pairs. The best context found on position 1, the best content found on position 0. Difference between best content and best context is -1. Number of identical characters in the best content and current content in sliding window is 0. Because none characters from the best content are copy, difference will encode by 0. New context-content pair will insert to position 2 in the dictionary.

al|f␣eats␣alfalfa

(0,0,f)

```
0    ϵ|alf
1    a|lf
2   al|f
```

Step 4 Fourth step is the same as third step.

alf|␣eats␣alfalfa

(0,0,␣)

```
0     ϵ|alf␣
1     a|lf␣
2   alf|␣
3    al|f␣
```

Step 5 Fifth step is the same as previous step.

alf␣|eats␣alfalfa

(0,0,e)

```
0      ϵ|alf␣e
1   alf␣|e
2      a|lf␣e
3    alf|␣e
4     al|f␣e
```

Step 6 In sixth step the best context found on position 3, the best content found on position 0. Difference between best content and best context is -3. Number of identical characters in the best content and current content in sliding window is 1. This character **a** will skip, in output triplet will encode next character **t**. The dictionary will be contains two new context-content pairs on positions 4 and 3.

alf␣e|ats␣alfalfa

(-3,1,t)

```
0       ϵ|alf␣eat
1    alf␣|eat
2       a|lf␣eat
3  alf␣ea|t
4   alf␣e|at
5     alf|␣eat
6      al|f␣eat
```

Step 7 In seventh step the best context found on position 6, the best content found on position 3. None identical character is in the best content and current content in sliding window. To the dictionary will insert on context-content pair to position 7.

alf␣eat|s␣alfalfa

(0,0,s)

```
0         ϵ|alf␣eats
1      alf␣|eats
2         a|lf␣eats
3    alf␣ea|ts
4     alf␣e|ats
5       alf|␣eats
6        al|f␣eats
7   alf␣eat|s
```

Step 8 In eight step the best context found on position 7, the best content found on position 5. Encoded difference between best content and best context in output triplet is -2. Number of copy characters from the best content is 1. Next character in output triplet is **a**. The dictionary will be contains two new context-content pairs on positions 8 and 2.

alf␣eats|␣alfalfa

(-2,1,a)

```
0           ϵ|alf␣eats␣a
1        alf␣|eats␣a
2   alf␣eats␣|a
3           a|lf␣eats␣a
4      alf␣ea|ts␣a
5       alf␣e|ats␣a
6         alf|␣eats␣a
7          al|f␣eats␣a
8    alf␣eats|␣a
9     alf␣eat|s␣a
```

Step 9 In ninth step are the best context and the best content on position 3. Number of copy characters is 2, next character is **a**. The dictionary will be contains three new context-content pairs on positions 4, 10 and 8.

alf␣eats␣a|lfalfa

(0,2,a)

```
 0            ϵ|alf␣eats␣alfa
 1         alf␣|eats␣alfa
 2    alf␣eats␣|alfa
 3            a|lf␣eats␣alfa
 4   alf␣eats␣a|lfa
 5       alf␣ea|ts␣alfa
 6        alf␣e|ats␣alfa
 7          alf|␣eats␣alfa
 8 alf␣eats␣alf|a
 9           al|f␣eats␣alfa
10  alf␣eats␣al|fa
11     alf␣eats|␣alfa
12      alf␣eat|s␣alfa
```

Step 10 In tenth step the best context found on position 5, the best content found on position 4. Number of identically characters in the best content and current content in sliding window is 3. The last identical character is on the last position of raw text. In output triplet is encoded next character from raw text because number of copy bytes will be only 2. Last character from raw text **a** will encode as next character. None context-content pair will insert to the dictionary on the end of compression.

alf␣eats␣alfa|lfa

(-1,2,a)

Result Result is 10 triplets: **(0,0,a) (0,0,l) (0,0,f) (0,0,␣) (0,0,e) (-3,1,t) (0,0,s) (-2,1,a) (0,2,a) (-1,2,a)**.

1.4 Decompression

Decompression starts with empty dictionary, in every algorithm iteration is inserted at least one context-content pair to the dictionary. Left section of sliding window, called context, always contain processed text. Right section of sliding window, called content, is prepare for copy new characters form the dictionary and encoded information.

Every algorithm iteration contains these five steps:

1. Read input triplet (d,l,b).

2. Find the best matching context for current context.

3. Find the best matching content by difference d from input triple.

Algorithm 1.2 ACB decompression

```
1: $i \leftarrow 0$
2: while !EOF do
3:     $d \leftarrow$ read difference between best content and best context from triplet
4:     $l \leftarrow$ read number of copy symbols from triplet
5:     $b \leftarrow$ read next symbol from triplet
6:     $c_x \leftarrow$ position of the best matching context for current context
7:     $c_n \leftarrow d + c_x$
8:     for $j \leftarrow 0$ to $l - 1$ do
9:         OUTPUT Copy($c_n + j$)
10:        DictionaryInsert($i + j$)
11:    end for
12:    $i \leftarrow i + l$
13:    OUTPUT $b$
14:    DictionaryInsert($i$)
15:    $i \leftarrow i + 1$
16: end while
```

4. Copy l+1 characters to output.

5. Insert new context-content pairs to the dictionary.

First step is loading input triplet contains difference between best content and best context, number of copy bytes from the best content and value of next character from raw text.

Second step is finding the best matching context for current context in sliding window. This process is same as process in compression.

Third step is finding the best matching content. Position of the best content is calculated from position of the best context and difference between best content and best context loaded from input triplet.

Forth step is copying l characters form the best content to output. Value of character b from input triplet is copy to output text too.

Fifth step is shifting sliding window about number of character copy from the best content increasing by one. Every shift sliding window by one character generate context-content pair which is inserted to the dictionary.

Algorithm finish when none triplet is on input.

1.4.1 Decompression example

For example I took this text encoded text: **(0,0,a) (0,0,l) (0,0,f) (0,0,␣) (0,0,e) (-3,1,t) (0,0,s) (-2,1,a) (0,2,a) (-1,2,a)**.

Step 1 First step starts with empty output text. The dictionary is empty. Number of copy characters from the best content is zero, the best context and the best content are insignificant. Next character **a** from input triplet will copy to output text. New context-content pair will insert to the dictionary.

(0,0,a)

a|

```
0  ϵ|a
```

Step 2 In second step is the same situation as in first step. Next character **l** from input triplet will copy to output text. New context-content pair will insert to position 1 in the dictionary.

(0,0,l)

al|

```
0   ϵ|al
1   a|l
```

Step 3 In third step is the same situation as in previous step. Next character **f** from input triplet will copy to output text. New context-content pair will insert to position 2 in the dictionary.

(0,0,f)

alf|

```
0    ϵ|alf
1    a|lf
2   al|f
```

Step 4 In fourth step is the same situation as in previous step. Next character ␣ from input triplet will copy to output text. New context-content pair will insert to position 2 in the dictionary.

(0,0,␣)

alf␣|

```
0     ϵ|alf␣
1     a|lf␣
2   alf|␣
3    al|f␣
```

Step 5 In fifth step is the same situation as in previous step. Next character **e** from input triplet will copy to output text. New context-content pair will insert to position 1 in the dictionary.

(0,0,e)

alf␣e|

```
0      ϵ|alf␣e
1   alf␣|e
2      a|lf␣e
3    alf|␣e
4     al|f␣e
```

Step 6 In sixth step is copying 1 character from the best content to output text. Find the best matching context with current context in sliding window. The best context found at position 3. Difference between best content and best context from input triplet is -3. Calculating position of the best content is 0. First character of this content is **a**. This character will copy to output text together with next character from input triplet **t**. The dictionary will be contains two new context-content pairs on positions 4 and 3.

(-3,1,t)

alf␣eat|

```
0        ϵ|alf␣eat
1     alf␣|eat
2        a|lf␣eat
3   alf␣ea|t
4    alf␣e|at
5      alf|␣eat
6       al|f␣eat
```

Step 7 In seventh step is copy none characters from the best content. Next character **s** from input triplet will copy to output text. New context-content pair will insert to position 7 in the dictionary.

(0,0,s)

alf␣eats|

```
0         ϵ|alf␣eats
1      alf␣|eats
2         a|lf␣eats
3    alf␣ea|ts
4     alf␣e|ats
5       alf|␣eats
6        al|f␣eats
7   alf␣eat|s
```

Step 8 In eight step is copying 1 character from the best content to output text. The best context found at position 7. Difference between best content and best context from input triplet is -2. Calculating position of the best content is 5. First character of this content is ␣. This character will copy to output text together with next character from input triplet **a**. The dictionary will be contains two new context-content pairs on positions 8 and 2.

(-2,1,a)

alf␣eats␣a|

```
0           ϵ|alf␣eats␣a
1        alf␣|eats␣a
2   alf␣eats␣|a
3           a|lf␣eats␣a
4      alf␣ea|ts␣a
5       alf␣e|ats␣a
6         alf|␣eats␣a
7          al|f␣eats␣a
8    alf␣eats|␣a
9     alf␣eat|s␣a
```

Step 9 In ninth step are copying 2 character from the best content to output text. The best context found at position 7. Best content is at the same position. First two characters of this content are **lf**. These characters will copy to output text together with next character from input triplet **a**. The dictionary will be contains three new context-content pairs on positions 4, 10 and 8.

(0,2,a)

alf␣eats␣alfa|

```
 0             ϵ|alf␣eats␣alfa
 1          alf␣|eats␣alfa
 2     alf␣eats␣|alfa
 3             a|lf␣eats␣alfa
 4    alf␣eats␣a|lfa
 5        alf␣ea|ts␣alfa
 6         alf␣e|ats␣alfa
 7           alf|␣eats␣alfa
 8  alf␣eats␣alf|a
 9            al|f␣eats␣alfa
10   alf␣eats␣al|fa
11      alf␣eats|␣alfa
12       alf␣eat|s␣alfa
```

Step 10 In last step are copying 2 character from the best content to output text too. The best context found at position 5. The best content is at position 4. First two characters of this content are **lf**. These characters will copy to output text together with next character from input triplet **a**. None context-content pairs will insert to the dictionary because decompression is at the end.

(-1,2,a)

Result Result is the text **alf eats alfalfa**.

Chapter 2

Previous implementations

In this chapter are describe previous implementations of ACB algorithm. To compare preview implementations I relied on master thesis Efficient implementation of ACB compression algorithm for ExCom library [11] published by Michal Valach in 2011. This thesis follow thesis published by Michal Valach, details and implementations usage are describe in the master thesis [11].

2.1 George Buyanovsky

George Buyanovsky is creator of original ACB algorithm. Algorithm was present in 1994 only in Russian. Publication also included implementation which is not distributed under free licence but only as MS-DOS executable.

The source code by George Buyanovsky has more variants, one of them he described in Description of acb published in comp.compression [3] in 1996. This publication describe some features of algorithm. The most important feature is using arithmetic coding before write data to output stream.

Together with the released application a README file are known this capabilities: the program can compress more files together, maximal memory consumption during compression and decompression is 16MB, maximal size of input file is 64MB.

2.2 Martin Děcký

Next implementation of ACB algorithm published Martin Děcký in 2006 as a proof of concept project in class at Faculty of Mathematics and Physics at Charles University in Prague [5]. Source code is written in C++ and distributed under the GNU GPL license.

Implementation use three variants of triplets. First variant encoding difference, number of symbols and new symbol. Second variant encoding difference and number of symbols. Third variant encoding only new byte. Each triplet is extend by two flags which define one of this triplet variant.

2.3 Filip Šimek

Filip Šimek is creator of the first ACB implementation distributed with ExCom library [10] as part of his master thesis Data compression library [9].

The dictionary is implemented as array of pointers. The dictionary is deleted every time if the new block of data is read, dictionary cannot be used for compression next block of data. Triplets are not encoded yet. Numbers are only writing to output stream, range is limit by number of bits for each number.

2.4 Lukáš Cerman

Lukáš Cerman is a creator of another implementation in his bachelor thesis Acb compression algorithm [4] in 2003. It was implemented in C in Microsoft Visual Studio.

The dictionary is partially store as hash table, partially as simple array. For triplets encoding use Huffman coding [6]. There are three variants of triplets. The implementation also contains basic support for the context files.

2.5 Michal Valach

Michal Valach is creator of the actual implementation of ACB algorithm in ExCom compression library [10]. This implementation is part of them master thesis Efficient implementation of ACB compression algorithm for ExCom library [11].

In this implementation the dictionary is represent by B+ tree. For output data using two variants of triplet. If none symbol is copy, encode 0 and new symbol. Otherwise encode difference, number of copy symbols and new symbol.

For triplets encoding use Huffman coding or arithmetic coding. Both variants using three different alphabets, each alphabet for one part of triplet. This feature decrease compression ration.

2.6 Implementations comparison

Compare compression ration this algorithms 2.1 is from master thesis Efficient implementation of ACB compression algorithm for ExCom library [11] by Michal Valach.

Table 2.1: Compression ratios of previous implmentations ACB algorithm for Calgary corpus

	Compression ratio [-]				
File	Buyanovsky	Cerman	Děcký	Šimek	Valach
bib	0.241	0.307	1.879	0.566	0.304
book1	0.290	0.374	2.159	0.735	0.373
book2	0.242	0.316	-	0.618	0.324
geo	0.570	0.658	3.946	1.004	0.650
news	0.290	0.359	-	0.681	0.359
obj1	0.438	0.522	3.436	0.804	0.510
obj2	0.275	-	-	0.546	0.357
paper1	0.293	0.362	2.413	0.618	0.357
paper2	0.293	0.364	2.401	0.648	0.360
pic	0.093	0.113	-	-	0.114
progc	0.291	0.358	2.414	0.599	0.352
progl	0.188	0.243	1.559	0.403	0.242
progp	0.188	0.238	1.591	0.407	0.237
transdfgg	0.161	0.206	1.305	0.405	0.206

Experimental measure use Calgary corpus. The compression ratios for some files are not listed because the compression was unable to end in a reasonable time.

The implementation by George Buyanovsky is the best for all files. The difference between implementation of George Buyanovsky and implementation of Michal Valach is approximately 5%. The implementation by Lukáš Cerman gives similar results as the implementation by Michal Valach. The other two implementations is not achieved such high quality results.

CHAPTER 3

Analysis

This chapter contains analysis data structures for effective implementation compression algorithm ACB, analysis effective storing data in memory and their encoding before store. This implementation use adaptive arithmetic coding. Finally this chapter describe new data structure for the best compression ration of ACB algorithm.

3.1 Data structures

Data structure for ACB algorithm is selected for fast context searching. Chapter 1 describe requirement fast searching. The best context is search during encoding every symbol.

During encoding every symbol is searching the best matching content for current content in surrounding of the best context. The surrounding of the best context is limited by number of bits using to encode different between best content and best context. Data structure can be selected with respect to these properties.

Using data structure can have fast insert operation of new item. After each encoding symbol have to be insert new context-content pair to the dictionary. This operation is describe in chapter 1.

These requirements most convenient search trees. Next parts of this chapter binary search tree and its balance variant red-black tree. Finally this chapter describe extension this search trees by statistical methods rank & select which compute item position in search tree and search item on concrete position in search tree.

3.1.1 Binary search tree

In this definition of binary search tree are all inserted item unique. Otherwise can be change inequalities.

Binary search tree is build from nodes. Each node can have maximal two children. In each leaf and inner node is save key. All keys in left sub-tree are lower than key in current node. All keys in right sub-tree are greater then key in current node.

Binary search thee enable three basic operations: insert new key to binary search tree, select node define by concrete key and delete node define by concrete key. All these operations have in average case asymptotic complexity O(log n), in the worst case have asymptotic complexity O(n), n is number of nodes in binary search tree. Binary search tree is not balanced, the worst case of binary search tree is linear list with n items.

Dictionary of ACB algorithm use only two operations: insert and delete. Operation delete is not use because dictionary in ACB algorithm by definition is not allow delete item by item. Dictionary can be delete only all in one step and replace by new empty dictionary.

Insert item to binary search tree Insert new item to binary search tree is realize by add new leaf to binary search tree. In the beginning it can search location to add new leaf. All properties of binary search tree have to be preserved. All keys in left sub-tree are lower than key in current node. All keys in right sub-tree are greater then key in current node. Each node can have maximal two children.

During search location to add new leaf is binary search tree traverse from root to leaf. In each node is decide if new key is lower or greater than key in current node. If the new key is lower than key in current node than recursive continue to left sub-tree. Otherwise recursive continue to right sub-tree. This action is repeat until sub-tree is not exist. In this case is insert new leaf instead of this sub-tree. This case terminate insertion process, all properties are true.

Search item in binary search tree Operation to search item in binary search tree is similar to operation search location to add new leaf. This operation is base on properties binary search tree.

During search operation is binary search tree traverse from root to leaf. In each node is decide if searched key is lower, greater or equal to key in current node. If the searched key is equal to the current key, search node found. If the searched key is lower than the current key than recursive continue to left sub-tree. If the searched key is greater than the current key than recursive continue to right sub-tree. If the sub-tree is not exist, search key is not in binary search tree.

3.1.2 Red-Black tree

Reference for this section is lecture Balanced BSTs - AVL and RB Trees [8].

Red-black tree is binary search tree extend by balance. Tree balancing guarantee maximal height of search tree. Operations insert search and delete can be compute faster.

Red-black tree is balance to i.e. black depth. Black depth is defined as the number of black nodes on the path from the root to leaf. As a contrary to AVL tree balance to sub-tree height is red-black tree less balance. Red-black tree need fewer rotations, its better for often insert operations.

Implemented ACB algorithm use red-black tree because encoding each symbol insert new context-content pair do the dictionary. Insert operations to search tree are very often.

Because red-black tree is balance, insert and search operations have complexity $\beta logn, 1 \leq \beta < 2$, n is number of items in red-black tree.

Red-black tree properties Red-black tree meet all the requirements binary search tree. Properties of red-black tree are extend by this properties:

- Each node is either red or black.
- The root of the tree is black.
- Each leaf is black NULL.
- A red node has both its children black.
- All leaves have the same black depth.

Rotations Red-black tree using four variant of rotation to guarantee balance the search tree. These rotations are: left rotation, right rotation, left-right rotation and right-left rotation.

Left rotation During left rotation node **x** is replace by its right sub-tree, node **x** is its left sub-tree. Left sub-tree of node **x.right** after rotation is right sub-tree node **x**.

Right rotation During right rotation node **x** is replace by its left sub-tree, node **x** is its right sub-tree. Right sub-tree of node **x.left** after rotation is left sub-tree node **x**.

Left-right rotation Left-right rotation is compose of left rotation around node **x.left** and right rotation around node **x**.

Right-left rotation Right-left rotation is compose of right rotation around node **x.right** and left rotation around node **x**.

Insert item to red-black tree Insert item do red-black tree is identically to insert item to binary search tree. Original black NULL node is replace by new node. This new node is coloured to red. Finally it is necessary to maintain properties of red-black tree.

There are four case to solve the violation red-black tree properties.

Coloured node to black In the case if parent of current node is not exist than only coloured current node to black. All red-black properties are true.

Coloured parent and uncle to black, grandparent to red In the case if parent and uncle are red than coloured parent and uncle to black, coloured grandparent to red. Grandparent is actually red, its can violate red-black properties. This problem solve recursion property check to grandparent.

Right or left rotation around grandparent In the case if parent is red and uncle is black or not exist then separate two variants.

The current node is left child of its parent, parent is left child of grandparent. In this case use right rotation around grandparent.

The current node is right child of its parent, parent is right child of grandparent. In this case use left rotation around grandparent.

Right-left or left-right rotation around grandparent In the case if parent is red and uncle is black or not exist then separate two variants.

The current node is right child of its parent, parent is left child of grandparent. In this case use right-left rotation around grandparent.

The current node is left child of its parent, parent is right child of grandparent. In this case use left-right rotation around grandparent.

Search item in red-black tree Search item in red-black tree is identically to search item in binary search tree. Advantage of red-black tree is asymptotic complexity which is in the worst case O(log n) in contrast O(n) in binary search tree.

3.1.3 Order statistical tree

Order statistical tree is extension binary search tree or balance red-black tree. Order statistical tree contain two new statistical function rank and select.

Asymptotic complexity for both rank and select operations is O(log n) for red-black tree, n is number of items in search tree.

Each node in binary search tree or red-black tree is extend by integer variable which contains size of sub-tree. This information is necessary for calculate rank and select operation in logarithmic time. Sub-tree size in update each insert item to sub-tree.

Algorithm ACB use functions rank and select for calculate position of the best context and the best content in sequence ordered by contexts. These positions are used for encoding difference between best content and best context.

Rank Function rank find position of item in order sequence.

Algorithm 3.3 Rank

```
Require: $node$
Ensure: $ranking$
1: $ranking \leftarrow 0$
2: while $node$ != NULL do
3:     if $node$ is right child its parent then
4:         $ranking = ranking$ + sub-tree size of parent left child + 1
5:     end if
6:     $node \leftarrow$ parent of $node$
7: end while
8: return $ranking$
```

Select Function find item on n^{th} position in order sequence.

Algorithm 3.4 Select

```
Require: $searchRank$
Ensure: $node$
1: $node \leftarrow$ root of search tree
2: $actualRank \leftarrow$ left sub-tree size
3: while $searchRank$ != $actualRank$ do
4:     if $searchRank < actualRank$ then
5:         $node \leftarrow$ left child of $node$
6:     else if
7:          then $node \leftarrow$ right child of $node$
8:         $searchRank \leftarrow searchRank - actualRank - 1$
9:     end if
10:    $actualRank \leftarrow$ left sub-tree size
11: end while
12: return $node$
```

3.2 Triplets

Algorithm ACB using three numbers for store information about compress data. First number represented difference of positions between best content and best context. Second number represented number of characters from left which are identically in the best content and current content. Third number represented value of next character in current content which did not copy from the best content.

This is three numbers used by George Buyanovsky in original ACB algorithm to store all required informations for data decompression.

There are more variants to effective store this informations to minimization memory requirements.

Encoding three numbers (d, l, b)

This variant using always all three numbers. Compression ratio is depend on number of bits use to encode particular numbers. The fewer bits are used, the less memory are required. On the other hand, fewer bits can retain smaller amount of information.

Character **d** represented difference between best content and best context. The best context is search in all dictionary. The best content is search only in limited surrounding of the best context. This surrounding is limited by number of bits to encoding this different. When its use a less bits may not be found as good context and consequently will not be copy as many symbols as it could be when using more bits.

Character **l** represented number of symbols copy from the best content. The more bits will be used to store this information, the more symbols can be copy.

Character **b** represented value of next symbol which did not copy from current content. For this information is always used 8 bits.

Encoding three numbers or two numbers (l, d, b) or (l, b)

This variant is different by previous variant only in special cases when can be used only two numbers. In this variant decoder have to decided if process three numbers or two numbers.

Three numbers is using in case if number of copy bytes from the best content is at least one. In this case is procedure identical as encoding three numbers whenever.

Two numbers is using in case if none symbol is copy from the best content. In this case information about difference between best content and best context is irrelevant. The best content will never be used.

First information which is encoding is number of copy symbols. This number decide if actual token contain three number or two numbers.

Encoding two numbers or one number (d, b) or (b)

This variant encoding two numbers or one number. Always is used 1 bit flag to determination how numbers can be decode.

Two numbers is using in case if number of copy bytes from the best content is at least one. In this case first number represented difference between best context and best context, second number represented value of next symbol which did not copy from current content.

One number represented value of next symbol which did not copy from current content is using in case if none symbol is copy from the best content.

Number of copy symbols from the best content have to compute both encoder and decoder. This variant is used for experiment describe in chapter 3.5.2.

3.3 Arithmetic coding

Output of compression algorithm ACB is sequence of numbers. This numbers have to be store to memory. Storing raw numbers to memory is inefficient, numbers can be compress by other algorithm. Arithmetic coding is good algorithm to compress numbers.

Details about arithmetic coding are published in Arithmetic coding for data compression [12].

For store raw numbers is use always same number of bits, arithmetic coding use different number of bits for different numbers. For numbers that occur more often, it is used a smaller number of bits than the number that occur less frequently.

Arithmetic coding encode input data to real number in interval $[0;1)$. With increasing size of input data is needed to use a real number with greater precision because differences is decreases.

During compression a decompression is using the dictionary with different probability of symbols. This probabilities are using to calculate cumulative probability of symbols and range for each symbol.

In the beginning of compression is used interval $[0;1)$. During the compression interval is reduce. In each iteration one symbol with its interval in the dictionary is selected. Lower and upper bounds of encoded interval are recalculated with using symbol interval. After last algorithm iteration is interval on output. Result is any number for this interval.

Encoded date have to be complete information about end of decompression. This information can be represented as special symbol at the end of input data. It can be also added to result number of compressed numbers.

Decompression is similar procedure as a compression.

3.3.1 Static arithmetic coding

Static variant of arithmetic coding is based on previously known probability distribution of symbols. Dictionary is build from this distribution before compression, during compression dictionary is not update.

The same dictionary have to known both encoder and decoder.

3.3.2 Adaptive arithmetic coding

Adaptive variant of arithmetic coding is based on dictionary update depending on input data. It is not important to know probability distribution of symbols before compression and decompression. In each iteration new symbol is insert to the dictionary, symbols probabilities are recalculated. Next iteration used different probabilities than current iteration.

Dictionary is build encoder as same as decoder. Decoder insert to the dictionary actually decoded symbol.

Adaptive model is suitable to use on unknown data. It is also numbers on output of ACB algorithm.

3.4 Dictionary mode

Algorithm ACB inserting new context-content pair to the dictionary after compression each symbol. Size of the dictionary directly proportional to size of input data.

3.4.1 Stream mode

Original algorithm published by George Buyanovsky using for compression all input data only one dictionary. Advantage of one dictionary for all compression is occurrence all contexts in the dictionary, algorithm will find the best of them. Similar input data in all input file use this fact but variable input data in input file did not use this fact a lot. This data will use recently inserted content-context pairs most frequently, advantage will not apply.

Huge dictionary is very memory consumption. Inserting and searching in huge dictionary is more time consuming than inserting and searching in small dictionary because these operations are logarithmically dependent on size of dictionary.

3.4.2 Block mode

Next variant is limited size of dictionary. Advantage of this variant is limited memory need to store the dictionary. Disadvantage of this variant is limited number of contexts in the dictionary. In the dictionary can not be store the best context. This is limitation only for similar input data in all input file. Variable input data in input file is not limited because this data will use recently inserted content-context pairs most frequently. This pairs will be always in the dictionary.

There is more variant to solve maximal size of the dictionary.

Use one dictionary forever In this case if the dictionary reaches the maximal size than new context-content pairs will not be inserting. Compression and decompression will be using this dictionary forever.

There is a risk a big decreasing compressing ratio because new context-content pairs will not be inserting.

Immediately dictionary delete, building new dictionary In this case if the dictionary reaches the maximal size than will delete immediately. New dictionary will be building. This dictionary will be using for compression and decompression.

New context-content pair will always insert to the dictionary. A big decreasing compressing ratio will not be there. After delete old dictionary new dictionary is empty. None context and content will not find in the dictionary.

Monitoring compression radio, dictionary delete when compressing ratio decrease This variant is combination of previews to variants. New context-content pairs is insert to the dictionary while dictionary has not maximal size. After than starts compression ration monitoring during compression next symbols. When compression ration decrease below certain threshold than the dictionary will delete. New dictionary will build.

While input data is similar than current context and content in similar to contexts and contents in the dictionary. Compression ratio is invariable. When input data is variable than context and content is not found. Compression ration is decreasing, old dictionary is delete and new dictionary is build. New context-content pairs is inserting to the dictionary.

Main goal of this variant is smaller number of dictionary deletions during similar input data. New dictionary is building after change of input data.

3.5 New improvements

3.5.1 Subtracting LCP from number of copy symbols

Original algorithm ACB published by George Buyanovsky in each iteration of compression search the best context in all dictionary and the best content in limited surrounding of the best context. This limit is depend on number of bits use to encode difference between best content and best context. In each iteration of decompression search the best context. Position of the best content is calculate form distance between best content and best context loading form input. Finally some bits from the best content are copy to output.

One variant to improvement compression ratio is in each iteration of compression search the best content and the second best content. From the best content and the second best content calculate LCP (longest common prefix). To output is now encode number of copy symbols from the best content minus

LCP instead of only number of copy symbols from the best content. Amount of this information is limited by number of bits use for encoding this information. This variant allow copy more symbols from the best context. This number is increasing by value of LCP.

For encoding output is using three or two numbers, this is describe in chapter 3.2. Its require to number of copy symbols have to be grater than zero if some symbols are copy. This is solution this problem. The best content is content which LCP with current content is the biggest. If more contents have same LCP with current content, the the best content is content on the smallest position. The second best content is content which LCP with current content is the biggest and this content is lexicographically smaller than the best content. Because the best content is lexicographically smaller content with the best LCP to current content, LCP of the best content and the second best content is always at least one smaller than number of copy symbols from the best content. Number of copy symbols form the best content minus LCP of the best content and the second best content is always grater than zero.

In each iteration of decompression is search the best context. Position of the best content is calculated from difference between best content and best context load from input. The second best content is search identical procedure as search the second best content during compression. LCP of the best content and the second best content is calculated. If number of copy symbols form the best content is grater than 0 then number of copy symbols is increasing by value of LCP.

This variant allow copy more symbols from the best content, numbers of bits to encode this information is constant.

3.5.2 Number of copy symbols is equal to value of LCP

Original algorithm ACB published by George Buyanovsky using three numbers to store all needed informations: difference between best content and best context, number of copy symbols from the best content and value of next symbol from current content. This information need fixed memory size.

One variant to improvement compression ratio is store less informations. Remaining informations can by calculated during both compression and decompression.

This variant searching the best content and the second best content identically as searching the best content and the second best content in previous variant. LCP of the best content and the second best content is computing. LCP is always at least one lower than number of symbols which can be copy form the best content.

Because LCP is always lower than number of symbols which can be copy form the best content then its can be copy from the best content only number of symbols equal to value of LCP. Nothing information will lose.

Copy number of symbols equal to value of LCP allow did not encode information about number of copy symbols from the best content. In each iteration of both compression and decompression LCP will be calculating and number of symbols equal to value of LCP will copy from the best content to output. Disadvantage this variant is copy at least one less number of copy symbols than number of copy symbols in original variant of algorithm.

For output is using two number or one number. It is describe in chapter 3.2.

3.5.3 Index shifting

Next variant to improvement compression ratio is use index shifting for greater similarity between best content and current content.

In each iteration is search the best context and the best content. Different between best content and best context is use in decompression to find the best content.

This information can be used to index shifting in next iteration too and search the best content in shift surrounding. Now is encoding to output $8-3 = 5$ if the best context is at position 3 and the best content is at position 8. In next iteration now is encoding to output $9-5 = 4$ if the best context is 2 and the best content is 9. Information from last iteration that difference between best content and best context is 7 can be used to approach values in actual iteration. It will be encoding $9-7-5 = -3$, information to next iteration is $9-7 = 2$.

In next iteration the best content in not search in surrounding of the best context but in surrounding of the best context shift by difference between best content and best context from last iteration. It can be grater chance to find better content here if there was the best content in last iteration.

This variant is base on copy more symbols from the best content. This variant can be apply to all previous variants of algorithm ACB.

Chapter 4

Implementation

Compression library ExCom [10] is written in C++ language. This module for ExCom library is written in C++ too.

For module development was using OS Linux and C++ IDE QtCreator. This IDE chose for syntax highlighting and intelligent code completion.

4.1 Module for ExCom library

Compression library ExCom [10] contain module for each algorithm. This library is easy extensible by new module contain new algorithm. New module extends class `CompModule` which integrate new module to library. This class manage OI operations and control algorithm properties by input parameters.

Each new module have to implement these functions: `connectIOModule`, `setOperation`, `setParameter`, `getValue`, `checkConnection` and `run`.

4.1.1 connectIOModule

This function manage connection to input and output stream. Class `IOReader` read data form input stream, class `IOWriter` write data to output stream.

After initialize these class is using only this classes to read input data and write output data.

IOReader The class `IOReader` allow read data from input stream byte by byte using function `int readByte()`. Data form input stream can be read by bites using function `int readNBits(unsigned int n, unsigned long *data)`. This function can read 1 to 32 bits together. Data from input stream can by read by blocks of bytes using function `int readBlock(unsigned char *dest, unsigned long len)`.

IOWriter The class `IOWriter` allow write data to output stream byte by byte using function `int writeByte(unsigned char value)`. Data to output stream can be write by bites using function `int writeNBits(unsigned int n, unsigned long value)`. This function can write 1 to 32 bits together. Data to output stream can be write by blocks of bytes using function `unsigned int writeBlock(const unsigned char *data, unsigned long len)`.

4.1.2 setOperation

This function only set flag by value of parameter set by user when program starts if compression or decompression operation will be run.

4.1.3 setParameter

Algorithms implemented in ExCom library are managed by parameters set by user when program starts.

Parameters for each algorithm are defined in header file `excom.h`. These parameters can user set when program starts and manage properties of algorithm.

This function parse input parameters and assign that to class variables. There are validate values of input parameters. If value is not correct than function return error code `EXCOM_ERR_PARAM`.

This algorithm contain five parameters to manage algorithm properties. When program starts parameters can be set by this characters:

- maximal dictionary size - **s**,
- maximal context length - **x**,
- maximal content length - **n**,
- number of bits to encode difference between content and context - **d**,
- number of bits to encode number of symbols copy from best content - **c**.

4.1.4 getValue

This function return value of variable assign to concrete parameter.

4.1.5 checkConnection

This function check correct connection to input and output stream. If check failed than function return error code `EXCOM_ERR_CHAIN`.

4.1.6 run

This function starts computing compression or decompression. This function is call after assign input and output stream and after set all parameters.

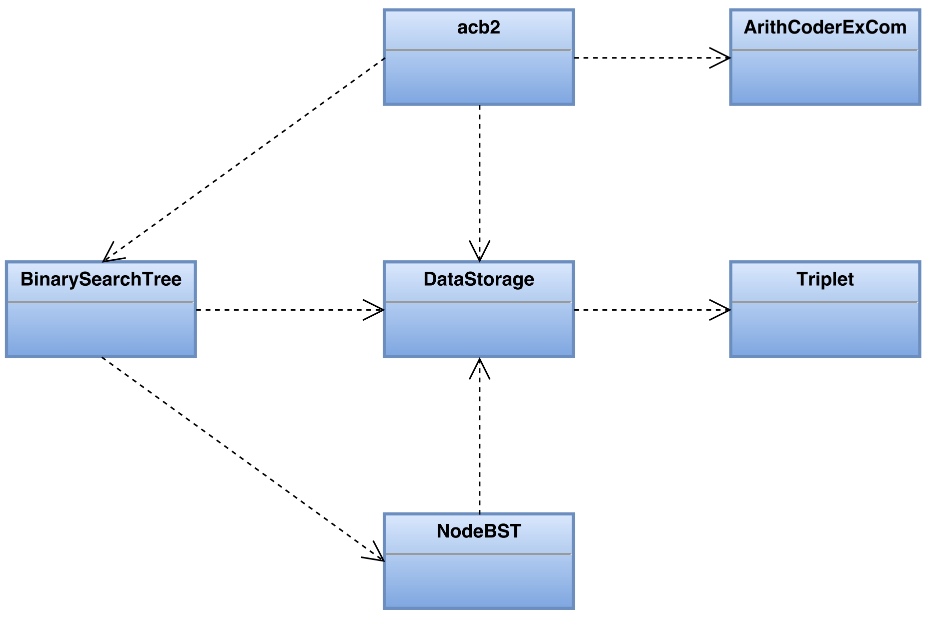

Figure 4.1: Class diagram of ACB2 module for ExCom library

4.2 Object Model

Design of ACB module is based on principles object oriented programming. The module consists fix classes. Each class is responsible for specific functionality. Class diagram is shown in figure 4.2.

acb2 This class contains basic logic of compression algorithm ACB. There is dictionary (`BinarySearchTree`), storage of data (`DataStorage`) and arithmetic coder (`ArithCoderExCom`). This class manage compression and decompression.

DataStorage In this class is store raw text and informations (triplets) for decompression. Triplets will compress by arithmetic coder yet. This class comparing contexts and contents, computing LCP of contents.

Triplet This class is data structure for store informations required to decompression.

BinarySearchTree This class is dictionary for ACB algorithm. During compression and decompression is building binary search tree to store context-content pairs. This class searching the best context, the best content and the second best content. Each node of binary search tree is class `NodeBST`.

NodeBST This class represent single node of binary search tree. In each node is store index to raw data which is situated in class `DataStorage`.

ArithCoderExCom This class is interface for arithmetic coder. Arithmetic coder is module of ExCom library. This class offer simple using this arithmetic coder.

4.3 Main compression and decompression logic

Main compression and decompression logic is contained in source file `acb2.cpp` and header file `acb2.hpp`.

These files contain class `CompAcb2`. Class `CompAcb2` extends class `CompModule`. This class have to be extend by each module integrated to compress library ExCom. This class contains methods managing input and output data, parsing parameters from user and start compression and decompression procedure.

Class `CompAcb2` contains private variables of type pointer to class `IOReader` a `IOWriter`. This classes manage read data from input stream and write data to output stream. This class contains also five integer variables. In this variables are store value of parameters affected properties compression algorithm.

Algorithm parameters Compression algorithm have to run compression and decompression procedure with same parameters. When starts compression, user can set any of parameter to different value. If parameter is not set than default value will use. Default values are chosen for optimal compression ratio for the expected data.

When starts compression, parameters set by user are discard. Parameters decompression have to be same as parameters to compression. This value have to be store during compression and load before decompression. Value of parameters are store to the beginning of compressed file using function `void writeHeader()` and load from this file using function `void readHeader()`. Number of bits using to store this informations is depend on range of this values. Maximal dictionary size use 4B, maximal context length and maximal content length use 2B, maximal different between best context and best content use 1B, maximal number of copy symbols from the best content use 1B too.

Compression and decompression This module contains 3 variants of ACB algorithm. These variants are detail describe in chapter 3.5. First variant is original version of ACB algorithm published by George Buyanovsky. Second variant (mark V1) contain improvement Subtracting LCP from number of copy symbols, third variant (mark V2) contain improvement Number of copy symbols is equal to value of LCP. Functions managing all compression logic are `int compress()`, `int compressV1()` and `int compressV2()`. Functions for decompression are `int decompress()`, `int decompressV1()` and `int decompressV2()`.

In the beginning of compression parameters of algorithm are writing to output stream by function `writeHeader()`. After that is creating store of data `DataStorage` and dictionary `BinarySearchTree`. Input file is reading byte by byte and storing to `DataStorage`. After that input data is transformed to triplets. This is main part of all compression. Finally triplets are compress by arithmetic coder.

In the beginning of decompression parameters of algorithm are reading from input stream by function `readHeader()`. After that is creating store of data `DataStorage` and dictionary `BinarySearchTree`. Arithmetic coder decompress triplets from input stream. After that triplets are transformed to raw data. This is main part of all decompression. Finally raw data are write to output stream byte by byte.

Transform raw data to triplets For transformation input data to triplets is using function `void bytesToTriplets(BinarySearchTree *dictionary, DataStorage *dataStorage)` and other variants of this function with mark V1 and V2. These functions are slightly different depending on the implemented improvements.

Basic version of this function processing input data byte by byte. In each iteration is searching the best context and the best content in the dictionary. Positions of the best context and the best content are compute by function `rank` returning position in ordered sequence. If the best content exist than number of copy bytes from the best content is calculated. If at least one byte is copy than three numbers are copy: different between best content and best context, number of copy bytes from the best content and value of next byte from current content. Otherwise different between best content and best context is encode as 0. None information is lose, for arithmetic coder is zero better then other numbers.

Case of variant V1 (Subtracting LCP from number of copy symbols) is similar to previous variant. In each iteration is searching the second best content yet. If the second best content exist than LCP of the best content and the second best content is calculate. If at least one byte is copy from the best content than encoding number of copy bytes from the best content minus LCP of the best content and the second best content instead of number of copy bytes from the best content.

Case of variant V2 (Number of copy symbols is equal to value of LCP) is similar to variant V1. In each iteration is calculated LCP of the best content and second the best content too. Number of copy bytes from the best content is equal to value of LCP. Information about number of copy bytes from the best content is not encoded now. Instead of is set flag means copy at least one byte from the best content.

Transform triplets to raw data For transformation triplets to output is using function `void tripletsToBytes(BinarySearchTree *dictionary, DataStorage *dataStorage)` and other variants of this function with mark V1 and V2. These functions are slightly different depending on the implemented improvements.

Basic version of this function processing triplets one by one. In each iteration difference between best content and best context, number of copy bytes prom the best content and value of next byte from current content is loading form current triplet. If number of copy bytes from the best content is grater then one than the best context is searching. After that position of the best content is computing. From the best content is copying bytes to output. Number of copy bytes is defined by number load from current triplet. Each copy byte is insert to the dictionary immediately. Finally is copying next byte loaded form current triplet to output. This byte is insert to the dictionary too.

Case of variant V1 (Subtracting LCP from number of copy symbols) is similar to previous variant. In each iteration is searching the second best content yet. If the second best content exist than LCP of the best content and the second best content is calculate. If at least one byte is copy from the best content than number of copy bytes is increasing by LCP value.

If in case of variant V2 (Number of copy symbols is equal to value of LCP) is set flag means copy at least one byte from the best content than searching the best content and the second best content. LCP of the best content and the second best content is calculate. Number of copy bytes from the best content to output is equal to value of LCP.

Arithmetic coding Function `void tripletsToArith(DataStorage *dataStorage)` is used to encoding triplets by arithmetic coding. This function is used to original variant of algorithm and variant V1 (Subtracting LCP from number of copy symbols).

In the beginning the encoder `ArithEncoder` is created. In each iteration by triplets is encoding next byte from current content and number of copy bytes. If number of copy bytes if grater then zero than different between content and context is encoding too. Different between content and context is increasing by half value of maximal range. This number never be negative.

In case of variant V2 (Number of copy symbols is equal to value of LCP) is number of copy bytes replace by 1 bit flag means copy at least one byte. This flag is important for decoder to distinguish meaning next number.

Function `void arithToTriplets(DataStorage *dataStorage)` is used to decoding triplets by arithmetic coding. This function is used to original variant of algorithm and variant V1 (Subtracting LCP from number of copy symbols).

In the beginning the decoder `ArithDecoder` is created. While end of data in not found than decode next byte from current content and number of copy bytes. If number of copy bytes if grater then zero than different between content and context is decoding too. Different between content and context is decreasing by half value of maximal range to give original value.

In case of variant V2 (Number of copy symbols is equal to value of LCP) is decoding 1 bit flag instead of number of copy bytes. This flag is important to distinguish meaning next number.

4.4 Balanced binary search tree with Rank & Select

Dictionary consists of balanced binary tree with statistical methods rank and select is contained in source files `BinarySearchTree.cpp` a `NodeBST.cpp` and in header files `BinarySearchTree.hpp` a `NodeBST.hpp`.

Files `BianarySearchTree.cpp` and `BinarySearchTree.hpp` contains class `BinarySearchTree`. This class is base of dictionary. This class contains several private variables: pointer to binary search tree root type of `NodeBST`, pointer to storage of data `DataStorage`, maximal dictionary size and maximal different between best content and best context.

Files `NodeBST.cpp` and `NodeBST.hpp` contains class `NodeBST`. This class represent single nodes of binary search tree. This class contains private variable which store informations about parent, left sub-tree, right sub-tree, predecessor and successor in order sequence. There is store index to raw data, size if sub-tree using by statistical functions rank and select and node color using by balanced red-black tree.

Insert item to dictionary One of few major properties of dictionary is insert new context-content pairs to the dictionary. Class `BinarySearchTree` contains function `void insertItem(int contextContentIndex)`. Input parameter of this function is index to raw data which define boundary between context and content. This index is using for comparing context and content.

If the dictionary is empty than newly created node of class `NodeBST` is now root of search tree. After that function to repair tree balancing is call. If in the previous step has reached maximum size dictionary than all dictionary is deleted and processed is similar as process with empty dictionary. If the dictionary is not full yet than function `insertNode(contextContentIndex, dataStorage)` is call to root of search tree.

Inserting new context-content pair is managed by singed nodes `NodeBST`. In the beginning the right place is searching recursively node by node from root to leaf. If insert context is smaller then current context process continue to left sub-tree, otherwise process continue to right sub-tree. If search context is equal to current context than nothing will be insert to the dictionary.

After finding the right place new node is join to the search tree. Pointers to parent and child are initialized, sub-tree size are recalculated up to root. Finally pointers to predecessor and successor are recalculated. In red-black tree variant of search tree the function to check tree balancing `repairRBProperties()` is call.

Binary search tree balancing Balanced binary search tree is implemented as red-black tree. Red-black tree is implemented extending class `NodeBST`. In this class in implemented private helper function `NodeBST* getGrandparent()` and `NodeBST* getUncle()` for search special nodes. This functions returning found node or NULL if node did not exist. There are implemented function for four rotations: `NodeBST* rotateRight()`, `NodeBST* rotateLeft()`, `NodeBST* rotateLeftRight()`, `NodeBST* rotateRightLeft()`. Main function is `void repairRBProperties()`. This function is manage rotations on unbalance search tree.

Function for individually rotations are implemented as describe chapter 3.1.2. Function `void repairRBProperties()` is always call to new inserted node. Function iterate over particular cases violate red-black properties. Solution of each violation is describe in chapter 3.1.2. This function can be call recursively to root unlit tree is not balance.

Search the best context Class `BinarySearchTree` contains function `NodeBST* searchBest(int searchedContextContentIndex)` used to search the best context. Input parameter of this function is index to raw data. Function return pointer to node `NodeBST` which context is the most similar to search context or NULL if dictionary is empty. If dictionary is not empty function `search(searchedContextContentIndex, true)` is called to root of search tree.

Class `NodeBST` contains function `NodeBST* search(int searchContextContentIndex, bool approximately)`. This function searching recursively from root to leaf context which is the most similar to search context. In each iteration is compare search context and current context. If search context is equal to current context than this node is result. If current node is leaf and current context is not equal to search context than result is one of two nearest nodes which context is more similar to search context.

Search the best content Class `BinarySearchTree` contains two functions to search the best content. One of them search the lexicographically most similar content to search content. Second of them search content with smaller index and also has the maximal LCP with search content.

Function for searching lexicographically most similar content is `NodeBST* searchContent(NodeBST *contextNode, int contextContentIndex`. This function starts at same position as position of the best content. This function searching content in surrounding defined by number of bits using to encode different between best context and best content. This function return content the most similar to search content.

Function for searching content with smaller index and also has the maximal LCP with search content is `NodeBST* searchContentMinimal(NodeBST *contextNode, int contextContentIndex`. This function is remember the best content and value of LCP with search content. If current LCP is grater then remember LCP than remember content is replace by current content. If current LCP is equal to remember LCP than content is remember content is replace by current content only if current content has lower index then remember content. Finally function return remember content.

Search the second best content Class `BinarySearchTree` contains function `NodeBST* searchContentTwoBestMinimal(NodeBST *context, NodeBST *content, int bytePositionLimit)` using for search the second best content. The second best content is content which is the lexicographically most similar to the best content and its index is lower then index of the best content. This definition of the second best content is used to easier design new improvements of ACB algorithm.

The second best content is searching in identically surrounding of the best context as searching the best content. Result is content satisfy the definition or NULL if that content did not exist.

Search the best content during decompression During compression and decompression are different input informations. During compression is known all raw data. During decompression is knows only processed data and additional informations using to decompression next bytes. Compression procedure using part data which is not known during decompression. On the other hand decompression procedure knows different between best content and best context which was calculate during compression. Using this different may be calculated position of the best content.

Function `NodeBST* getContentByOffset(NodeBST *contextNode, int offset)` from class `BinarySearchTree` return the best content. Input parameters of this functions are the best context and offset to shift index. Searching is only traversing neighbours nodes by offset distance.

Rank & Select Functions rank and select are statistical functions provide additional informations of nodes binary search tree. Function rank return node position in sequence order by the best context. Function select return n^{th} node in this sequence. Detail information about this functions are describe in chapter 3.1.3.

Class `BinarySearchTree` contains function `int rank(NodeBST *node)`. This function traversal nodes of binary search tree from original node to root. In each iteration calculate rank of original node. If current node is right child its parent than result is increase by size of left sub-tree its parent.

Class `BinarySearchTree` contains function `NodeBST* select(int rank)`. This function traversal nodes of binary search tree from root to leaf. In each iteration calculate rank of current node and compare it with search rank. If rank of current node is equal to search rank this node is return. If search rank is lower then current rank than continue to left sub-tree. Otherwise continue to right sub-tree and search rank is decreased by current rank plus one.

4.5 Storage of data

Storage of all data needed for compression and decompression is contained in source files `DataStorage.cpp` and `Triplets.cpp` and in header files `DataStorage.hpp` and `Triplets`.

Files `Triplets.cpp` and `Triplets.hpp` contain class `Triplets`. This class is using for store three numbers of informations needed for decompression. These numbers represented different between best content and best context, number of bytes copy from the best content and value of next byte from current content.

Files `DataStorage.cpp` and `DataStorage.hpp` contain class `DataStorage`. In private variable are store array of raw data and its size, array of triplets and its count. There are store input parameters needed to true compare contexts and contents. These parameters are maximal context length, maximal content length, maximal difference between best content and best context and maximal number of bytes which can be copy from the best content.

Adding and getting raw data and triplets For managing arrays of raw data and triplets are implemented functions loading and storing this data. New item is insert always to end of array. Any item from array can be taken.

Function `int addByte(unsigned char byte)` is using to add new byte to raw data, function `int addTriplet(int contextContentDiff, int sameBytesCount, unsigned char newByte)` is using to add new triplet. This functions manage dynamic allocated memory.

Function `unsigned char getByte(int position)` is using to get byte from raw data, function `Triplet getTriplet(int position)` is using to get triplet. Both function return item at a given position if this item exists. Otherwise return zero or triplet of zeros.

Context comparison Algorithm ACB comparing context lexicographically from right to left. All raw data in store in class `DataStorage`. Context is defined by index to this raw data. Context start at position defined by index minus one.

Class `DataStorage` contains function `int compareContext(int actual-ContextContentIndex, int BSTContextContentIndex)` used to compare contexts. Input parameters this function are indexes to raw data. Function return 1 if first context is greater than second context. Function return -1 if first context is lower then second context. If both contexts are equal function return 0.

Raw data is compare byte by byte from starts positions to left. Comparison finished when bytes are different or maximal context length is exceeded.

Class `DataStorage` contains function `int compareContextSimilarity( int actualContextContentIndex, int contextContentIndexA, int contextContentIndexB)` using for choice lexicographically more similar context. Function return 0 if both contexts are equal. Function return 1 if context A is lexicographically more similar to search context then content B. Otherwise function return -1.

Content comparison Algorithm ACB comparing content lexicographically from left to right. All raw data in store in class `DataStorage`. Content is defined by index to this raw data. Context start at position defined by index.

Class `DataStorage` contains function `int int contextContentIndexA, int contextContentIndexB` using to compare contents. Input parameters this function are indexes to raw data. Function return 1 if content A is greater than content B. Function return -1 if content A is lower then content B. If both contents are equal function return 0.

Raw data is compare byte by byte from starts positions. Comparison finished when bytes are different or maximal content length is exceeded.

Class `DataStorage` contains function `int compareContentSimilarity(-int actualContextContentIndex, int contextContentIndexA, int contextContentIndexB)` using for choice lexicographically more similar content. Function return 0 if both contents are equal. Function return 1 if content A is lexicographically more similar to search content then content B. Otherwise function return -1.

Contents common length Algorithm ACB after found the best context have to calculate number of identically bytes in the best content and current content from start this contents. This number is equal to number of bytes which can be copy from the best content.

Class `DataStorage` contains function `int contentMatchCount(int actualContextContentIndex, int BSTContextContentIndex, int lcp = 0)` using this comparison. Contents are compare lexicographically byte by byte until are not different. Maximal number of compare bytes is limited by maximal content length or maximal number of bits use to encode number of bytes copy from the best content.

This function has variable parameter `int lcp = 0`. This parameter is used by new variant of ACB algorithm. If value of parameter is greater then zero than can be compare more bytes. Limit define by number of bits use to encode number of copy bytes is increase by this parameter. It is due to the fact that this value is subtracted from the number of copy bytes form the best content.

LCP calculation New variant of ACB algorithm require calculation LCP of the best content and the second best content.

Class `DataStorage` contains function `int contentLcp(int contextContentIndexA, int contextContentIndexB)` using for LCP calculation. This function is similar to function for compare contents common length. This function return number of identically bytes in both contents.

4.6 Arithmetic module

Input of compression procedure in sequence of bytes, output of compression procedure is sequence of numbers. Input of decompression procedure is sequence of numbers, output of decompression procedure is sequence of bytes. Store numbers to memory is ineffective. It is convenient compress this numbers with other algorithm designed to compression numbers. In his implementation is used adaptive arithmetic coding.

Compression library ExCom [10] contains module for arithmetic coding. For more simple usage this module is implemented interface for this module. This module interface is contained in source file `ArithCoderExCom.cpp` and header file `ArithCoderExCom.hpp`.

Files `ArithCoderExCom.cpp` and `ArithCoderExCom.hpp` contain class `ArithEncoder` using for encoding numbers by arithmetic coder and class `ArithDecoder` using for decoding this number by arithmetic coder.

Adaptive arithmetic coding using dictionary with probabilities of occurrence for each symbol. The greater probability of a given symbol, the less bits will be used for its encoding. Algorithm ACB write to output stream triplet of numbers. Each number has different meaning and different probabilities of occurrence. There are using three different alphabets each for one number. Arithmetic coding achieves better compression ratio.

Module for arithmetic coding from ExCom library allow to use more different dictionaries together. Each dictionary is define by size of alphabet. During encoding and decoding each number is choice just one dictionary.

4.6.1 Arithmetic encoder

Class `ArithEncoder` is used as interface to arithmetic coding module from ExCom library. In private variables are store pointer to arithmetic coder and pointers to each alphabets.

In constructor is created arithmetic coder for output stream to write encoded data. Here are set size of alphabets by input parameters.

In arithmetic coding have to be marked end of data for decoder. Otherwise it is not possible to know end of data. The marker is realize by using one higher size of alphabet for encoding next bytes for current content. Number 256 means end of data.

Functions `void encodeDifference(int symbol)`, `void encodeCopy(int symbol)`, `void encodeByte(int symbol)` and `void encodeFlag(int symbol)` is using for encoding numbers using the appropriate alphabet. Function `void flush()` is using to write marker to output stream.

4.6.2 Arithmetic decoder

Class `ArithDecoder` is used as interface to arithmetic encoding module from ExCom library. In private variables are store pointer to arithmetic coder and pointers to each alphabets.

In constructor is created arithmetic decoder for input stream to read encoded data. Here are set size of alphabets by input parameters.

Functions `int decodeDifference()`, `int decodeCopy()`, `int decodeByte()` and `int decodeFlag()` is using for decoding numbers using the appropriate alphabet. Function `bool isEof()` is using to read marker from input stream. When it was read the last number yet return true, otherwise return false.

Chapter 5

Experimental measurement

In chapter 2 are describe previous implementations of ACB algorithm. At the end of this chapter is table 2.1 contain comparison compression ratios the algorithms. The best compression ration has original implementation of ACB algorithm by George Buyanovsky, the second best is ACB algorithm from ExCom library [10] implemented by Michal Valach [11]. Different variants of ACB algorithm implemented in this thesis are compared with implementation by Michal Valach.

This chapter are describe experimental measurements and results. There are compare compression ratio and computing speed of three variants of ACB algorithm implemented in this thesis.

The testing platform included the AMD E-350 Processor with 1.60 GHz (64-bit architecture) and 4 GB DDR3 RAM. The operating system was Zorin OS 10 (based on Ubuntu 15.04) with 3.9.0-58-generic linux kernel version. ExCom library and all implemented methods were compiled with gcc 4.9.2.

All experimental measurements were performed on Canterbury Corpus, Calgary Corpus and Prague Corpus.

5.1 Compression corpus

Compression corpus is a set of various files, used for evaluating different compression algorithms. Main advantages of using corpus is content of various file that examine various properties of the tested algorithms.

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 was firstly published in their paper Modeling For Text Compression [2] in 1989. It consists of 18 files (9 different types) with total size 3,266,560 bytes. This files are describe in table 5.1.

Table 5.1: Calgary corpus file 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
paper3	46,526	english text
paper4	13,286	english text
paper5	11,954	english text
paper6	38,105	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 Canterbury Corpus

The Canterbury Corpus was published by Ross Arnold and Tim Bell [1] in 1997. This corpus consists of 11 distinct files of a total size 2,810,784 bytes, this files are describe in table 5.2. The corpus collection was also designed to satisfy the following conditions: the files should be all public domain and the total size should not be bigger than necessary to ensure the useful distribution.

Table 5.2: Canterbury corpus file description

File name	Size [B]	Description
alice29.txt	152,089	nnglish text
asyoulik.txt	125,179	Shakespeare
cp.html	24,603	HTML source code
fields.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

Table 5.3: Prague corpus file 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

5.1.3 Prague Corpus

Prague Corpus was published by Jakub Řezníček in his master thesis Corpus for comparing compression methods and an extension of a ExCom library [7] in 2010. This corpus consists of 30 files from 8 categories of a total archive size 58,265,600 bytes. This files are describe in table 5.3.

Table 5.4: Compression ratios of different variants ACB algorithm for Canterbury corpus

	Compression ratio [-]			
File	Valach	Léhar	Léhar V1	Léhar V2
alice29.txt	0.338	0.354	0.333	0.503
asyoulik.txt	0.380	0.398	0.376	0.565
cp.html	0.353	0.366	0.348	0.508
fields.c	0.307	0.319	0.310	0.530
grammar.lsp	0.396	0.406	0.392	0.565
kennedy.xls	0.199	0.228	0.292	0.422
lcet10.txt	0.297	0.313	0.294	0.452
plrabn12.txt	0.364	0.382	0.363	0.544
ptt5	0.112	0.124	0.119	0.165
sum	0.362	0.380	0.370	0.579
xargs.1	0.474	0.489	0.470	0.665

5.2 Performance experiments

5.2.1 Comparison implemented variants ACB algorithm

In this section are comparing 4 different implementations of the algorithm ACB. First implementation is ACB algorithm from ExCom library implemented by Michal Valach. The second implementation is original implementation from this thesis. The third implementation is variant V1 (Subtracting LCP from number of copy symbols) of algorithm from this thesis. The last implementation is variant V2 (Number of copy symbols is equal to value of LCP) of algorithm from this thesis. Three variants of algorithm from this thesis are describe in chapter 3.5

All four implementations were tested on Canterbury Corpus, Calgary Corpus and Prague Corpus. All algorithms were measured with same input parameters.

Results measure on Calgary Corpus are in table 5.4. The best of new implementations is variant V1 (Subtracting LCP from number of copy symbols). Only on one file (kennedy.xls) has better compression ration original variant of algorithm. Variant V2 (Number of copy symbols is equal to value of LCP) is not compute as good results as other variants. Comparing to implementation by Michal Valach has variant V1 better compression ratio on 7 files from 11 files. The bigger different of compression ration is on file kennedy.xls, implementation by Michal Valach is better then variant V1 about 10%.

Results measure on Canterbury Corpus are in table 5.5. The best of new implementations is variant V1 (Subtracting LCP from number of copy symbols). This variant has the best compression ration on all files. Variant V2 (Number of copy symbols is equal to value of LCP) is not compute as good

Table 5.5: Compression ratios of different variants ACB algorithm for Calgary corpus

	Compression ratio [-]			
File	Valach	Léhar	Léhar V1	Léhar V2
bib	0.301	0.315	0.298	0.476
book1	0.360	0.378	0.354	0.514
book2	0.316	0.321	0.302	0.460
geo	0.660	0.691	0.685	0.879
news	0.354	0.370	0.350	0.540
obj1	0.532	0.543	0.531	0.754
obj2	0.359	0.373	0.360	0.562
paper1	0.357	0.374	0.355	0.558
paper2	0.356	0.372	0.351	0.536
paper3	0.396	0.414	0.391	0.587
paper4	0.447	0.465	0.443	0.635
paper5	0.458	0.476	0.454	0.653
paper6	0.366	0.381	0.362	0.569
pic	0.112	0.124	0.119	0.165
progc	0.358	0.372	0.356	0.571
progl	0.242	0.250	0.237	0.402
progp	0.238	0.246	0.235	0.414
trans	0.207	0.214	0.204	0.401

results as other variants. Comparing to implementation by Michal Valach has variant V1 better compression ratio on 15 files from 18 files. The bigger different of compression ration is on file geo, implementation by Michal Valach is better then variant V1 about 2.5%.

Results measure on Prague Corpus are in table 5.6. The best of new implementations is variant V1 (Subtracting LCP from number of copy symbols). Only on one file (firewrks) has better compression ration variant V2 (Number of copy symbols is equal to value of LCP). Variant V2 (Number of copy symbols is equal to value of LCP) is not compute as good results as other variants. Comparing to implementation by Michal Valach has variant V1 better compression ratio on 13 files from 28 files, on 2 files was the compression ratio same. For compression of file firewrks is the best algorithm variant 2, only this algorithm has compression ration smaller then one.

The best compression ratio has variant V1 (Subtracting LCP from number of copy symbols) of algorithm implemented in this master thesis. The best results were measure on Canterbury Corpus and Calgary Corpus. Main advantage of this variant is opportunity to copy more bytes from the best context with same number of bits using to encode this information. The second best is implementation of ACB algorithm by Michal Valach. Variant V2 (Number

Table 5.6: Compression ratios of different variants ACB algorithm for Prague corpus

	Compression ratio [-]			
File	Valach	Léhar	Léhar V1	Léhar V2
abbot	0.990	0.970	0.968	0.946
age	0.456	0.472	0.465	0.684
bovary	0.303	0.302	0.284	0.394
collapse	0.454	0.462	0.450	0.654
compress	0.197	0.205	0.195	0.339
corilis	0.543	0.537	0.519	0.635
cyprus	0.029	0.033	0.031	0.098
drkonqi	0.380	0.394	0.385	0.586
emission	0.116	0.125	0.123	0.242
firewrks	1.009	1.007	1.005	0.947
flower	0.517	0.494	0.475	0.660
gtkprint	0.326	0.339	0.333	0.510
handler	0.264	0.272	0.264	0.438
higrowth	0.424	0.449	0.446	0.658
hungary	0.024	0.024	0.023	0.080
libc06	0.376	0.394	0.387	0.605
lusiadas	0.316	0.336	0.324	0.469
lzfindmt	0.231	0.239	0.231	0.437
mailflder	0.225	0.235	0.225	0.389
mirror	0.426	0.442	0.431	0.624
modern	0.362	0.380	0.357	0.524
nightsht	0.876	0.949	0.932	0.991
render	0.263	0.271	0.262	0.454
thunder	0.767	0.824	0.817	0.926
ultima	0.706	0.692	0.675	0.715
usstate	0.280	0.289	0.278	0.415
venus	0.771	0.777	0.744	0.832
wnvcrdt	0.046	0.054	0.051	0.089
w01vett	0.050	0.057	0.053	0.121
xmlevent	0.318	0.328	0.320	0.514

of copy symbols is equal to value of LCP) unfulfilled expectations. From the best content is not copy enough bytes to get better compression ration but average compression ration about 0.53 is not poor.

5.2.2 Index shifting

Main idea of this variant of algorithm describe in chapter 3.5.3 is found better context for current content. This variant is searching the best content in surrounding of the best context shifted by different between best content and best context from last iteration. It is likely that there will be found the better context than in normal surrounding of the best context when to be found here in last iteration.

Results were measure for all three new implemented variants of ACB algorithm. Results measure on Canterbury Corpus are in table 5.7, results measure on Calgary Corpus are in table 5.8.

The results for all files on both corpora are almost identical. It is not depend of variant of algorithm. Specifically variant without index shifting always has better compression ratio of 1-3% then variant with index shifting.

The idea about occurrence better content in shift surrounding was not confirmed. Better context were found in normal surrounding than shift surrounding.

This variant of algorithm will not use later because its not improve compression ratio.

Table 5.7: Compression ratios of index shifting variant for Canterbury corpus

	Compression ratio [-]					
	Léhar		Léhar V1		Léhar V2	
File	orig.	shift	orig.	shift	orig.	shift
alice29.txt	0.354	0.375	0.333	0.354	0.503	0.516
asyoulik.txt	0.398	0.422	0.376	0.398	0.565	0.577
cp.html	0.366	0.385	0.348	0.367	0.508	0.518
fields.c	0.319	0.333	0.310	0.324	0.530	0.547
grammar.lsp	0.406	0.425	0.392	0.410	0.565	0.577
kennedy.xls	0.228	0.242	0.292	0.278	0.422	0.435
lcet10.txt	0.313	0.334	0.294	0.314	0.452	0.464
plrabn12.txt	0.382	0.407	0.363	0.385	0.544	0.555
ptt5	0.124	0.130	0.119	0.126	0.165	0.165
sum	0.380	0.397	0.370	0.388	0.579	0.586
xargs.1	0.489	0.510	0.470	0.491	0.665	0.680

Table 5.8: Compression ratios of index shifting variant for Calgary corpus

	Compression ratio [-]					
	Léhar		Léhar V1		Léhar V2	
File	orig.	shift	orig.	shift	orig.	shift
bib	0.315	0.334	0.298	0.317	0.476	0.487
book1	0.378	0.402	0.354	0.376	0.514	0.525
book2	0.321	0.342	0.302	0.322	0.460	0.472
geo	0.691	0.698	0.685	0.692	0.879	0.879
news	0.370	0.393	0.350	0.371	0.540	0.550
obj1	0.543	0.557	0.531	0.546	0.754	0.757
obj2	0.373	0.391	0.360	0.378	0.562	0.572
paper1	0.374	0.397	0.355	0.378	0.558	0.572
paper2	0.372	0.396	0.351	0.375	0.536	0.551
paper3	0.414	0.440	0.391	0.416	0.587	0.601
paper4	0.465	0.493	0.443	0.471	0.635	0.651
paper5	0.476	0.500	0.454	0.479	0.653	0.668
paper6	0.381	0.406	0.362	0.384	0.569	0.584
pic	0.124	0.130	0.119	0.126	0.165	0.165
progc	0.372	0.393	0.356	0.377	0.571	0.584
progl	0.250	0.265	0.237	0.252	0.402	0.413
progp	0.246	0.261	0.235	0.249	0.414	0.424
trans	0.214	0.227	0.204	0.217	0.401	0.406

5.2.3 Number of bits to encode difference

Number of bits using to encode difference between best content and best context define size of surrounding of the best context to search the best content. The greater number of these bits, the greater probability to found better content.

The fewer number of these bits, the smaller input alphabet of arithmetic coder. Smaller number of symbols in input alphabet allow better compression ratio of arithmetic coder.

The goal is to find a compromise for the best compression ratio.

Results measure on Canterbury Corpus are in table 5.9. Number of bits to better compression ratio is highly dependent on input file. In average case is the best using 10 bits to encode difference between best content and best context on Canterbury Corpus.

Results measure on Calgary Corpus are in table 5.10. In average case is the best using 8 bits to encode difference between best content and best context on Calgary Corpus.

Find the optimal number of bits to encode difference between best content and best context is very difficult. The best boundary between competing demands is highly dependent on input file.

Table 5.9: Compression ratios of different number of bits to encode different between best content and best context for Canterbury corpus

	Compression ratio [-]			
	Number of bits			
File	4	6	8	10
alice29.txt	0.352	0.340	0.333	0.333
asyoulik.txt	0.397	0.384	0.376	0.375
cp.html	0.342	0.342	0.348	0.353
fields.c	0.307	0.309	0.310	0.317
grammar.lsp	0.382	0.386	0.392	0.400
kennedy.xls	0.260	0.279	0.292	0.215
lcet10.txt	0.314	0.302	0.294	0.290
plrabn12.txt	0.390	0.374	0.363	0.358
ptt5	0.119	0.119	0.119	0.118
sum	0.370	0.369	0.370	0.371
xargs.1	0.456	0.461	0.470	0.480

Table 5.10: Compression ratios of different number of bits to encode different between best content and best context for Calgary corpus

	Compression ratio [-]			
	Number of bits			
File	4	6	8	10
bib	0.310	0.302	0.298	0.297
book1	0.380	0.364	0.354	0.350
book2	0.320	0.309	0.302	0.298
geo	0.670	0.680	0.685	0.666
news	0.362	0.355	0.350	0.348
obj1	0.514	0.520	0.531	0.546
obj2	0.362	0.359	0.360	0.360
paper1	0.363	0.356	0.355	0.357
paper2	0.367	0.357	0.351	0.353
paper3	0.402	0.394	0.391	0.394
paper4	0.440	0.440	0.443	0.450
paper5	0.449	0.450	0.454	0.462
paper6	0.369	0.364	0.362	0.366
pic	0.119	0.119	0.119	0.118
progc	0.361	0.355	0.356	0.361
progl	0.243	0.239	0.237	0.238
progp	0.239	0.235	0.235	0.238
trans	0.207	0.204	0.204	0.206

Table 5.11: Compression ratios of different number of bits to encode number of copy bytes from best content for Canterbury corpus

Compression ratio [-]				
	Number of bits			
File	4	6	8	10
alice29.txt	0.332	0.332	0.333	0.337
asyoulik.txt	0.375	0.374	0.376	0.380
cp.html	0.350	0.344	0.348	0.360
fields.c	0.309	0.303	0.310	0.326
grammar.lsp	0.377	0.376	0.392	0.422
kennedy.xls	0.291	0.291	0.292	0.294
lcet10.txt	0.295	0.293	0.294	0.296
plrabn12.txt	0.362	0.362	0.363	0.366
ptt5	0.139	0.125	0.119	0.120
sum	0.384	0.370	0.370	0.378
xargs.1	0.449	0.453	0.470	0.502

5.2.4 Number of bits to encode copy

Number of bytes which can be copy from the best content is defined by number of bits using to encode this information. The greater number of these bits, the more bytes can be copy from the best content.

The fewer number of these bits, the smaller input alphabet of arithmetic coder. Smaller number of symbols in input alphabet allow better compression ratio of arithmetic coder.

The goal is to find a compromise for the best compression ratio.

Results measure on Canterbury Corpus are in table 5.11, results measure on Calgary Corpus are in table 5.12.

Compression ratio is not very dependent on the number of bit using to encode number of bytes copy from the best content. Different of compression ratio is about 1% dependent on input file.

5.2.5 Maximal size of dictionary

The most memory consumption of compression and decompression is store of dictionary. Dictionary size is linear dependent on size of input file.

One of the variant to reduce memory consumption of ACB algorithm is limited maximal size of dictionary. This variant is describe in chapter 3.4.2.

Effect of maximal dictionary size were measure for variant V1 (Subtracting LCP from number of copy symbols) which has the best compression ratio of all new implemented variants. Maximum size of dictionary was vote with respect to size of files. Measure results are comparing to unlimited dictionary size.

Table 5.12: Compression ratios of different number of bits to encode number of copy bytes from best content for Calgary corpus

	Compression ratio [-]			
	Number of bits			
File	4	6	8	10
bib	0.302	0.297	0.298	0.302
book1	0.354	0.354	0.354	0.357
book2	0.303	0.302	0.302	0.304
geo	0.685	0.683	0.685	0.692
news	0.356	0.350	0.350	0.352
obj1	0.542	0.529	0.531	0.546
obj2	0.369	0.361	0.360	0.363
paper1	0.355	0.352	0.355	0.362
paper2	0.349	0.349	0.351	0.357
paper3	0.388	0.388	0.391	0.400
paper4	0.433	0.435	0.443	0.462
paper5	0.444	0.445	0.454	0.474
paper6	0.362	0.359	0.362	0.371
pic	0.139	0.125	0.119	0.120
progc	0.357	0.353	0.356	0.365
progl	0.246	0.239	0.237	0.243
progp	0.246	0.236	0.235	0.241
trans	0.221	0.207	0.204	0.208

Table 5.13: Compression ratios of different dictionary size for Canterbury corpus

	Compression ratio [-]				
	Dictionary size				
File	1,000	2,000	5,000	10,000	∞
alice29.txt	0.544	0.507	0.461	0.429	0.333
asyoulik.txt	0.567	0.532	0.491	0.463	0.376
cp.html	0.525	0.481	0.429	0.393	0.348
fields.c	0.434	0.386	0.350	0.320	0.310
grammar.lsp	0.438	0.406	-	-	0.392
kennedy.xls	0.215	0.212	0.203	0.197	0.292
lcet10.txt	0.544	0.503	0.453	0.419	0.294
plrabn12.txt	0.577	0.547	0.508	0.480	0.363
ptt5	0.115	0.114	0.114	0.116	0.119
sum	0.468	0.451	0.428	0.379	0.370
xargs.1	0.576	0.524	-	-	0.470

Table 5.14: Compression ratios of different dictionary size for Calgary corpus

	Compression ratio [-]				
	Dictionary size				
File	1,000	2,000	5,000	10,000	∞
bib	0.564	0.517	0.459	0.418	0.298
book1	0.588	0.555	0.513	0.483	0.354
book2	0.543	0.503	0.455	0.423	0.302
geo	0.746	0.731	0.719	0.708	0.685
news	0.606	0.558	0.503	0.469	0.350
obj1	0.572	0.553	0.533	0.524	0.531
obj2	0.488	0.446	0.407	0.389	0.360
paper1	0.557	0.509	0.463	0.431	0.355
paper2	0.556	0.515	0.467	0.436	0.351
paper3	0.573	0.535	0.492	0.459	0.391
paper4	0.570	0.533	0.489	0.462	0.443
paper5	0.572	0.534	0.498	0.471	0.454
paper6	0.543	0.499	0.455	0.426	0.362
pic	0.115	0.114	0.114	0.116	0.119
progc	0.523	0.484	0.438	0.404	0.356
progl	0.401	0.351	0.293	0.271	0.237
progp	0.412	0.361	0.290	0.267	0.235
trans	0.490	0.433	0.359	0.297	0.204

Compression ratio of files with smaller size then maximal dictionary size in not published. This compression ratio is identically as compression ratio unlimited dictionary size.

Results measure on Canterbury Corpus are in table 5.13, results measure on Calgary Corpus are in table 5.14.

On almost all files is better compression ratio with larger maximal dictionary size. In bigger dictionary is found the best context more similar to current context than is more probability to found the best content more similar to current content and can be copy more bytes.

In some cases is not better compression ration with larger maximal dictionary size. If current compression text is not dependent on its context it can be found more similar content to current content in smaller dictionary with only newest context-content pairs. Searching of the best content is limited to surrounding of the best context.

Even in cases of very limited memory (1000 context-content pairs in dictionary) algorithm does not have wrong compression ratio.

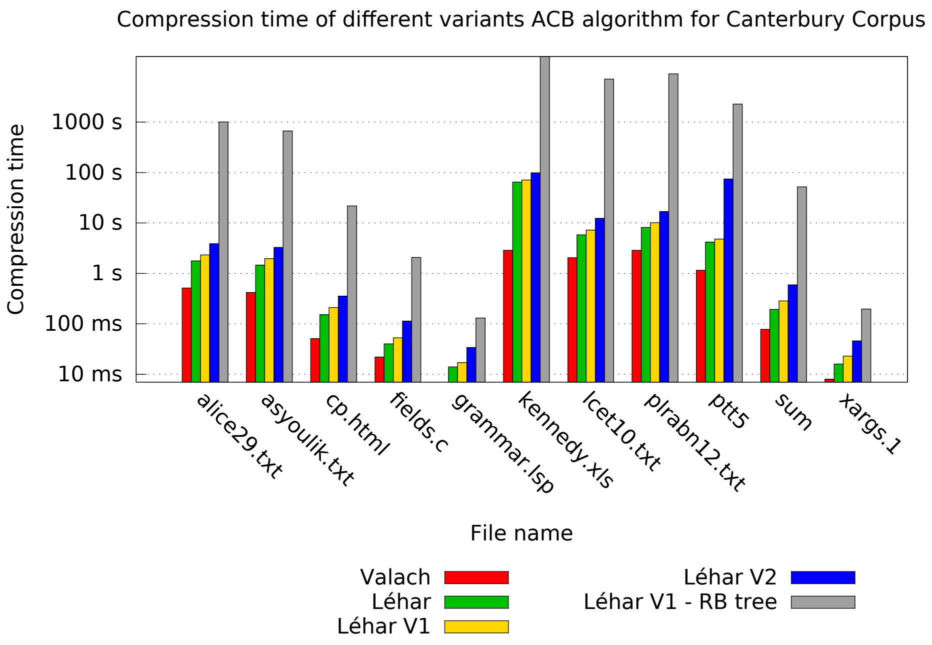

Figure 5.1: Compression time of different variants ACB algorithm for Canterbury corpus

5.2.6 Compression and decompression time

Main goal this master thesis is achieve the best possible compression ratio. Computing speed is also important too. There is measure compression and decompression times.

Compression and decompression time is measure for all variants of algorithm implemented in this thesis. For variant V1 (Subtracting LCP from number of copy symbols) with the best compression ratio is measure version with red-black tree instead of binary search tree on Canterbury Corpus.

Results measure compression time on Canterbury Corpus are in graph on figure 5.2.6, decompression times of this files are in graph on figure 5.2.6. Results measure compression time on Calgary Corpus are in graph on figure 5.2.6, decompression times of this files are in graph on figure 5.2.6.

The best compression times were measure for actual implementation of ACB algorithm in ExCom library implemented by Michal Valach. Original implementation in this thesis is about 2-3x slower. Variant V1 (Subtracting LCP from number of copy symbols) implemented in this thesis is about 3-5x slower then implementation by M. Valach, variant V2 (Number of copy symbols is equal to value of LCP) is about 8-10x slower then implementation by M. Valach.

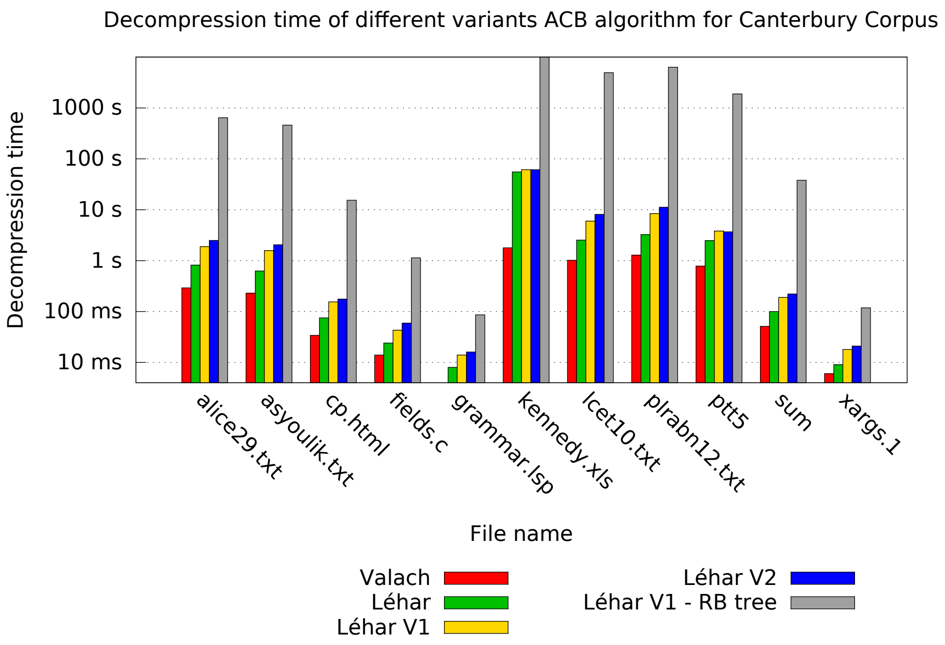

Figure 5.2: Decompression time of different variants ACB algorithm for Canterbury corpus

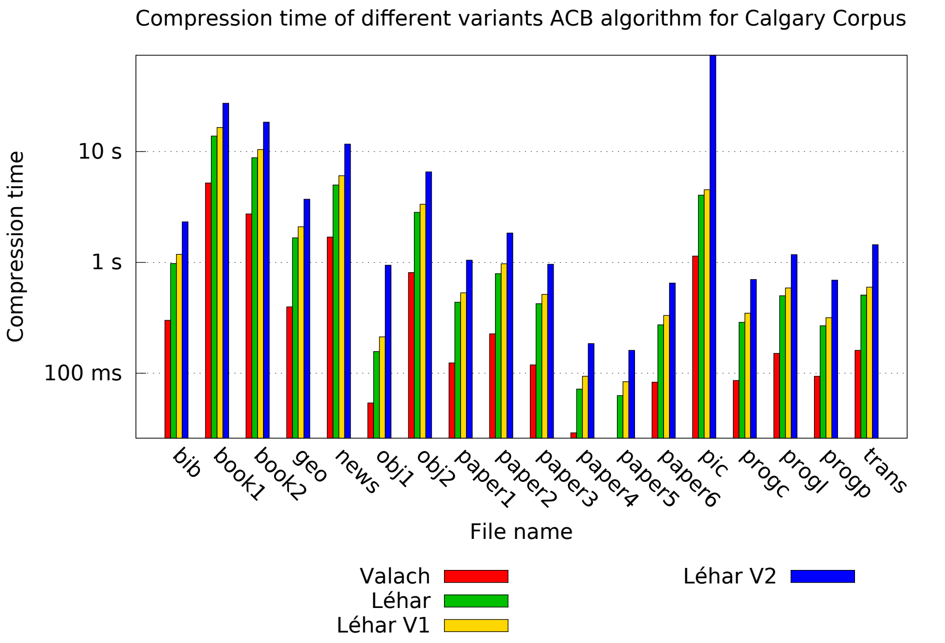

Figure 5.3: Compression time of different variants ACB algorithm for Calgary corpus

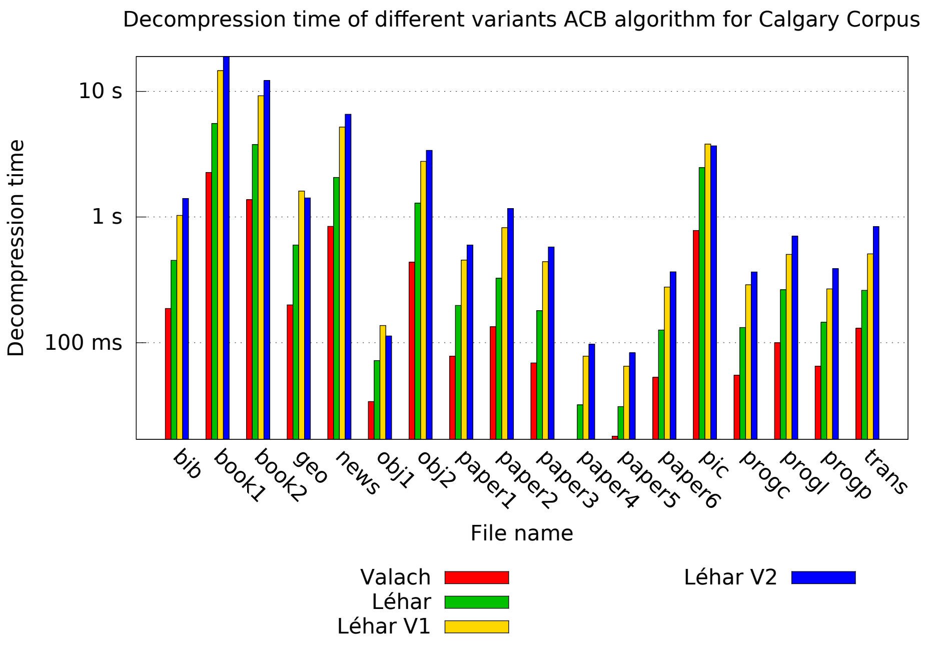

Figure 5.4: Decompression time of different variants ACB algorithm for Calgary corpus

Decompression times for implementation by M. Valach, original implementation in this thesis and variation V2 of this algorithm is about 2x faster then compression times. Decompression times variant V1 of this algorithm is about 1.2-1.5x faster then compression times.

For possible improvements of computing speed was added version with replacing binary search tree in dictionary by its balance variant red-black tree. This version is not applicable because computing times was increasing. Algorithm ACB using the same insert and search operation in dictionary. Insert operation is much more time consuming due tree balancing.

Compression ratio was improved at the expense of speed. Overtime calculation time is constant with file size. In real usage this algorithm depends on the priority compression ratio and computing speed.

Conclusion

The main goal this master thesis was research structures for indexing context and content. The compression ratio should be improvement.

Outcome this thesis is analysis of improvement original algorithm ACB published by G. Buyanovsky. This improvements were implemented in C++ and integrated to compression library ExCom.

This master thesis contain three improvements: Subtracting LCP from number of copy symbols, Number of copy symbols is equal to value of LCP, Index shifting.

Improvement Subtracting LCP from number of copy symbols has improved compression ratio of actual ACB implementation in ExCom library implemented by M. Valach. Compression ratio of original implementation by G. Buyanovsky has not improved.

The other improvements not improved compression but also has good compression ratio.

Newly implemented algorithm to ExCom library has better compression ration then actual implementation. The computing time is about 3-5x slower then computing time of actual implementation.

Future work

The main goal of extending this master thesis is improving computing time of new variant Subtracting LCP from number of copy symbols of ACB algorithm. It would be very well decrease computing time to values as implementation of ACB algorithm by M. Valach.

To improvement the compression ratio could be achieved by replacing actual arithmetic coder with high-quality arithmetic coder. Algorithm implemented in this thesis using arithmetic coder from ExCom library which do not reach optimal compression ratio.

Bibliography

[1] Arnold, R.; Bell, T. A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference, 1997. DCC'97. Proceedings*, IEEE, 1997, pp. 201–210.

[2] Bell, T.; Witten, I. H.; Cleary, J. G. Modeling for text compression. *ACM Computing Surveys (CSUR)*, volume 21, no. 4, 1989: pp. 557–591.

[3] Buyanovsky, G. Description of acb published in comp.compression [ONLINE]. Aug 1996. Available from: `http://www.cbloom.com/news/bygeorge.html`

[4] Cerman, L. Acb compression algorithm. Czech Technical University in Prague, Jan. 2003.

[5] Decky, M. Associative coder of buyanovsky [ONLINE]. 2006. Available from: `http://http://projects.decky.cz/ACB/`

[6] Huffman, D. A.; et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, volume 40, no. 9, 1952: pp. 1098–1101.

[7] Reznicek, J. *Corpus for comparing compression methods and an extension of a excom library.* Master's thesis, Czech Technical University in Prague, May 2010.

[8] Scholtzova, J.; Trvdik, P. Balanced BSTs - AVL and RB Trees. 2014.

[9] Simek, F. *Data compression library.* Master's thesis, Czech Technical University in Prague, May 2009.

[10] Simek, F.; Reznicek, J. ExCom library [ONLINE]. 2009–2010. Available from: `http://www.stringology.org/projects/ExCom/`

[11] Valach, M. *Efficient implementation of ACB compression algorithm for ExCom library.* Master's thesis, Czech Technical University in Prague, May 2011.

[12] Witten, I. H.; Neal, R. M.; Cleary, J. G. Arithmetic coding for data compression. *Communications of the ACM*, volume 30, no. 6, 1987: pp. 520–540.

[13] Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, volume 23, no. 3, 1977: pp. 337–343.

Appendix A

Acronyms

ACB	Associative Coder of Buyanovsky
AVL	Adelson-Velsky and Landis
BST	Binary Search Tree
CCITT	Consultative Committee for International Telephony and Telegraphy
EOF	End Of File
ExCom	Extensible Compression Library
GNU	GNU's Not Unix
GPL	General Public License
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IO	Input/Output
LGPL	Lesser General Public License
LZ77	Lempel-Ziv compression method from 1977
LZ78	Lempel-Ziv compression method from 1978
LCP	Longest Common Prefix
PDF	Portable Document Format
PHP	PHP Hypertext Preprocessor
PPM	Prediction by Partial Matching
RB	Red-Black

SPARC	Scalable Processor ARChitecture
XML	Extensible Markup Language

Appendix B

Building the ExCom library

The source code of ExCom library is placed on enclosed CD in the directory `/excom`. The source code is built using the GNU build system by following instructions.

1. Apply autotools

   ```
   autoreconf -i -s -f
   ```

   This command should be executed from the root directory of the source tree. It will run all necessary autotools programs in correct order. As a result, it will create a script called configure.

2. Run configure

   ```
   mkdir bin
   cd bin
   ../configure -C --enable-perf-measure
   ```

   Although the configure script can be run from the directory where it resides, it's wise to create a distinct build directory to separate the source files from the object files.

   The configure process can be customized by setting environment variables or by passing parameters to the script. The list of all variables and parameters can be obtained by running `configure --help`.

3. Make the library

   ```
   make
   ```

   This command should recursively enter all directories, and build the library and programs. By default, both static and dynamic version of the library is built.

Appendix C

User manual

This appendix describes usage of ExCom library. Console application `app` is located in `bin/src/app` after building the ExCom library. Application contains many parameters for controlling it, each compression method can include new parameters.

Algorithm ACB implemented in this thesis is included in ExCom library as **acb2** module.

To list all available parameters can be used command `./app --help`:

```
Usage: ./app [options]
where options may be:
  -d, --decompress    decompress input file (default is compress)
  -e, --except=<path> path to exceptions' file (required for DCA,
                      ignored for other)
  -f, --ignore-first  don't count first run to the overall timing
                      The first run may be way off because of
                      empty cache
  -h, --help          print this help
  -i, --input=<path>  path to the input file
  -m, --method=<met>  select method <met>, use ? for a list
  -o, --output=<path> path to the output file
  -p, --param=<prm>   <prm> is a comma separated list of
                      parameters of the method, use ? for a list
  -q, --quiet         don't output anything except errors
  -r, --repeat=<T>    repeat the process T times
  -t, --timing        measure time spent by the process
```

To list all existing methods can be used command `./app -m ?`:

```
Supported compression methods:
  copy    Just copies input to output
  acb     Associative coder of Buyanovsky
  acb2    Associative coder of Buyanovsky (version 2)
  arith   Arithmetic coding
  bwt     Burrows-Wheeler transform
  dca     Data compression using antidictionaries
  dhuff   Dynamic Huffman coding
  integer Integer compression methods
  lz77    Lempel-Ziv compression method from 1977
  lz78    Lempel-Ziv compression method from 1978
  lzap    Variant of LZW based on LZMW by Storer from 1988
  lzmw    Lempel-Ziv-Miller-Wegman, variant of LZW from 1985
  lzss    Lempel-Ziv-Storer-Szymanski comp. method from 1982
  lzw     Lempel-Ziv-Welch compression method from 1984
  lzy     Variant of LZW by Dan Bernstein
  mtf     Move-to-front transform
  ppm     Prediction by partial matching
  rle_n   RLE-N compression method
  sfano   Shannon-Fano coding
  shuff   Static Huffman coding
```

To list all available parameters for ACB2 method can be used command `./app -m acb2 -p ?`:

```
Parameters available for compression method 'acb2':
  s=<N>  Dictionary size (N >= 100). Default value is INFINITY.
  x=<N>  Context length (2 <= N <= 1024). Default value is 32.
  n=<N>  Content length (2 <= N <= 1024). Default value is 512.
  d=<N>  Difference #bits (2 <= N <= 16). Default value is 8.
  c=<N>  Copy bytes #bits (2 <= N <= 16). Default value is 8.
```

Appendix D

Contents of enclosed CD

```
readme.txt ...................... the file with CD contents description
├── corpus ........................ the directory with corpus archives
├── excom ............... the directory with source codes of ExCom library
├── src ......... the directory with source codes of new ACB implementation
└── text ................................ the directory with thesis text
    ├── DP_Lehar_Adam_2016.pdf ............ the thesis text in PDF format
    └── src ............... the directory with LaTeX source codes of thesis
```