FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Implementation of BWC compression method and its variants in Java programming language
Student:	Filip Geletka
Supervisor:	Ing. Radomír Polách
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

1) Research BWC compression method and its variants including related algorithms (BWT, MTF, AC, HC, etc.).
2) Design and implement BWC compression method and its variants.
3) Implementation should be done as a part of the library provided by the supervisor.
4) Test the implementation using appropriate compression corpora and create documentation.

References

Will be provided by the supervisor.

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

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 7, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Implementation of BWC compression method and its variants in Java programming language

Filip Geletka

Department of Software Engineering
Supervisor: Ing. Radomír Polách

May 14, 2019

Acknowledgements

I want to thank my thesis advisor Ing. Radomír Polách. The door to Mr Polách's office was always open whenever I ran into a trouble spot or had a question about my research or writing. I must express my very profound gratitude to my parents and my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. I'd also like to extend my gratitude to Dávid Žalúdek for mentoring me throughout my studies. This accomplishment would not have been possible without them. Thank you.

Declaration

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

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

In Prague on May 14, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Geletka, Filip. *Implementation of BWC compression method and its variants in Java programming language.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Táto bakalárska práca sa zameriava na kompresné algoritmy založené na Burrows-Wheelerovej transformácií. Cieľom tejto práce je analýza, implementácia a testovanie Burrows-Wheelerovej transformácie a súvisiacich algoritmov. Implementácia je súčasťou open-source knižnice Small Compression Toolkit, ktorá je vyvíjaná v programovacom jazyku Java. Funkčnosť novo implementovaných algoritmov je overená pomocou skúšobných meraní. Tieto merania zahŕňajú sledovanie kompresného pomeru a času potrebného pre kompresiu a dekompresiu dát.

Klíčová slova kompresia, dekompresia, Java, Burrows-Wheelerová transformácia, Move-to-front transformácia, Run-length kódovanie, Aritmetické kódovanie, Huffmanovo kódovanie

Abstract

This thesis focuses on the topic of data compression, particularly on Burrows-Wheeler compression and its variants. The goal of this thesis is to analyse, implement and test Burrows-Wheeler transform and all related algorithms. These algorithms are: The implementation is done as a part of the Small Compression Toolkit library. All newly implemented features are verified by

experimental measurements. This verification measures the compression ratio and time needed to compress and decompress data.

Keywords compression, decompression, Java, Burrows-Wheeler transform, Move-to-front transform, Run-length encoding, Arithmetic coding, Huffman coding

Contents

Introduction **1**
Motivation . . . 1
Main goals . . . 2
Thesis structure overview . . . 2

1 Basic notions **3**
1.1 Symbol . . . 3
1.2 Alphabet . . . 3
1.3 String . . . 3
1.4 Codeword . . . 3
1.5 Code . . . 4
1.6 Data compression . . . 4
1.7 Data decompression . . . 4
1.8 Compression algorithm . . . 4
1.9 Decompression algorithm . . . 4
1.10 Compression method . . . 4
1.11 Model . . . 5
1.12 Adaptive compression method . . . 5
1.13 Compression ratio . . . 5
1.14 Entropy . . . 5
1.15 Redundancy . . . 6
1.16 Corpus . . . 6

2 Analysis **7**
2.1 Algorithms . . . 7
2.1.1 Burrows-Wheeler transform . . . 7
2.1.2 Move-to-front transform . . . 12
2.1.3 Run-length encoding . . . 14
2.1.4 Adaptive Arithmetic coding . . . 17

2.1.5 Adaptive Huffman coding . . . 18
2.2 Burrows-Wheeler transform and bzip algorithms . . . 21
2.3 Non-functional requirements . . . 21
2.4 Functional requirements . . . 22
2.5 Related work . . . 22
2.6 Associated theses . . . 23

3 Design and implementation **25**
3.1 Overview . . . 25
3.2 External libraries . . . 25
3.3 Dataflow in application . . . 26
3.4 Input and output . . . 27
3.5 Burrows-Wheeler transform . . . 27
3.6 Move-to-front transform . . . 28
3.7 Run-length encoding . . . 28
3.8 Coder . . . 29
3.9 Client . . . 30

4 Measurements and results **31**
4.1 Overview . . . 31
4.2 Calgary corpus . . . 31
4.3 Canterbury corpus . . . 32
4.4 Prague corpus . . . 33

Conclusion **35**
Future work . . . 35

Bibliography **37**

A Acronyms **41**

B Contents of enclosed memory card **43**

List of Figures

2.1 Example of the Arithmetic coding. 18
2.2 Example of the Huffman coding. 20

3.1 UML class diagram for `ChainBuilder`. 26
3.2 Default dataflow and component chaining. 27
3.3 UML class diagram for the BWT algorithm. 27
3.4 UML class diagram for the MTF algorithm. 28
3.5 UML class diagram for the RLE algorithm. 28
3.6 UML class diagram for the coder. 29

List of Tables

2.1 Example of the BWT algorithm 11
2.2 Computed symbols counts for a given example 11
2.3 Computed running totals for a given example 12
2.4 Computed transformation vector for a given example 12
2.5 Example of the MTF algorithm 14
2.6 Example of the inverse MTF algorithm 14
2.7 Example of the RLE algorithm 16
2.8 Example of the inverse RLE algorithm 17

4.1 Performance testing results for the Calgary corpus. 32
4.2 Performance testing results for the Canterbury corpus. 33
4.3 Performance testing results for the Prague corpus. 34

Introduction

Motivation

Use of computers on a regular basis in our life is very important. Computers are used to convert raw facts and data into meaningful information and knowledge. In order to be as efficient as possible in storing and transferring these data, many compression algorithms have been invented. However, only a few of them are so efficient in the matter of speed, reliability and memory efficiency that they became standardized and widely used. Example of such algorithms would be LZ77, LZ78, LZW, RLE or Huffman and Arithmetic coding. These algorithms can standalone achieve great compression ratios with adequate time, but by providing more-easily-compressible data as input for these algorithms, even better performance can be achieved.

Burrows-Wheeler compression (BWC for short) is a general name for compression method which is based on Burrows-Wheeler transform (BWT from now on). The BWT is an algorithm used to prepare data for use with data compression techniques such as Huffman and Arithmetic coding. It was invented and described by Michael Burrows and David Wheeler in 1994. It is based on a previously unpublished transformation discovered by Wheeler in 1983[1]. The transform is done by creating a table of all the circular shifts of data bytes, followed by lexicographical sort and by extracting the last column and the index of the original data in the set of sorted permutations. The remarkable thing about the BWT is not that it generates a more easily encoded output, an ordinary sort would do that, but that it is reversible, allowing the original data to be regenerated from the transformed data.

Basic and the most common version of the BWC is the one where BWT is followed by Move-to-front transform (MTF from now on), Run-length encoding (RLE for short) and some entropy coder, such as Huffman coder (HC) and Arithmetic coder (AC). Enhanced implementation of this version is used by *bzip* and *bzip2* compression programs[2].

Main goals

The main objective of this thesis is to analyse, design and implement BWC as part of the Small Compression Toolkit (SCT for short) library. The SCT library is developed at the Faculty of Information Technology at Czech Technical University in Prague and contains multiple compression algorithms[3]. The programming language of the library is Java. After the integration to the SCT library is done, the next challenge will be performance testing of this implementation. The BWC implementation will be tested on multiple corpora and time, and the compression ratio will be measured.

Since BWC is not strictly defined and refers to a wide group of algorithms based on BWT, it was decided that SCT library will be enriched by its most common version. This version consists of BWT, MTF, RLE and the entropy coder. Huffman and Arithmetic coders were chosen for this task.

Thesis structure overview

This section briefly explains what is covered in particular chapters and what information can be found.

Chapter 1 is devoted to defining basic notions important for further chapters.

Chapter 2 analyses and explains the algorithms implemented in this thesis. Each algorithm is accompanied by its pseudocode and an example of use.

Chapter 3 is dedicated to the implementation of algorithms mentioned in Chapter 2. Each implemented algorithm is referenced to its pseudocode, and UML class diagram is shown.

Chapter 4 focuses on performance testing. In this chapter, it is shown what compression ratios can be achieved on real data.

Chapter 1

Basic notions

This chapter is dedicated to defining and describing basic terms which are relevant to the topic of this thesis. Most of the definitions in this chapter are referencing to the book *The Data Compression Book* written by *Mark Nelson*[4].

1.1 Symbol

Symbol is an element from the alphabet. In the scope of this thesis, one symbol in equal to one byte. Since the size of one byte is eight bits, a symbol has 256 values ranging from 00000000 to 11111111.

1.2 Alphabet

Alphabet is a finite set of distinguishable symbols. In the scope of this thesis, the alphabet of 256 symbols will be used. This is equal to the Extended ASCII table[5]. It is assumed that after applying lexicographical sort, values of the symbols from the alphabet will be sorted from 00000000 to 11111111.

1.3 String

String is a finite sequence of symbols from the alphabet. In the scope of this thesis, string refers to a finite sequence of bytes.

1.4 Codeword

Codeword is a sequence of bits. In other words, a codeword is a string over the binary alphabet.

1.5 Code

Code is a system of rules to convert information, such as a symbol or string, into another form or representation. Code substitutes string with codeword from the binary alphabet for the purpose of converting the original information.

1.6 Data compression

Data compression or *encoding* is a process of transforming the input string to the output string of different format, which in most cases has a shorter length. This can be very useful because it reduces the resources required to store and transmit the data. Data compression can be lossless or lossy. Lossless compression allows the original data to be entirely reconstructed from the compressed data, while lossy compression reduces the size of compressed data by removing unnecessary or less important information[6]. Data compression is subject to a space-time complexity trade-off, meaning it involves trade-offs among various factors, such as the degree of compression, time complexity and the computational resources required to compress and decompress the data[7].

1.7 Data decompression

Data decompression or *decoding* is the action of reversing data compression which transforms compressed data to their original uncompressed form. When using lossy compression, it is not possible to entirely reconstruct original data since some information has been omitted while doing data compression.

1.8 Compression algorithm

Compression algorithm or *encoding algorithm* is a finite sequence of steps needed for data compression[6].

1.9 Decompression algorithm

Decompression algorithm or *decoding algorithm* is a finite sequence of steps needed for data decompression[6].

1.10 Compression method

Compression method is a common name used to label a particular compression algorithm together with its related decompression algorithm[6].

1.11 Model

Model is an internal structure used by compression method which holds information about currently processed input. In order for the decompression algorithm to be successful, it has to use the same model as the compression algorithm. Otherwise, it won't be able to output the same string as was the input for the compression algorithm.

1.12 Adaptive compression method

Adaptive compression method adapts data model used during compression in accordance with the compressed data. The compressed data has not to contain the data model. The decoder creates the same model as the encoder. Both the encoder and decoder begin with a trivial model, yielding poor compression of initial data, but as they learn more about the data, performance improves. The encoder compresses the next data unit first and then adapts the model in accordance with this unit. This order enables the decoder to create an identical model. If the encoder changes the model before it compresses the unit, the decoder cannot decompress the data because the next unit is compressed using an unknown data model. Many of commonly used compression methods are adaptive[8].

1.13 Compression ratio

Compression ratio is the amount of data reduction gained by the compression algorithm. This compression ratio is a ratio of the length of compressed data to the original size of data (Formula 1.1)[9].

$$Compression\ ratio = \frac{Length\ of\ compressed\ data}{Length\ of\ original\ data} \tag{1.1}$$

For instance, the compression ratio of 0.75 means that data compression was able to shrink the compressed file size to 75% of its original size. If the result of this equation is greater than 1, it means compression of the input data resulted in negative compression.

1.14 Entropy

Entropy in data compression denote the randomness of the data that are subject to the compression with the compression algorithm. The more the entropy, the lesser the compression ratio. That means the more random the data are, the lesser you can compress it[10].

1.15 Redundancy

Redundancy is a fraction of data which is unnecessary and hence repetitive in the sense that if it were missing, the data would still be essentially complete, or at least could be completed[11]. Redundancy is related to the extent to which it is possible to compress the data. What lossless data compression does is reduce the number of bits used to encode a message by identifying and eliminating statistical redundancy. Data compression is a great way to reduce or eliminate unwanted redundancy[12].

1.16 Corpus

Corpus is a collection of text and binary data files, commonly used for comparing data compression algorithms[13]. Corpora are intended for use as a benchmark for testing lossless data compression.

Chapter 2

Analysis

This chapter is partly dedicated to definition and analysis of algorithms used in BWC, namely BWT, MTF, RLE, AC and HC. It is essential to examine and explore these algorithms in order for later implementation to be successful and effective.

After the part dedicated to algorithms, special attention is given to the relation between BWC and bzip algorithms which are widely used compression algorithms vastly related to the topic of this thesis.

Another part of this chapter focuses on specifying functional and non-functional requirements resulting from the assignment of this thesis. Focus is going to be given on application requirements and explain what is required and expected from the final product.

Last part of this chapter is committed to related work associated with the topic of this thesis.

2.1 Algorithms

2.1.1 Burrows-Wheeler transform

The BWT, or also called block sorting, is an auxiliary algorithm used in data compression techniques described by Michael Burrows and David Wheeler in 1994. It doesn't compress the data on its own but rather rearranges the symbols from the input so that that there are lots of clusters with repeated symbols, but in such a way that it is still possible to recover the original input[14]. The transform is done by sorting all the cyclic rotations of input in lexicographic order and by extracting the last column and the index of the original string in the table of sorted permutations of the input string. The BWT requires that additional information is stored, therefore making the transformed data slightly larger than its original form. However, that is only a small space-time tradeoff for improving efficiency and performance of further compression techniques. The BWT is useful for following compression

methods since it tends to be easier and faster to compress a string that has runs of repeated symbols by techniques such as MTF or RLE. This characteristic of the BWT has another side effect. It means that, in general, the bigger the block size, the better the compression ratio because bigger blocks will generate longer runs of repeated symbols, leading to the improved compression ratio of the following compression methods[15]. The obvious downsides to large blocks are that blocks must be stored somewhere, so you need enough memory to hold the block, and it's cyclic rotations. The cyclic rotations are also sorted, and sorting is not a linear operation. Thus larger blocks will also take more time. Potential optimizations to overcome these issues will be explained later.

The BWT works by firstly making all cyclic rotations of the input string and numbering them. Then, the lexicographical sort is applied to the resulting phrases, and the number of the line containing the input string is remembered. Now, last symbols from lexicographically sorted phrases are concatenated together to create the output string, and by appending remembered index, the output of the BWT is created[8]. The BWT algorithm is described in Algorithm 1.

Algorithm 1 BWT encode

```
1: procedure BWT(input)
2:     tab ← table of all cyclic rotations of input
3:     lexicographically sort cyclic rotations in tab
4:     idx ← the number of the line containing input in sorted tab
5:     str ← last column of tab
6:     output str, idx
7: end procedure
```

Time and space complexity highly depends on several optimizations which can make the BWT more time and space efficient. Algorithm 1 requires to hold a table of all cyclic rotations which leads to polynomial space complexity of $\mathcal{O}(n^2)$, where n is the length of the input string to be encoded. However, there is no need to hold the whole table, and every cyclic rotation can be represented as an index to the input string. Therefore, we reduce space complexity to $\mathcal{O}(n)$, where n is the length of the input string to be encoded. Time complexity highly depends on the chosen sorting algorithm. As Burrows and Wheeler suggest, time complexity can be reduced from $\mathcal{O}(n * \log n)$ to $\mathcal{O}(n)$, where n is the length of the input string to be encoded. This improvement can be achieved by using suffix trees and suffix arrays for the sorting procedure. It is done by building a suffix tree, which then can be walked in lexicographical order to recover the sorted suffix array in linear time[1].

The inverse BWT transform is a little bit complicated. It is possible to reconstruct the original table of all cyclic rotations used in the encoding. In the beginning, take the input string and insert this string as a column to the table from the left and lexicographically sort rows of the table. These two

steps are repeated as many times as there are symbols in the input string. Since the input to the inverse BWT also includes the index of the output string in the table of sorted permutations, we can simply retrieve this string by looking at the row in the restored table pointed by this index. The inverse BWT algorithm is described in Algorithm 2.

Algorithm 2 BWT decode

```
1: procedure BWT(input, idx)
2:     tab ← empty table
3:     len ← length of input
4:     for i ← 0 to len − 1 do
5:         insert input as a column to tab from the left
6:         lexicographically sort rows of tab
7:     end for
8:     output row from tab on idx position
9: end procedure
```

However, this version of the inverse BWT algorithm is highly ineffective. To determine the position of a symbol in the output given its place in the input, we can use the knowledge about the fact that the output is sorted. Each symbol will show up in the output in position, where the position is the sum of all symbols in the input string that precedes symbol in the alphabet, plus the count of all occurrences of symbol previously in the input string. First, we calculate the running totals for all the symbols in the alphabet. That satisfies the first part of the equation needed to determine where each symbol will go in the output string. When the running totals array is filled in, we have half the information needed to position each symbol. The next piece of information is the number of the same symbol that appears before this symbol in the input string. We keep track of that information in the separate array as we go. By adding those two numbers together, we get the destination of each symbol, and that allows us to fill in all the positions in the transformation vector. Once the transformation vector is in place, writing the output string is just a matter of following the indices[15]. Improved version of the inverse BWT algorithm is described in Algorithm 3. This version takes the same transformed string as Algorithm 2, but instead of the index of the output string in the table of sorted permutations, we only need to pass an index to the first symbol of the original input data before BWT took place.

Calculating the running totals for all the symbols in the alphabet, finding the number of the same symbol that appears before the current symbol in the input string and filling the transformation vector can all be done in linear time and space. Thus, we can safely say that time, and space complexity of the inverse BWT algorithm presented in Algorithm 3 is $\mathcal{O}(n)$, where n is the length of the input string to be decoded.

An example will shed some light on how the BWT works. To make this

Algorithm 3 BWT decode

```
1: procedure BWT(input, idx)
2:     len ← length of input
3:     CA ← count array for symbols from alphabet
4:     RT ← running total
5:     TV ← transformation vector
6:     FillCA(CA, input)
7:     FillRT(RT, CA, input)
8:     FillTV(TV, RT, CA, input)
9:     for i ← 0 to len − 1 do
10:        output input[idx]
11:        idx ← TV[idx]
12:    end for
13: end procedure
14: procedure FillCA(CA, input)
15:    CA ← initialised to 0
16:    len ← length of input
17:    for i ← 0 to len − 1 do
18:        j ← position of input[i] in alphabet
19:        CA[j] ← CA[j] + 1
20:    end for
21: end procedure
22: procedure FillRT(RT, CA, input)
23:    sum ← 0
24:    k ← number of symbols in alphabet
25:    for i ← 0 to k − 1 do
26:        RT[i] ← sum
27:        sum ← sum + CA[i]
28:        CA[i] ← 0
29:    end for
30: end procedure
31: procedure FillTV(TV, RT, CA, input)
32:    len ← length of input
33:    for i ← 0 to len − 1 do
34:        j ← position of input[i] in alphabet
35:        TV[CA[j] + RT[j]] ← i
36:        CA[j] ← CA[j] + 1
37:    end for
38: end procedure
```

example simpler, the demonstration of the RLE algorithm will be shown using the English alphabet consisting of 26 symbols instead of using the alphabet of size 256, which the implementation will be later working with. An example of encoding with the input string $S = mississippi$ is shown in Table 2.1 using Algorithm 1.

Table 2.1: Example of the BWT algorithm

Index	**All cyclic rotations**	**Lexicographically sorted rotations**
0	mississippi	imississip**p**
1	ississippim	ippimissis**s**
2	ssissippimi	issippimis**s**
3	sissippimis	ississippi**m**
4	issippimiss	*mississipp**i***
5	ssippimissi	pimississi**p**
6	sippimissis	ppimississ**i**
7	ippimississ	sippimissi**s**
8	ppimississi	sissippimi**s**
9	pimississip	ssippimiss**i**
10	imississipp	ssissippimi**i**

After using Algorithm 1 on the input string $S = mississippi$, we get a pair of string and value $S' = (pssmipissii, 4)$. This pair is used as an input for Algorithm 2. But since Algorithm 2 is highly ineffective, example of decoding is shown using Algorithm 3. However, Algorithm 3 operates with a pair consisting of encoded string and the index to the first symbol of the original input data before BWT took place. This means using pair $S' = (pssmipissii, 3)$ which is gathered from Table 2.1. An example of decoding using Algorithm 3 with the input pair $S' = (pssmipissii, 3)$ is as follows: symbols counts are in Table 2.2, symbols running totals are in Table 2.3 and transformation vector is shown in Table 2.4.

Table 2.2: Computed symbols counts for a given example using Algorithm 3. We have alphabet consisting of 4 symbols i, m, p, s and input string $pssmipissii$.

Index	**Symbol**	**Symbols count value**
0	i	CA[0] = 4
1	m	CA[1] = 1
2	p	CA[2] = 2
3	s	CA[3] = 4

Now, using Table 2.3 and Algorithm 3, we can easily assemble decoded string $S'' = mississippi$. And because $S = S''$, we can safely say that we have successfully shown the use of the BWT algorithm and its inverse version.

Table 2.3: Computed running totals for a given example using Algorithm 3. We mustn't forget to reset symbol count for every symbol after this step.

Index	Symbol	Symbols running total value
0	i	RT[0] = 0
1	m	RT[1] = 4
2	p	RT[2] = 5
3	s	RT[3] = 7

Table 2.4: Computed transformation vector for a given example using Algorithm 3.

Index	Value
0	TV[0] = 4
1	TV[1] = 6
2	TV[2] = 9
3	TV[3] = 10
4	TV[4] = 3
5	TV[5] = 0
6	TV[6] = 5
7	TV[7] = 1
8	TV[8] = 2
9	TV[9] = 7
10	TV[10] = 8

2.1.2 Move-to-front transform

The MTF is an algorithm used to improve the performance of techniques of compression by decreasing information entropy[16]. It doesn't compress data, but instead transforms input data to help following algorithms, such as RLE, with more efficient compression[8]. This algorithm, as the name suggests, uses a list of possible symbols and modifies this list at every cycle (moving one symbol, the last used). Long sequences of identical symbols are replaced by as many zeros, whereas when a symbol that has not used in a long time, it is replaced with a large number. Thus at the end, the data is transformed into a sequence of integers; if the data exhibits a lot of local correlations, then these integers tend to be small. This algorithm is designed to improve the performance of entropy encoding techniques of compression[17][18]. The MTF algorithm is described in Algorithm 4.

The time complexity of the MTF algorithm is $\mathcal{O}(nk)$, where n is the length of the input string to be encoded, and k is the number of symbols in the alphabet. This is because, for every symbol from the input string, we need to find its position in the list of all the symbols from the alphabet and then reorder alphabet order. Since we only need to hold current order of all the

Algorithm 4 MTF encode

```
1: procedure MTF(input)
2:     alphOrder ← default symbol order in alphabet
3:     while input ≠ EOF do
4:         currSymbol ← read symbol from input
5:         idx ← position of currSymbol in alphOrder
6:         output idx
7:         MoveToFront(currSymbol, alphOrder)
8:     end while
9: end procedure
```

symbols from the alphabet, we can easily say that space complexity of the MTF algorithm is $\mathcal{O}(k)$, where k is the number of symbols in the alphabet[19].

It is easy to see that the MTF is reversible. Simply maintain the same list of all the symbols from the alphabet and decode by reading each index from the encoded string, output the symbol at that index from the list and move the symbol to the front of the list. The inverse MTF algorithm is described in Algorithm 5.

Algorithm 5 MTF decode

```
1: procedure IMTF(input)
2:     alphOrder ← default symbol order in alphabet
3:     while input ≠ EOF do
4:         currIdx ← read index from input
5:         symbol ← symbol at currIdx index in alphOrder
6:         output symbol
7:         MoveToFront(symbol, alphOrder)
8:     end while
9: end procedure
```

Since both the MTF algorithm and its inverse version execute the same instruction and use the same data structures, time and space complexity of the inverse MTF algorithm are the same as the original MTF transform.

An example will shed some light on how the transform works. For the sake of simplicity, the demonstration of the MTF algorithm will be shown using the English alphabet consisting of 26 symbols instead of using the alphabet of size 256, which the implementation will be later working with. An example of encoding with the input string $S = mississippi$ is shown in Table 2.5 using Algorithm 4.

After applying Algorithm 4 on the input string $S = mississippi$, we get the output sequence of integers $S' = 12, 9, 18, 0, 1, 1, 0, 1, 16, 0, 1$. This sequence is used as an input for the inverse MTF algorithm. An example of

Table 2.5: Example of the MTF algorithm

Input	Output	Current order of the symbols in alphabet
mississippi	12	abcdefghijklmnopqrstuvwxyz
ississippi	9	mabcdefghijklnopqrstuvwxyz
ssissippi	18	imabcdefghjklnopqrstuvwxyz
sissippi	0	simabcdefghjklnopqrtuvwxyz
issippi	1	simabcdefghjklnopqrtuvwxyz
ssippi	1	ismabcdefghjklnopqrtuvwxyz
sippi	0	simabcdefghjklnopqrtuvwxyz
ippi	1	simabcdefghjklnopqrtuvwxyz
ppi	16	ismabcdefghjklnopqrtuvwxyz
pi	0	pismabcdefghjklnoqrtuvwxyz
i	1	pismabcdefghjklnoqrtuvwxyz

decoding with the input sequence $S' = 12, 9, 18, 0, 1, 1, 0, 1, 16, 0, 1$ is shown in Table 2.6 using Algorithm 5.

Table 2.6: Example of the inverse MTF algorithm

Input	Output	Current order of the symbols in alphabet
12, 9, 18, 0, 1, 1, 0, 1, 16, 0, 1	m	abcdefghijklmnopqrstuvwxyz
9, 18, 0, 1, 1, 0, 1, 16, 0, 1	i	mabcdefghijklnopqrstuvwxyz
18, 0, 1, 1, 0, 1, 16, 0, 1	s	imabcdefghjklnopqrstuvwxyz
0, 1, 1, 0, 1, 16, 0, 1	s	simabcdefghjklnopqrtuvwxyz
1, 1, 0, 1, 16, 0, 1	i	simabcdefghjklnopqrtuvwxyz
1, 0, 1, 16, 0, 1	s	ismabcdefghjklnopqrtuvwxyz
0, 1, 16, 0, 1	s	simabcdefghjklnopqrtuvwxyz
1, 16, 0, 1	i	simabcdefghjklnopqrtuvwxyz
16, 0, 1	p	ismabcdefghjklnopqrtuvwxyz
0, 1	p	pismabcdefghjklnoqrtuvwxyz
1	i	pismabcdefghjklnoqrtuvwxyz

After applying Algorithm 5 on the input sequence of integers $S' = 12, 9, 18, 0, 1, 1, 0, 1, 16, 0, 1$, we get the output string $S'' = mississippi$. Since $S = S''$, we have successfully shown use of the MTF algorithm and its inverse version.

2.1.3 Run-length encoding

The RLE is a simple and popular data compression algorithm that offers great compression ratios with data that contain lots of redundant symbols[20]. The

output of the MTF algorithm is a great example of such data with lower entropy and high redundancy and thus, making it suitable for use with the RLE algorithm[16]. This algorithm converts consecutive long sequences of identical symbol runs into a code consisting of the symbol and the number marking the length of the run. The longer the run, the better the compression ratio[21]. The RLE algorithm is described in Algorithm 6.

Algorithm 6 RLE encode

```
1:  procedure RLE(input)
2:      lastSymbol ← last symbol read
3:      runLen ← 0
4:      while input ≠ EOF do
5:          currSymbol ← read symbol from input
6:          if currSymbol = lastSymbol then
7:              runLen ← runLen + 1
8:          else
9:              output currSymbol, runLen
10:             runLen ← 0
11:         end if
12:         lastSymbol ← currSymbol
13:     end while
14:     output currSymbol, runLen
15: end procedure
```

The time complexity of the RLE algorithm is $\mathcal{O}(n)$, where n is the length of the input string to be encoded. This is because we need to iterate through every symbol from the input string and as we access the symbol, we either increase the symbol run length or output the symbol followed with its run length. Since we only need to maintain a few additional variables, we can safely say that the space complexity of the RLE algorithm is $\mathcal{O}(1)$, meaning constant memory usage[21].

It is undeniable to notice that the RLE is easily reversible, considering we only need to expand symbols run lengths. We iterate through the encoded string, read symbols followed by their run lengths and output symbol n times, where n is the symbols run length. The inverse RLE algorithm is described in Algorithm 7.

Same as the RLE algorithm, its inverse version also needs to iterate through entire string, meaning time complexity is $\mathcal{O}(n)$, where n is the length of the input string to be decoded. While decoding the input string with the inverse RLE algorithm, we only need to hold few additional variables, and therefore we can say that space complexity of the inverse RLE algorithm is $\mathcal{O}(1)$, meaning constant memory usage[21].

For the sake of simplicity, the following demonstration of the RLE algorithm will be shown using the English alphabet consisting of 26 symbols

Algorithm 7 RLE decode

```
1: procedure iRLE(input)
2:     while input ≠ EOF do
3:         currSymbol ← read symbol from input
4:         runLen ← read symbols run length from input
5:         while runLen ≥ 0 do
6:             output currSymbol
7:         end while
8:     end while
9: end procedure
```

instead of using the alphabet of size 256, which the implementation will be later working with. An example of encoding with the input string $S = aabbbbbcdddeeeeff$ is shown in Table 2.7 using Algorithm 6.

Table 2.7: Example of the RLE algorithm

Input	Output
aabbbbbcdddeeeeff	a2
bbbbbcdddeeeeff	b5
cdddeeeeff	c1
dddeeeeff	d3
eeeeff	e4
ff	f2

After applying Algorithm 6 on the input string $S = aabbbbbcdddeeeeff$, we get the output string $S' = a2b5c1d3e4f2$. We can see that the RLE algorithm was able to shrink string size from 17 symbols to just 12, however for symbol c, the size was doubled from 1 symbol in S, to 2 symbols in S'. It is obvious from this example why is a specific type of data so important for great compression ratio and why data encoded with the MTF algorithm are such a great candidate for this role. String S' can be now used as an input for the inverse RLE algorithm. An example of decoding with the input string $S' = a2b5c1d3e4f2$ is shown in Table 2.8 using Algorithm 7.

After applying Algorithm 7 on the input string $S' = a2b5c1d3e4f2$, we get the output string $S'' = aabbbbbcdddeeeeff$. We can see that $S = S''$, therefore we have successfully shown use of the RLE algorithm and its inverse version.

Table 2.8: Example of the inverse RLE algorithm

Input	Output
a2b5c1d3e4f2	aa
b5c1d3e4f2	bbbbb
c1d3e4f2	c
d3e4f2	ddd
e4f2	eeee
f2	ff

2.1.4 Adaptive Arithmetic coding

Arithmetic coding is a form of entropy encoding used in lossless data compression, that allows each symbol to be coded with a non-whole number of bits (when averaged over the entire data), thus improving compression ratio[22]. Arithmetic coding represents the current data as a range, defined by lower and upper bounds and encodes the entire data into a single number from this interval, an arbitrary-precision fraction n where $0 \leq n < 1$[23]. Starting with the interval $\langle 1, 0)$, each interval is divided into several subintervals, which sizes are proportional to the current probability of the corresponding symbols of the alphabet. The subinterval from the currently coded symbol is then taken as the interval for the next symbol. The output is a number from the interval of the last symbol.

One of the most important advantages of AC is its flexibility and the fact that it can be used in conjunction with any model that provides a sequence of event probabilities. This advantage is quite important because great compression ratios can be obtained only through the use of sophisticated models of the input data. Another significant advantage of AC is its optimality. When the probability of some single symbol is close to 1, AC does give considerably better compression ratio than other compression methods. So the bigger the probability of symbols, the more efficient the arithmetic coding is. A minor disadvantage is the need to indicate the end of the file[24].

Adaptive AC is an adaptive compression method which starts with flat probabilities of symbols and updates them after each symbol is processed, thus making it reflect the statistics of the data being compressed[22]. When the coder encodes the data, it counts the frequencies of the symbols that have occurred so far in order to obtain a model of the probabilities for the future symbols. The coder is called adaptive because the model evolves gradually while the coder scans its input[25]. The Adaptive AC is described in Algorithm 8.

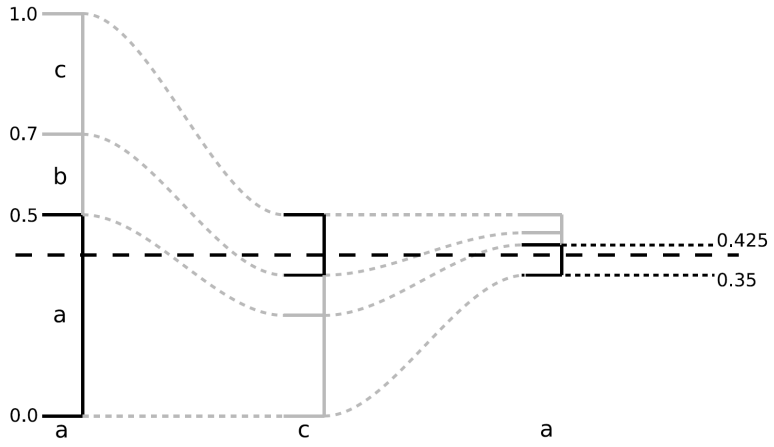
The time complexity of the Adaptive AC encode algorithm is $\mathcal{O}(k + n)$, where n is the length of the input string to be encoded, and k is a number of different input symbols. Space complexity depends on a number of different input symbols; therefore we get $\mathcal{O}(k)$, where k is the number of symbols in

Algorithm 8 Adaptive AC encode

```
1: procedure AACEncode(input)
2:     n ← size of the alphabet
3:     prob ← set probabilities of all symbols to 1/n
4:     I ← interval ⟨1, 0)
5:     divide I according to the probabilities of all symbols
6:     while input ≠ EOF do
7:         currSymbol ← read symbol from input
8:         I ← subinterval I corresponding to currSymbol
9:         prob ← update the probabilities of all symbols
10:        divide I according to the probabilities of all symbols
11:    end while
12:    output arbitrary-precision fraction from I
13: end procedure
```

the alphabet[8]. An example of probabilities and interval division is shown in Figure 2.1.

Figure 2.1: Example of Arithmetic coding with the input string $S = aca$ and alphabet consisting of symbols a, b, c with probabilities $P(a) = 0.5, P(b) = 0.2, P(c) = 0.3$[24].



2.1.5 Adaptive Huffman coding

Huffman coding is an algorithm of entropy coder based on the probabilities of the data occurring in the sequence. HC encodes symbols into variable length, depending on the probability of each symbol. The symbols which occur more frequently will need fewer bits than symbols with less frequency[17]. First, the algorithm counts frequencies (probabilities) of all single symbols in the input alphabet. According to these frequencies, it generates a binary tree, which

has edges with value 0 or 1 and in its leaf nodes there are symbols of input alphabet. Likewise, it holds that the highest is a probability of symbol; the nearer is the appearance of a node to the root of the tree. Then these symbols are replaced with binary code, which corresponds to concatenation of values of edges, which we pass on the way from the root to the given leaf node. Since one of the values 0 or 1 always appears on edges leading to the right successor (the other to the left successor) and symbols of input alphabet are located only in leaf nodes, the resulting code is a prefix. It is necessary to attach this tree or at least the information about frequencies of appearance of single symbols in input data to the resulting sequence. Then the decompression is available[8].

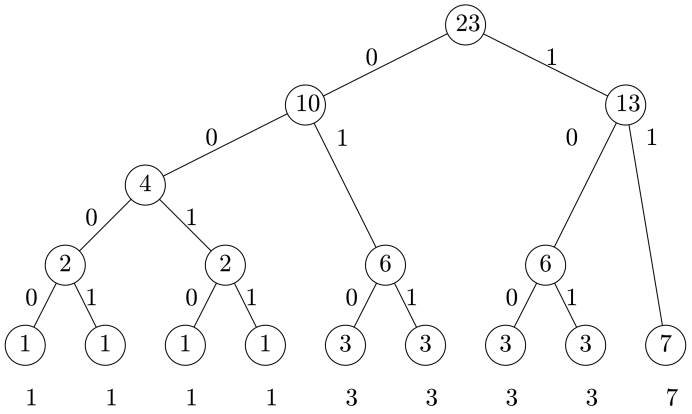
Adaptive HC is an adaptive compression method which involves calculating the probabilities dynamically based on recent actual frequencies in the sequence of source symbols, and changing the coding tree structure to match the updated probability estimates. The coder is called adaptive because the binary tree is changing simultaneously with processed data in order for the coder to remain optimal for the current probability estimates. Since it permits building the code as the symbols are being transmitted, having no initial knowledge of source distribution, only one pass over the data is required. Another benefit of the one-pass procedure is that the source can be encoded in real time. The Adaptive HC is described in Algorithm 9[8].

In the worst case (coding tree needs to be changed after each single symbol in the input string on all levels) is estimated time complexity $\mathcal{O}(n * \log k)$, where n is the length of the input string to be encoded, and k is a number of different input symbols in the alphabet. The coding tree has k leaf nodes, and depth is roughly $\log k$ (though the tree needn't be balanced, this could do as an estimate). Therefore its size is about $2 * k$ and we can tell that the space complexity is $\mathcal{O}(k)$[8]. An example of the created coding tree from given symbols and frequencies is shown Figure 2.2.

Algorithm 9 Adaptive HC encode

```
1: procedure AHCEncode(input)
2:     ZERO ← create node ZERO
3:     while input ≠ EOF do
4:         currSymbol ← read symbol from input
5:         if first read of currSymbol then
6:             output ZERO, currSymbol
7:             newNode ← new node with next nodes ZERO and new node currSymbol
8:             UpdateTree(newNode)
9:         else
10:            output currSymbol
11:            UpdateTree(currSymbol)
12:        end if
13:    end while
14: end procedure
15: procedure UpdateTree(inNode)
16:    while inNode ≠ ROOT do
17:        if exist node N with same value and greater order then
18:            change inNode and N
19:        end if
20:        inNode ← increment value
21:        inNode ← parent(inNode)
22:    end while
23:    inNode ← increment value, update leaf nodes
24: end procedure
```

Figure 2.2: Example of Huffman coding tree with the alphabet of size 9 and symbols input frequencies $\{1, 1, 1, 1, 3, 3, 3, 3, 7\}$. Assigned codeword for every symbol can be easily gathered by following the path from the root of the Huffman coding tree to the desired leaf[26].



2.2 Burrows-Wheeler transform and bzip algorithms

Bzip and its more popular ancestor bzip2 are open source lossless data compression programs which are based on Burrows-Wheeler transform. These algorithms use the BWT, MTF and RLE algorithms as the main parts of their schemes. Excluding special optimization steps, these algorithms embrace the same compression and decompression schemes as a program created as a part of this thesis. Original bzip, made first public in 1996, used AC but due to the acquisition of patents by IBM switched to HC and thus, bzip2 was created[8]. As per the information published by the developer, bzip2 is capable of compressing files down to 15% or 10% of other available techniques and operates at twice the compression speed and six times the decompression speed than another popular compression algorithm, gzip. The bzip2 program processes data in blocks ranging from 100 to 900 kilobytes in size with the default block size of 900 kilobytes. Performance of bzip2 is asymmetric, as decompression is relatively quick[2].

2.3 Non-functional requirements

Non-functional requirements refer to constraints and behavioural properties of the system. They specify criteria that can be used to judge the operation of a system, rather than specific behaviours. A typical non-functional requirement contains a unique name and number and a brief summary. This information is used to better understand why the requirement is needed and can be used to track the requirement through the development of the system.

- **N1 SCT library -** Implementation has to be done as a part of the Small Compression Toolkit library. The SCT library is developed at th Faculty of Information Technology at Czech Technical University in Prague and already contains multiple compression algorithms[3].
- **N2 Java programming language -** Implementation has to be done in Java programming language which is the programming language used to develop SCT library.
- **N3 Integrability -** Implementation has to provide an interface which makes it possible to integrate new components later.
- **N4 Documentation -** Implementation has to be appropriately and sufficiently documented, and this documentation has to be compliant with documentation standards for Java programming language.
- **N5 Readability -** The created code has to be easily readable and has to follow Java programming language conventions.

2.4 Functional requirements

Functional requirements specify a function that a system or system component must be able to perform. They refer to services that the system should provide. A typical functional requirement contains a unique name and number and a brief summary. This information is used to better understand why the requirement is needed and can be used to track the requirement through the development of the system.

- **F1 BWT implementation -** Implement Burrows-Wheeler transform. This implementation must be compliant with standards set for method interfaces in the SCT library.
- **F2 MTF implementation -** Implement Move-to-front transform. This implementation must be compliant with standards set for method interfaces in the SCT library.
- **F3 RLE implementation -** Implement Run-length encoding. This implementation must be compliant with standards set for method interfaces in the SCT library.
- **F4 AC implementation -** Implement Arithmetic coding. This implementation must be compliant with standards set for method interfaces in the SCT library.
- **F5 HC implementation -** Implement Huffman coding. This implementation must be compliant with standards set for method interfaces in the SCT library.
- **F6 Client -** Implement client which can be used to run the program from the command-line interface. The functionality of this client can be controlled by parameters.

2.5 Related work

As already mentioned, bzip and bzip2 heavily rely on the BWT, MTF and RLE algorithms. Apart from these programs, there are many other compression tools containing the BWT algorithm, such as ZZip or ZPAQ. Another usage of the BWT algorithm can be found in the sequence alignment and sequence analysis in bioinformatics. Bowtie is an ultrafast, memory-efficient alignment program for aligning short DNA sequence reads to large genomes. For the human genome, the BWT indexing allows Bowtie to align more than 25 million reads per CPU hour with a memory footprint of approximately 1.3 gigabytes. Bowtie extends previous Burrows-Wheeler techniques with a novel quality-aware backtracking algorithm that permits mismatches. Multiple processor cores can be used simultaneously to achieve even greater alignment speeds[27].

2.6 Associated theses

There have already been four theses dedicated to implementing compression algorithms as a part of the SCT library. Thesis *Implementation of the ACB compression method improvements in the Java language* written by Jiří Bican laid the foundation for SCT library and added the ACB algorithm as the first of many to come[24]. Jakub Novák with his thesis *Implementace kompresní metody DCA v jazyce Java* contributed by implementing the DCA algorithm[28]. The LZ77, LZ78 and LZW algorithm were implemented by Ladislav Zemek in his thesis *Implementace kompresních metod LZ77, LZ78, LZW v jazyce Java*[29]. The last contribution was made by Ján Bobot and his thesis *Implementace kompresních metod LZY, LZMW a LZAP v jazyce Java* when he implemented the LZY, LZMW and LZAP algorithms[30]. Apart from these algorithms, the SCT library also involves tools for manipulation with input and output files.

Chapter 3

Design and implementation

This chapter is dedicated to the implementation of required functionality with special attention given to meet functional and non-functional requirements. Assignment of this thesis was the implementation of the BWC compression method and its variants in Java programming language. This chapter deals with this process and explains the steps needed to implement it.

3.1 Overview

The functionality was implemented as a part of the SCT library which is developed in Java programming language. Since the SCT library is already using Git, the choice of a version control system (or VSC for short) used to track changes in source code during development is clear and obvious[3]. A handy tool for software development is an integrated development environment (IDE for short) which can provide many useful utilities to make work more productive, such as integrated VCS, UML class diagram generator or built-in build automation tool like Apache Maven. IDE chosen for this task was IntelliJ IDEA developed by JetBrains.

3.2 External libraries

Two main libraries are used to provide support for the main functionality. These libraries don't affect dataflow itself but rather help with support processes such as logging or parsing the input from the command line.

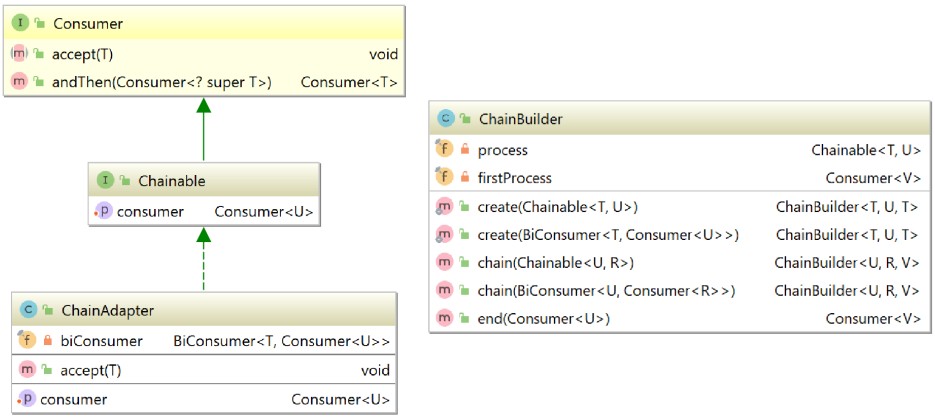
- **Apache Commons CLI -** This library provides an API for parsing command line options passed to program. It's also able to print help messages detailing the options available for a command line tool.

- **Apache Log4j 2 -** This utility is one of the most widely used logging frameworks used in the Java programming language. It allows fast and efficient management of the logging files.

3.3 Dataflow in application

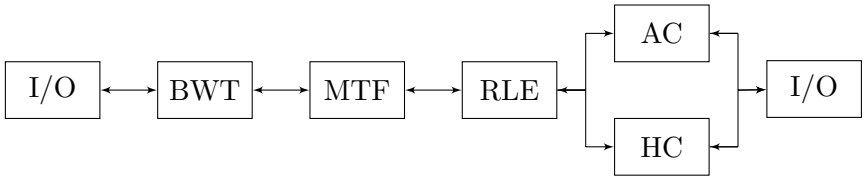
The critical task to do is to ensure that data can flow freely and no component creates a bottleneck for data processing. The SCT library already contains tools for such a task. The component chaining is done by `ChainBuilder` class which provides methods for creating dataflow stream. This class ensures that every component is fully responsible for handling the given data. The only requirement made for a component in order to be added to the dataflow chain is that the component must take input from the preceding segment as an input and provide an output which can be processed by the following chain segment. The UML class diagram for `ChainBuilder` is shown in Figure 3.1.

Figure 3.1: UML class diagram for `ChainBuilder`.



Components implemented as a part of this thesis works with `ByteBuffer` class. `ByteBuffer` is a Java class which wraps raw data bytes and provides methods for their convenient manipulation. Since all components implemented as a part of this thesis provide the same input and output interface, they can be randomly and freely chained and exchanged in dataflow stream. This possibility creates space for other experiments and opportunities for more effective data compression. Default dataflow and component chaining used in this thesis is show in Figure 3.2. If data are being compressed, the data flows from left to right and for decompression its the other way around.

Figure 3.2: Default dataflow and component chaining.



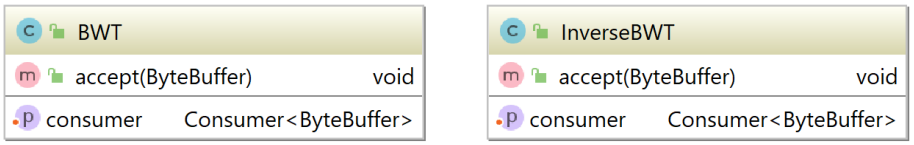
3.4 Input and output

As shown in Figure 3.2, the dataflow always starts and ends in the component responsible for file input and output. This functionality is already available in the SCT library. Class `FileIO` takes care of opening files, reading from files, writing to files and finally closing the files. `FileIO` class requires only one parameter - block size. This parameter specifies how big the chunks of data read from the input file will be. The bigger the block, the better the compression ratio but this also means higher memory usage. If a user doesn't specify the size of blocks, the default size of 5 megabytes is used. Unfortunately, the previously implemented `FileIO` class has a little bit different interface that is desired, and instead of using `ByteBuffer` class the data have to be altered before further use. However, this only means small inconvenience and a few additional lines of code.

3.5 Burrows-Wheeler transform

The BWT algorithm which encodes the date is described in Algorithm 1 and implemented in `BWT` class. For lexicographically sorting the rotations of the input string, `Arrays.sort()` with own comparator is used. This comparator is used to specify the order in which the `Arrays.sort()` sorts the data. The inverse BWT algorithm used to data decoding is described in Algorithm 3 and implemented in `InverseBWT` class. Special attention is given to properly documenting the source code to make it as readable as possible. The UML class diagram for the BWT algorithm is shown in Figure 3.3.

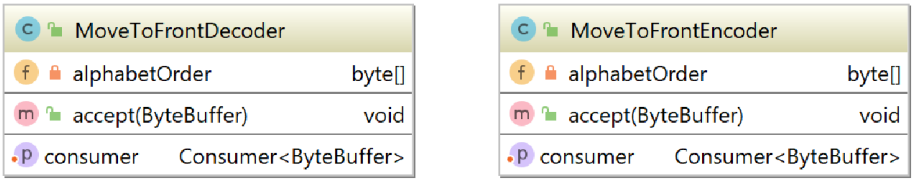
Figure 3.3: UML class diagram for the BWT algorithm.



3.6 Move-to-front transform

The MTF is a fairly simple algorithm consisting of only a few steps. Encoding with the MTF algorithm is described in Algorithm 4 and this implementation is done in `MoveToFrontEncoder` class. The inverse MTF algorithm which decodes the given data is described in Algorithm 5 and implemented in `MoveToFrontDecoder` class. Special emphasis is given to properly documenting the source code in order to make it as readable as possible. The UML class diagram for the MTF algorithm is shown in Figure 3.4.

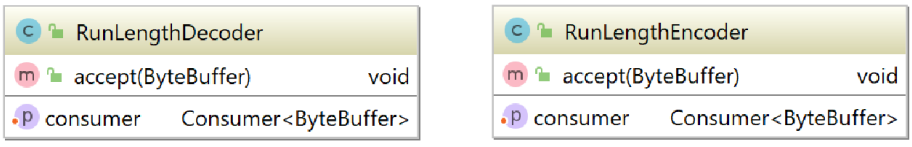
Figure 3.4: UML class diagram for the MTF algorithm.



3.7 Run-length encoding

The RLE algorithm described in Algorithm 6 was slightly modified for better optimisation. Instead of writing run length after every symbol, which would often be 1, run length is counted and written only if two or more consecutive symbols occur in the data stream. This slightly modified version of the Algorithm 6, which encodes data using the RLE algorithm, is implemented in `RunLengthEncoder` class. The inverse RLE algorithm also had to be modified to some extent. This modified version of Algorithm 7 is implemented in `RunLengthDecoder` class and is used to decode the given input. Appropriate attention is given to creating readable code. The UML class diagram for the RLE algorithm is shown in Figure 3.5.

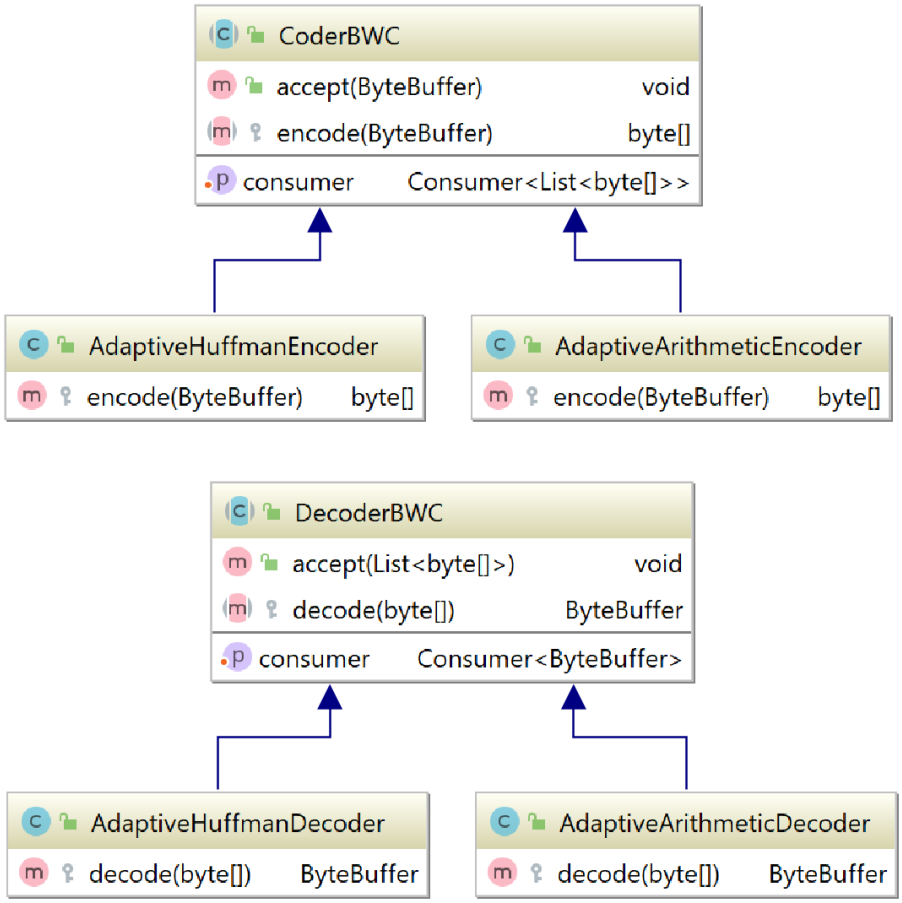
Figure 3.5: UML class diagram for the RLE algorithm.



3.8 Coder

Encoding and decoding process currently supports two entropy coders - Adaptive Arithmetic coding and Adaptive Huffman coding. Abstract classes `CoderBWC` and `DecoderBWC` serve as wrappers for these coders and every new coder added to the SCT library has to implement these abstract classes in order to be sufficient for further use in the library. Both of these classes communicate with previously implemented `FileIO` class, and thus their interface is a little bit different in order to be possible to integrate them into the library. This can be better seen in Figure 3.6.

Figure 3.6: UML class diagram for the coder.



Both Adaptive AC and Adaptive HC are outsourced by opensource library developed by Nayuki[22, 31]. Classes `AdaptiveArithmeticEncoder`, `AdaptiveArithmeticDecoder`, `AdaptiveHuffmanEncoder` and `AdaptiveHuffmanDecoder` implements already mentioned abstract classes

`CoderBWC` and `DecoderBWC` and slightly adjust data before forwarding them to the external library. To make the source code as readable as possible, special emphasis is given to properly documenting it. The UML class diagram for the coder is shown in Figure 3.6.

3.9 Client

Class `BWCClient` is an executable program and represents client which can be run from the command line. Its behaviour can be altered by console parameters. Following text summarizes the usage of implemented `BWCClient`. This help message is printed if any error occurs while parsing parameters from the command line or can be purposely printed with `-h` parameter.

```
usage: bwc.jar input output [options]
input - input file
 -b <block>                size of blocks in bytes - default
                           size is 5 MB
 -cd,--coding <arg>        coding used in BWC - arithmetic or
                           huffman (default is huffman
                           coding) [ar][hf]
 -de,--decompress          decompress input (default is to
                           compress)
 -h,--help                 print this help
 -l,--log-level <level>    sets logging level of the
                           application
 -m,--measure              measured time and compression
                           ratio and print
```

Chapter 4

Measurements and results

4.1 Overview

This chapter is dedicated to performance testing of the created implementation. The BWC implementation is tested on multiple corpora and time, and the compression ratio is measured. For the purpose of evaluating the capabilities of the implementation, Adaptive AC and the default block size of 5 megabytes are used. Basic overviews of the used corpora are defined in dedicated sections alongside with results of performance testing. Since many other processes can affect the performance of the used platform, the measurements were performed multiple times and a result with the lowest time consumed was used as a result of the measurement.

All measurements are done on the platform with the following configuration:

- CPU - Intel Core i7 8750H 2,2 GHz hexa-core,
- RAM - 32 GB DDR4,
- OS - Windows 10 64-bit architecture.

4.2 Calgary corpus

The Calgary corpus is a collection of text and binary data files, commonly used for comparing data compression algorithms. It was created by Ian Witten, Tim Bell and John Cleary from the University of Calgary in 1987 and was commonly used in the 1990s. In 1997 it was replaced by the Canterbury corpus, based on concerns about how representative the Calgary corpus was, but the Calgary corpus still exists for comparison and is still useful for its originally intended purpose. It contains 18 files of 9 different types with complete size 3,266,560 bytes[13]. Final results for this corpus are shown in Table 4.1.

Table 4.1: Performance testing results for the Calgary corpus.

File name	Size [B]	Compr. ratio	Compr. time [μs]	Decompr. time [μs]
calgary.tar	3266560	0.301	16408	1152
bib	111261	0.265	223	75
book1	768771	0.332	1143	388
book2	610856	0.282	930	244
geo	102400	0.621	209	83
news	377109	0.335	653	167
obj1	21504	0.507	86	36
obj2	246814	0.325	471	106
paper1	53161	0.334	129	57
paper2	82199	0.329	169	80
paper3	46526	0.366	109	59
paper4	13286	0.418	66	31
paper5	11954	0.431	59	28
paper6	38105	0.344	97	57
pic	513216	0.109	13488	98
progc	39611	0.336	100	50
progl	71646	0.236	148	80
progp	49379	0.235	145	61
trans	93695	0.205	222	70

4.3 Canterbury corpus

The Canterbury Corpus was published by Ross Arnold and Tim Bell in 1997. The aim was to replace outdated Calgary Corpus and to provide more relevant testing for new compression algorithms. The files were selected based on their ability to provide representative performance results. In its most commonly used form, the corpus consists of 11 files, selected as ”average” documents from 11 classes of documents, totalling 2,826,240 bytes[32]. Final results for this corpus are shown in Table 4.2.

Table 4.2: Performance testing results for the Canterbury corpus.

File name	**Size [B]**	**Compr. ratio**	**Compr. time [μs]**	**Decompr. time [μs]**
canterbury.tar	2826240	0.213	16063	767
alice29.txt	152089	0.309	314	90
asyoulik.txt	125179	0.344	248	101
cp.html	24603	0.328	79	40
fields.c	11150	0.297	58	27
grammar.lsp	3721	0.38	46	12
kennedy.xls	1029744	0.111	791	192
lcet10.txt	426754	0.277	715	173
plrabn12.txt	481861	0.332	830	233
ptt5	513216	0.109	13631	97
sum	38240	0.354	135	60
xargs.1	4227	0.447	48	17

4.4 Prague corpus

The Prague corpus is specific because of its diversity. In order to keep the corpus up to date, a methodology for regular updates of the corpus was designed. Being the largest from those three corpora, this corpus contains 30 files of the total size 58,265,600 bytes[33]. Final results for this corpus are shown in Table 4.3.

Table 4.3: Performance testing results for the Prague corpus.

File name	Size [B]	Compr. ratio	Compr. time [μs]	Decompr. time [μs]
prague.tar	58265600	0.574	185035	24950
abbot	349055	0.92	621	529
age	137216	0.43	395	89
bovary	2202291	0.266	3281	763
collapse	2871	0.464	42	11
compress	111646	0.186	283	82
corilis	1262483	0.491	77295	507
cyprus	555986	0.027	2710	102
drkonqi	111056	0.348	225	76
emission	2498560	0.098	14585	339
firewrks	1440054	0.96	2029	975
flower	10287665	0.393	11317	4503
gtkprint	37560	0.303	129	41
handler	11873	0.257	67	19
higrowth	129536	0.403	253	80
hungary	3705107	0.017	29317	428
libc06	48120	0.351	133	54
lusiadas	625664	0.308	4128	248
lzfindmt	22922	0.229	81	30
mailflder	43732	0.216	118	38
mirror	90968	0.402	216	73
modern	388909	0.333	695	218
nightsht	14751763	0.815	18354	8221
render	15984	0.26	62	32
thunder	3172048	0.792	4354	1951
ultima	1073079	0.666	1565	545
usstate	8251	0.278	56	24
venus	13432142	0.715	17433	7038
w01vett	1381141	0.053	3335	175
wnvcrdt	328550	0.042	933	66
xmlevent	7542	0.307	55	24

Conclusion

The goal of this thesis was to implement a set of tools for data compression as a part of the SCT library. Maximum effort was given into fulfilling this goal, and the toolkit successfully implements all of the techniques discussed in the Analysis chapter, such as BWT, MTF, RLE, AC and HC. The toolkit also fulfils all the functional requirements and satisfy all non-functional requirements. All relevant algorithms were implemented, and they underwent performance testing to test their compression and time efficiency. I hope somebody finds some uses for techniques described and implemented in this thesis since they have the potential for further use and additional improvements. Personally, I have found this topic to be very interesting, and I hope more attention would be given to it.

Future work

While working on this thesis, many new ideas for further improvements emerged. The SCT library doesn't have a unified file extension, and every implemented algorithm uses its own. This creates a significant opportunity for creating `.sct` file extension, and the required data would be written to file header which would have unified layout. Components implemented as part of this thesis provide the same input and output interface, allowing free and random component chaining. This opens a lot of options for further experiments and library extensions. However, previously implemented algorithms don't have this advantage and therefore have limited use. Changing input and output interfaces to a unified form would greatly expand their field of use.

Bibliography

[1] Burrows, M.; Wheeler, D. A Block-sorting Lossless Data Compression Algorithm. *Systems Research Center*, 1994. Available from: https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

[2] Seward, J. bzip2. *bzip2*, 1996. Available from: http://www.bzip.org/

[3] Small Compression Toolkit. 2016. Available from: https://gitlab.fit.cvut.cz/polacrad/sct

[4] Nelson, M. *The Data Compression Book*. Wiley, second edition, 1995, ISBN 1558514341.

[5] ASCII Code - The extended ASCII table. 2019. Available from: https://www.ascii-code.com/

[6] Data compression. 2019. Available from: https://en.wikipedia.org/wiki/Data_compression

[7] Mittal, S.; Vetter, J. A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems. *IEEE Xplore*, 2015. Available from: https://ieeexplore.ieee.org/document/7110612?arnumber=7110612

[8] Data Compression Applets Library. 2019. Available from: http://www.stringology.org/DataCompression/content.html

[9] Salomon, D. *A Concise Introduction to Data Compression*. Springer London, illustrated edition, 2008, ISBN 1848000715.

[10] Fossum, V. Entropy, Compression, and Information Content. *Information Sciences Institute*, 2013.

[11] Kortman, C. M. Data compression by redundancy reduction. *IEEE Spectrum*, volume 4, 1967: pp. 133–139.

[12] Dollors. What is Information? Part 2a - Information Theory. *Cracking the Nutshell*, 2013. Available from: https://crackingthenutshell.org/what-is-information-part-2a-information-theory/

[13] Calgary corpus. 2019. Available from: https://en.wikipedia.org/wiki/Calgary_corpus

[14] Wayne, K. Burrows-Wheeler Data Compression Algorithm. *Princeton University COS 226*, 2004. Available from: http://www.cs.princeton.edu/courses/archive/spr07/cos226/assignments/burrows.html

[15] Nelson, M. Data Compression with the Burrows-Wheeler Transform. *Dr. Dobb's Journal*, 1996. Available from: https://marknelson.us/posts/1996/09/01/bwt.html

[16] Zalik, B.; Lukac, N. Chain code lossless compression using move-to-front transform and adaptive run-length encoding. *Sig. Proc.: Image Comm.*, volume 29, 2014: pp. 96–106.

[17] Liu, Y.; Goutte, R. Lossy and lossless spectral image compression using quaternion Fourier, Burrows-Wheeler and Move- to-Front transforms. *2012 IEEE 11th International Conference on Signal Processing*, volume 1, 2012: pp. 619–622.

[18] Ryabko, B. Y. Data compression by means of a 'book stack'. *Probl. Inf. Transm.*, volume 16, 1980: pp. 265–269.

[19] Latief, A. S.; Lawi, A. Data Compression Scheme with Composition of The Burrows-Wheeler Transform, Move-to-Front Transform, and Huffman Coding. *Information Processing Society of Japan*, 2013.

[20] RLE compression. 2019. Available from: https://www.prepressure.com/library/compression-algorithm/rle

[21] Murray, J. D.; vanRyper, W. *Encyclopedia of Graphics File Formats (2Nd Ed.)*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996, ISBN 1-56592-161-5.

[22] Nayuki. Reference arithmetic coding. *Project Nayuki*, 2018. Available from: https://www.nayuki.io/page/reference-arithmetic-coding

[23] Witten, I. H.; Neal, R. M.; et al. Arithmetic Coding for Data Compression. *Commun. ACM*, volume 30, no. 6, June 1987: pp. 520–540, ISSN 0001-0782, doi:10.1145/214762.214771. Available from: http://doi.acm.org/10.1145/214762.214771

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

[25] Uhrig, C. Adaptive Arithmetic Coder (A0Coder). *Algorithmic Solutions*, 2017. Available from: `http://www.algorithmic-solutions.info/leda_manual/A0Coder.html`

[26] Holub, J. Statistical methods, Shannon-Fano coding, Huffman coding. *MI-KOD*, 2019. Available from: `https://courses.fit.cvut.cz/MI-KOD/lectures/mi-kod-03-huffman.pdf`

[27] Langmead, B.; Trapnell, C.; et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 2009. Available from: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2690996/`

[28] Novák, J. *Implementace kompresní metody DCA v jazyce Java*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2018.

[29] Zemek, L. *Implementace kompresních metod LZ77, LZ78, LZW v jazyce Java*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2018.

[30] Bobot, J. *Implementace kompresních metod LZY, LZMW a LZAP v jazyce Java*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2019.

[31] Nayuki. Reference Huffman coding. *Project Nayuki*, 2018. Available from: `https://www.nayuki.io/page/reference-huffman-coding`

[32] The Canterbury corpus. 1997. Available from: `http://corpus.canterbury.ac.nz/`

[33] Holub, J.; Reznicek, J.; et al. Lossless Data Compression Testbed: ExCom and Prague Corpus. In *Lossless Data Compression Testbed: ExCom and Prague Corpus*, 03 2011, p. 457, doi:10.1109/DCC.2011.61.

Appendix A

Acronyms

BWC Burrows-Wheeler compression

BWT Burrows-Wheeler transform

MTF Move-to-front transform

RLE Run-length en-coding

AC Arithmetic coder

HC Huffman coder

SCT Small Compression Toolkit

CPU Central processing unit

RAM Random-access memory

VCS Version control system

UML Unified Modeling Language

IDE Integrated development environment

CLI Command-line interface

OS Operating system

Appendix B

Contents of enclosed memory card

- `readme.txt` ..... the file with memory card contents description
- `src` ..... the directory of source codes
- `text` ..... the thesis text directory
  - `BP_Geletka_Filip_2019.pdf` ..... the thesis text in PDF format
  - `src` ..... the directory of LaTeX source codes of the thesis