FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Compression method LZFSE
Student:	Martin Hron
Supervisor:	Ing. Jan Baier
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2018/19

Instructions

Introduce yourself with LZ family compression methods. Analyze compression method LZFSE [1] and explain its main enhancements. Implement this method and/or these enhancements into the ExCom library [2], perform evaluation tests using the standard test suite and compare performance with other implemented LZ methods.

References

[1] https://github.com/lzfse/lzfse
[2] http://www.stringology.org/projects/ExCom/

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

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

Prague January 17, 2018

Bachelor's thesis

Compression method LZFSE

Martin Hron

Department of Theoretical Computer Science

Supervisor: Ing. Jan Baier

May 12, 2018

Acknowledgements

I would like to thank my supervisor, Ing. Jan Baier, for guiding this thesis and for all his valuable advice.

Declaration

In Prague on May 12, 2018 . . . .

Citation of this thesis

Hron, Martin. *Compression method LZFSE*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

This thesis focuses on LZFSE compression method, which combines a dictionary compression scheme with a technique based on ANS (asymmetric numeral systems). It describes the principles on which the method works and analyses the reference implementation of LZFSE by Eric Bainville.

As part of this thesis, the LZFSE method is added as a new module into the ExCom library and compared with other implemented compression methods using the files from the Prague Corpus. The impacts that different settings of adjustable LZFSE parameters have are also examined.

Keywords LZFSE, ExCom library, data compression, dictionary compression methods, lossless compression, finite state entropy, asymmetric numeral systems

Abstrakt

Tato práce se zabývá kompresní metodou LZFSE, která kombinuje slovníkovou kompresi s technikou založenou na ANS (asymmetric numeral systems). Práce popisuje principy, na kterých tato metoda funguje, a analyzuje referenční implementaci, jejíž autorem je Eric Bainville.

V rámci této práce je metoda LZFSE přidána jako nový modul do knihovny ExCom a porovnána s ostatními implementovanými metodami na datech Pražského Korpusu. Dále je prozkoumán vliv nastavitelných parametrů metody LZFSE.

Klíčová slova LZFSE, knihovna ExCom, komprese dat, slovníkové kompresní metody, bezeztrátová komprese, finite state entropy, asymmetric numeral systems

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Motivation

The amount of computer data produced by various information systems and applications is huge and grows every day. Therefore, vast quantity of data needs to be transmitted over the network or stored on some medium. To speed up data transmission and conserve storage space it is useful to reduce size of computer data while still maintaining the information that is contained in it. This is the purpose of data compression. Compression achieves data size reduction by decreasing redundancy and changing information representation to one that is more efficient.

Data compression is used very commonly even by non-professional computer users. Often we want to archive some files, transfer them to another computer or share them with other users. In those common cases, compression ratio (i.e. how much space is saved) and speed are typically more or less equally important, so we generally need to make a compromise between speed and compression quality. Also for compressing general files, a lossless method is usually required so that no vital information is lost.

For example, ZIP is frequently used archive file format that is probably known to most computer users. ZIP archives are usually compressed using a lossless algorithm called DEFLATE, which is designed to keep good balance between compression ratio and speed.

LZFSE is a new open source data compression algorithm developed by Apple Inc. It is designed for similar purpose as DEFLATE algorithm but is claimed to be significantly faster (up to three times) and to use less resources while preserving comparable compression ratio. LZFSE combines dictionary compression with an implementation of asymmetric numeral systems. Asymmetric numeral systems is a recent entropy coding method based on the work of Jarosław Duda, which aims to end the tradeoff between compression ratio and speed.

Various compression methods are collected in the ExCom library. ExCom is a library developed as part of Filip Šimek's thesis on CTU in 2009, which supports benchmarking and experimenting with compression methods. ExCom contains many dictionary algorithms. However, it does not yet contain the LZFSE method, nor any other method that would use asymmetric numeral systems. Integrating LZFSE into the ExCom library will allow to run, experiment with this method and perform benchmarks on it to confirm its enhancements.

Goal

The goal of this thesis is to analyse the LZFSE compression method, explain its main enhancements and implement it into the ExCom library. Then evaluate the implementation using the standard test suite, perform benchmarks and compare results with other dictionary compression methods to verify these enhancements.

Organization of this thesis

The first chapter explains basic concepts of data compression. It also introduces various classifications of compression methods and defines terms used further in this thesis.

The second chapter focuses on the LZ family of dictionary compression algorithms. It describes the LZ77 and LZ78 compression methods, which commonly serve as a foundation for other dictionary methods.

Asymmetric numeral systems, a family of entropy encoding methods, is addressed in the third chapter. This chapter first introduces basic concepts of entropy coding and briefly summarises most common approaches. The principles on which asymmetric numeral systems work are then explained. A variant of asymmetric numeral systems called finite state entropy, which is used in the LZFSE method, is also discussed further in this chapter.

The fourth chapter focuses on the LZFSE method. It describes how this method works and analyses its reference implementation written by Eric Bainville.

The fifth chapter deals with the integration of LZFSE into the ExCom library. It describes the steps taken to implement LZFSE into ExCom and lists deviations from the reference implementation.

Finally, the sixth chapter presents the results of benchmarks made using the ExCom library and compares performance of LZFSE with other compression methods. The impacts that different settings of adjustable LZFSE parameters have on compression ratio and speed are also discussed here.

Chapter 1

Data compression

This chapter introduces basic concepts of data compression and defines terms used further in the text. Classification of compression methods into various categories and performance measurement of methods are also discussed in this chapter.

1.1 Basic data compression concepts

Data compression is a process of transforming computer data into a representation requiring fewer bits while preserving information contained in the data. The reverse process of reconstructing the original data from this representation is called *decompression.* The terms *encoding* and *decoding* will also be used to refer to the compression and decompression processes respectively.

The input data (for the compression process) are referred to as the *original* or *unencoded* data. The output data are considered *compressed* or *encoded.*

Algorithm that takes the original data and generates a representation with smaller size (compresses data) is called *compression algorithm.* Algorithm that operates on this compressed representation to reconstruct the original data or produce their approximation is called *reconstruction algorithm.* Both the compression and reconstruction algorithms are referred to together by the term *compression method.*

1.2 Information entropy and redundancy

Two concepts of information theory, entropy and redundancy, are introduced in this section.

Information entropy is a term introduced by Claude Shannon in 1948 [1]. Entropy measures the amount of information contained in the data. Intuitively, the more diverse and unpredictable the data are, the more information they contain. It is possible to view entropy as an average minimum number of bits needed to store the information.

Redundancy measures the inefficiency of information representation. It is a difference between entropy of data of given length and its maximum value. Compression methods may attempt to reduce data redundancy.

1.2.1 Entropy

Let X be a random variable that takes n values with probabilities $p_1, p_2, \ldots, p_n$ respectively. Then *entropy* of X, denoted $H(X)$, is defined to be [1]:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

1.2.2 Redundancy

Let S be a string of l characters and $H(c_i)$ be the entropy of its i-th character. Then entropy of string S is $H(S) = \sum_{i=1}^{l} H(c_i) = l \cdot H_{AVG}$, where H_{AVG} is average symbol entropy. [2]

Let $L(S)$ be the length of S in bits. Then *redundancy* of string S, denoted $R(S)$, is defined as

$$R(S) = L(S) - H(S)$$

Lossless methods of compression (see Section 1.3.1) work by minimizing redundancy in the input data. [2]

1.2.3 Source coding theorem

The *source coding theorem* formulated by C. Shannon in 1948 in [1] establishes theoretical limits of data compression.

Let S be a sequence of N symbols. Each of those symbols is an independent sample of a random variable X with entropy $H(X)$. Let L be the average number of bits required to encode the N symbol sequence S. The source coding theorem states that the minimum L satisfies [3]:

$$H(X) \leq L < H(X) + \frac{1}{N}$$

1.3 Classification of compression methods

There exists a large variety of data compression methods intended for several different purposes. Some basic classifications of compression methods based on their various properties are discussed here.

1.3.1 Lossy/lossless compression

Lossless compression methods allow the exact reconstruction of the original data, and thus no information is lost during the compression. Lossless compression methods should be used when even a small change to the data would have serious impact. For instance, when compressing text files, a change to just a few letters may alter the meaning of the original text or render it unreadable. This is especially true for compression of source codes.

Lossy compression methods, on the other hand, lose some information during the compression process. They achieve better compression at the expense of preventing the exact reconstruction of the data. They are generally used for applications where this is not a problem. An example of such application is video compression. We generally do not care that the reconstruction of a video is slightly different from the original as long as noticeable artifacts are not present. As such, video is usually compressed using lossy compression methods [4, p. 5].

1.3.2 Adaptability

Another possible classification of compression methods is according to how they manage their model of the input data and how they modify their operations.

A *nonadaptive* compression method does not modify its operations based on the data being processed. They may be faster because there is no time spent on updating the model. However, they would generally achieve worse compression ratio. Such methods perform best on data that is all of a single given type [5, p. 8].

An *adaptive* compression method, on the other hand, modifies its algorithms according to the input data. The model is updated dynamically as the input is processed. Unlike for semi-adaptive methods, the input is processed in single run.

If the method adapts itself to local conditions in the input stream and changes its model as it moves through the input, it is called *locally adaptive.* For example move-to-front algorithm uses this approach [5, p. 37].

Semi-adaptive methods use a 2-pass algorithm. During the first pass, whole input data are read and a model is built. In the second pass this model is used for compression. As the model is built specifically for the data being compressed, they generally achieve very good compression. However, it is not possible to infer the model from the compressed data. For this reason, the model must be appended to the compressed data for reconstruction algorithm to work, thus worsening the compression ratio. Moreover, these methods may be slow as the input is read twice.

1.3.3 Symmetrical compression

The situation when compression algorithm is similar to the reconstruction algorithm but works in the "opposite" direction is called *symmetrical compression* [5, p. 9]. In such case, complexity of the compression algorithm is generally the same as complexity of the reconstruction algorithm. Symmetrical methods are useful when compression and decompression are performed equally frequently.

Asymmetrical compression is useful when decompression is performed much more often than compression and vice versa. For instance, when a file is compressed once, then uploaded to a server and downloaded by users and decompressed frequently, it is useful to have simple and fast reconstruction algorithm at the expense of a more complex compression algorithm.

1.3.4 Types of lossless compression

- *Dictionary methods* use some form of *dictionary*, a data structure where previously seen occurrences of symbol sequences are saved. When a sequence present in the dictionary is encountered in the input again, a reference to the dictionary is written to output instead. LZ family algorithms belong to this category.

- *Statistical methods* work by assigning shorter codes to symbols with higher probability of occurrence and longer codes to rare symbols. They often use semi-adaptive approach (see Section 1.3.2). During the first pass, frequencies of symbols are counted and statistical model is built. During the second pass, the input is then encoded. Examples of this category include Huffman coding and arithmetic coding, which are both very briefly described in Chapter 3.

Apart from those two main types, other types exist such as contextual methods (for example PPM and DCA algorithms).

1.4 Measures of performance

Performance of compression methods may be expressed using various measures. One of commonly used metrics is compression ratio.

1.4.1 Compression ratio

Compression ratio is defined to be:

$$\text{Compression ratio} = \frac{\text{size of compressed data}}{\text{size of original data}}$$

The lower the compression ratio is, the bigger amount of space was saved thanks to the compression. For instance compression ratio of 0.8 (or 80%) means that the compressed data size is 80% the original size.

If the compression ratio has value greater than 1, in other words the size of compressed data is greater than the size of original data, then we will say *negative compression* occurred.

1.4.2 Data corpora

To test and compare performance of different compression algorithms various corpora exists. *Corpus* is a standardized set of files containing most common types of binary and text files to test compression performance on.

Several corpora were published, for instance Calgary Corpus (1987) or Canterbury Corpus (1997). In 2011, the Prague Corpus was established on Czech Technical University in Prague [6].

1.5 ExCom library

ExCom [7] is a modular compression library written in C++. ExCom contains many compression methods and is extensible by adding new modules. It provides support for module testing, time measurement and benchmarking. It is described in detail in [2].

1.6 Hash function and hash table

1.6.1 Hash function

Hash function is a mathematical function that maps input of arbitrary length to fixed-size output. This output value is called a *hash*.

Let U be a set of all possible inputs: $U = \bigcup_{i=1}^{n} \{0,1\}^i$, n is maximum input length (theoretically n may even be infinity). Then function

$$f\colon U \to \{0,1\}^l$$

is a hash function that maps any element of U to output of fixed length l.

The hash function must be deterministic. If $k_1 = k_2$, then also $f(k_1) = f(k_2)$. So for given input, hash function f always produces the same output value. If $k_1 \neq k_2$, but $f(k_1) = f(k_2)$, then we say a *collision* occurred.

Good hash function (for use in hash tables) is usually required to have the following properties:

- It should distribute the output values evenly to minimize number of collisions.
- It should be very fast to compute.

1.6.2 Multiplicative hash

One possible method to obtain a good hash function is described in [8, Section 11.3]. The input value k is multiplied by a constant $A \in [0, 1]$ and the fractional part of the result is then multiplied by integer m. The floor of this number is the hash value. The hash function then looks as follows:

$$f(k) = \lfloor m(kA \bmod 1) \rfloor$$

$kA \bmod 1 = kA - \lfloor kA \rfloor$ is the fractional part of kA. Value of m is not critical, it is usually chosen as a power of 2. [8]

See [8] for implementation details of this method. This type of hash function is also used in the LZFSE reference implementation.

1.6.3 Hash table

Hash table is a data structure which implements dictionary used by dictionary compression methods. It stores its elements in an array and uses hash function for indexing. When accessing an element, its hash is computed and used as an index. However, since hash function may produce collisions, two different elements may get mapped to same position in the array.

Various techniques are used to resolve collisions:

- *Separate chaining* – For each position in the array (i.e. each possible hash value), list of elements that map to this position is kept. Linked list is most commonly used for this purpose. When collision occurs during insertion, element is simply added at the end of the corresponding list. When accessing elements, the list on given position must be scanned until the element being accessed is found.
- *Open addressing* – All elements are stored directly in the array. Insertion is done by searching for first empty position using some predefined order. When accessing an element, the array is searched in the same order until either the required element or an empty cell is encountered.

Chapter 2

LZ family algorithms

LZ family algorithms is a family of dictionary compression algorithms named after *LZ77* and *LZ78*, two algorithms published by Abraham Lempel and Jacob Ziv in 1977 [9] and 1978 [10] respectively. LZ77 and LZ78 will both be described in this chapter.

LZ77 and LZ78 form the basis for many other dictionary compression methods, for example LZSS, LZW or LZ4 methods. LZFSE described in this thesis is also Lempel-Ziv style compression method.

2.1 LZ77

LZ77 (also referred to as *LZ1*) is a dictionary compression method designed by Abraham Lempel and Jacob Ziv and published in 1977 [9]. The main idea of this method is using the previously proccesed part of input as the dictionary [5, p. 176]. It is suitable for data that are compressed once and decompressed frequently. LZ77 is commonly used as a foundation for more complex compression methods.

2.1.1 Compression

LZ77 maintains part of input data in a structure called *sliding window*. Sliding window is divided into two parts called search buffer and look-ahead buffer. *Search buffer* contains part of already processed input, while *look-ahead buffer* contains unencoded data yet to be compressed. Sliding window has a fixed size. Size of the search buffer is commonly 8 192 bits, while the look-ahead buffer usually has about 10 to 20 bits [11].

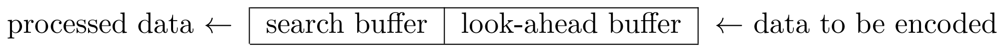

Figure 2.1: LZ77 sliding window

As the input is processed, the sliding window moves forward in the input. Compressed data from look-ahead buffer transit into the search buffer. Data from the beginning of the search buffer are shifted out and look-ahead buffer is filled with new unencoded data.

In each step, the compression algorithm searches the whole search buffer to find the longest prefix of look-ahead buffer that is present in it. It scans the search buffer backwards until the symbol look-ahead buffer starts with is encountered. Then it counts how many characters following this symbol match the look-ahead buffer prefix. If this match is the longest found so far, its position and length are saved. The position of the longest match is kept as an *offset*, distance from the beginning of the look-ahead buffer. Those steps are repeated until the beginning of the search buffer is reached. A triplet containing offset of the longest prefix, its length and the symbol that follows it is then written to the output. The sliding window is then moved and the whole process is repeated. See Algorithm 1 for pseudocode of the compression process.

Algorithm 1: LZ77 compression

```
1 initialize moving window (divided into search and look-ahead buffer)
2 fill look-ahead buffer from input
3 while look-ahead buffer is not empty:
4     find longest prefix $p$ of look-ahead buffer by scanning the search
        buffer backwards
5     $offset$ := distance of $p$ from the beginning of the look-ahead buffer
6     $length$ := length of $p$
7     $X$ := first character after $p$ in look-ahead buffer
8     output triplet ($offset$, $length$, $X$)
9     shift (move) window by $length + 1$
```

2.1.2 Compression example

An example of LZ77 compression on input string “possessed posy” is shown in Figure 2.2. Size of the sliding window in this example is 8 characters, half of which is search buffer.

As seen in the example, size of the sliding window strongly affects compression ratio. For instance, in step 7 we search for a prefix beginning with “p”. No match is found. However string “pos” was encountered earlier in the input and would yield a match of length 3 if present in the search buffer.

The example input string was 14 characters long and it was encoded into 10 triplets. Depending on the representation of triplets, this could result in negative compression in this case.

step		sliding window				output
1.			poss	essed␣posy	⇒	(0, 0, p)
2.		p	osse	ssed␣posy	⇒	(0, 0, o)
3.		po	sses	sed␣posy	⇒	(0, 0, s)
4.		pos	sess	ed␣posy	⇒	(1, 1, e)
5.	p	osse	ssed	␣posy	⇒	(3, 3, d)
6.	posse	ssed	␣pos	y	⇒	(0, 0, ␣)
7.	posses	sed␣	posy		⇒	(0, 0, p)
8.	possess	ed␣p	osy		⇒	(0, 0, o)
9.	possesse	d␣po	sy		⇒	(0, 0, s)
10.	possessed	␣pos	y		⇒	(0, 0, y)

Figure 2.2: Example of LZ77 compression

2.1.3 Decompression

Decompression uses sliding window of the same size as compression to keep previous output. For each triplet, the sequence from previous output it points to is copied on the current position in the output and then the following symbol contained in the triplet is also written to output. See Algorithm 2 for simplified pseudocode of the decompression process. Decompression is much simpler and faster than compression. Therefore, LZ77 is classified as an asymmetric compression method.

Algorithm 2: LZ77 decompression

```
1 while input is not empty:
2     read triplet ($offset$, $length$, $X$) from input
3     go back by $offset$ characters in previous output and output $length$
      characters starting on that position
4     output $X$
5     move the sliding window by $length + 1$
```

2.1.4 Decompression example

Figure 2.3 shows LZ77 decompression of data encoded earlier in the compression example (see Figure 2.2). Size of the sliding window is always the same as during compression. In this case it is 8 characters.

2.2 LZ78

LZ78 (sometimes referred to as *LZ2*) was introduced by Abraham Lempel and Jacob Ziv in 1978 [10].

step	input		output	sliding window		
1.	(0, 0, p)	⇒	p			p
2.	(0, 0, o)	⇒	o		p	o
3.	(0, 0, s)	⇒	s		po	s
4.	(1, 1, e)	⇒	se		pos	se
5.	(3, 3, d)	⇒	ssed	p	osse	ssed
6.	(0, 0, ␣)	⇒	␣	posse	ssed	␣
7.	(0, 0, p)	⇒	p	posses	sed␣	p
8.	(0, 0, o)	⇒	o	possess	ed␣p	o
9.	(0, 0, s)	⇒	s	possesse	d␣po	s
10.	(0, 0, y)	⇒	y	possessed	␣pos	y

Figure 2.3: Example of LZ77 decompression

Unlike LZ77, which uses a fixed size sliding window (see Section 2.1), LZ78 maintains a dictionary of previously seen *phrases* (sequences of symbols). This dictionary is built as the input is processed and it may potentially contain unlimited number of phrases. When a repeated occurrence of a phrase is found during encoding, dictionary index is output instead.

2.2.1 Compression

In each step, the dictionary of previously seen strings is searched for the longest prefix of the unprocessed part of the input. A pair $(index(w), K)$ is written to the output, where $index(w)$ is an index referring to the longest matching dictionary entry w and K is a symbol immediately following w in the input [12]. Additionally, new phrase consisting of w concatenated with K (denoted wK) is inserted into the dictionary. This process is repeated until the whole input is encoded. See Algorithm 3 for the compression pseudocode[1].

2.2.2 Decompression

During decompression, the dictionary is gradually build in a similar way as during compression. See Algorithm 4 for the pseudocode of LZ78 reconstruction algorithm. LZ78 also preserves an important property of LZ77 that the decompression is generally significantly faster than the compression [12].

2.2.3 Dictionary

To represent the dictionary a structure called *trie* (or prefix tree) may be used. Trie is an ordered tree data structure used to store strings. All vertices

[1]NIL denotes an empty string. The dictionary starts with an empty string as its only element.

Algorithm 3: LZ78 compression [12]

```
w := NIL
while input is not empty:
    K := next symbol from input
    if wK exists in the dictionary:
        w := wK
    else:
        output pair (index(w), K)
        add wK to the dictionary
        w := NIL
```

Algorithm 4: LZ78 decompression

```
while input is not empty:
    read pair (i, K) from input
    w := get phrase pointed to by i from dictionary
    output wK
    add wK to the dictionary
```

with common parent begin with the same prefix. The root node represents an empty string and going from it to a certain node yields a phrase from the dictionary.

In the beginning, dictionary contains only one phrase – an empty string. As mentioned before, the dictionary may conceptually contain unlimited number of phrases. Because of this, some LZ78 implementations may have very high space requirements. To avoid this problem, it is possible to limit the size of dictionary by some upper bound and remove the least common phrase when this bound is reached or simply clear the dictionary and start building it from scratch.

Several algorithms based on LZ78 exists, differing mainly in dictionary implementation and usage. One of the most popular modifications is LZW made by Terry Welche in 1984. In LZW the dictionary is initialized with all symbols from the input alphabet and so a match is always found. Therefore, only the index to the dictionary is output in each step [12].

Chapter 3

Asymmetric numeral systems

Asymmetric numeral systems (ANS) is a family of entropy coding methods based on the work of Jarosław Duda. A variant of asymmetric numeral systems called *finite state entropy* (FSE) is also part of the LZFSE compression method.

This chapter will introduce important entropy coding concepts and some of the most common methods. Asymmetric numeral systems and their FSE variant will be described further in the chapter.

3.1 Entropy coding

Entropy coding (or entropy encoding) is a lossless data compression scheme. It achieves compression by representing frequently occurring symbols with fewer bits and rarely occurring elements with more bits.

Huffman coding and arithmetic coding are two of the most common entropy coding methods and will be briefly described below.

Asymmetric numeral systems is another family of entropy coding methods. ANS methods are increasingly used in compression as they combine compression ratio of arithmetic coding with compression speed similar to that of Huffman coding method [13]. ANS is described in Section 3.2 and finite state entropy, which is their variant, in Section 3.3.

Entropy coding methods are commonly used in combination with some dictionary compression scheme. For instance, the algorithm DEFLATE[2] uses a combination of LZ77 (see Section 2.1) and Huffman coding. LZFSE also belongs to this category as it combines a LZ-style compression algorithm with finite state entropy coding [14].

[2]DEFLATE is a lossless compression algorithm originally designed for the ZIP file format.

3.1.1 Huffman coding

Huffman coding is an entropy coding technique developed by David A. Huffman. Huffman coding approximates probability of symbol occurrence as a (negative) power of 2. Unlike more complex methods, Huffman coding always uses integral number of bits to represent a symbol and it encodes each symbol separately.

Every occurrence of one symbol is always encoded into same code word. Code words are assigned to symbols in a way that no code word is prefix of any other (such code is called a *prefix code*). Shorter codes are assigned to common symbols.

When the probabilities of the symbols are negative powers of two, Huffman coding produces the best results [5, p. 74].

Huffman coding is a simple compression method with low procession cost. Other entropy coding methods, such as arithmetic coding or ANS, may achieve considerably better compression ratio.

3.1.2 Arithmetic coding

Arithmetic coding is another entropy coding technique. It encodes the input data as an interval of real numbers between 0 and 1. This interval becomes smaller as the input is encoded and number of bits needed to specify it grows [15].

Unlike Huffman coding, which approximates symbol probabilities by powers of 2, arithmetic coding is very precise but it is also more complex. Description of arithmetic coding is not the focus of this work and may be found in [15].

3.2 Asymmetric numeral systems

Asymmetric numeral systems is an entropy coding technique introduced by Jarosław (Jarek) Duda.

Apart from ANS, most common approaches to entropy coding are Huffman coding and arithmetic coding (both were briefly described in previous section). Huffman coding approximates symbol probabilities with powers of 2 and so it generally does not achieve as good compression ratio. Arithmetic coding uses almost exact probabilities and so it achieves compression ratio close to the theoretical limit (see Section 1.2.3), but it has larger computational cost than Huffman coding. According to its author, asymmetric numeral systems achieve comparable compression ratio as arithmetic coding while having procession cost similar to that of Huffman coding. [13]

While arithmetic coding uses two numbers (states) to represent a range, asymmetric numeral systems uses only one state – a single natural number. ANS works by encoding information into this single natural number. To add new information to the information already stored in this number (state), new digits could be appended either in the most or the least significant position. Arithmetic coding is an example of the first approach. ANS uses the second option, new digits are added in the least significant position. [16]

In the standard binary numeral system a sequence of n bits $(s_0, s_1, \ldots, s_{n-1})$ would be encoded as a natural number $x = \sum_{i=0}^{n-1} s_i 2^i$. To add information from a symbol $s \in \{0, 1\}$, all bits of x are shifted left and the least significant bit value is changed to s (i.e. s is appended to x). This changes x to $x\prime = C(x, s) = 2x + s$, where C is a *coding function.* The reverse process of decoding value of s and previous state x would use a *decoding function* $D(x\prime) = (x, s) = (\lfloor x\prime/2 \rfloor, x\prime \bmod 2)$. Encoding a symbol sequence would be done by starting with some initial state and repeatedly using the coding function until all symbols are encoded. To decode, decoding function would be applied until state x is equal to the initial state. Symbols are recovered in *reverse* order.

The scheme with coding and decoding functions presented above is optimal for any input with uniform probability distribution of symbols $P(s_i = 0) = P(s_i = 1) = \frac{1}{2}$. The basic concept of ANS is to change this behaviour to make it optimal for any chosen asymmetric probability distribution. In the above example, $x\prime$ is either even or odd, based on the value of s. Therefore, $x\prime$ is x-th even or odd number, depending on s. As stated before, this scheme is optimal for storing uniformly distributed symbols. In ANS this division into even and odd subsets of natural numbers is replaced with division into subsets whose densities correspond with the chosen distribution [16].

To optimize the coding procedure for the assumed probability distribution: $P(s_i = s) = p_s$, subsets are redefined in such way, that their densities correspond to this probability distribution. The rule to add information from symbol s to the information that is already stored in number x is: $x\prime = C(s, x)$. The coding function $C(s, x)$ returns the x-th appearance of s in the corresponding subset [16].

Figure 3.1 shows example of asymmetric binary system for probability distribution: $P(s_i = 0) = \frac{1}{4}, P(s_i = 1) = \frac{3}{4}$. Encoding a binary sequence 1101 using tables shown in the figure and starting with initial state $x = 1$ would produce:

$$\text{standard binary system: } 1 \xrightarrow{1} 3 \xrightarrow{1} 7 \xrightarrow{0} 14 \xrightarrow{1} 29$$

$$\text{asymmetric binary system: } 1 \xrightarrow{1} 2 \xrightarrow{1} 3 \xrightarrow{0} 12 \xrightarrow{1} 17$$

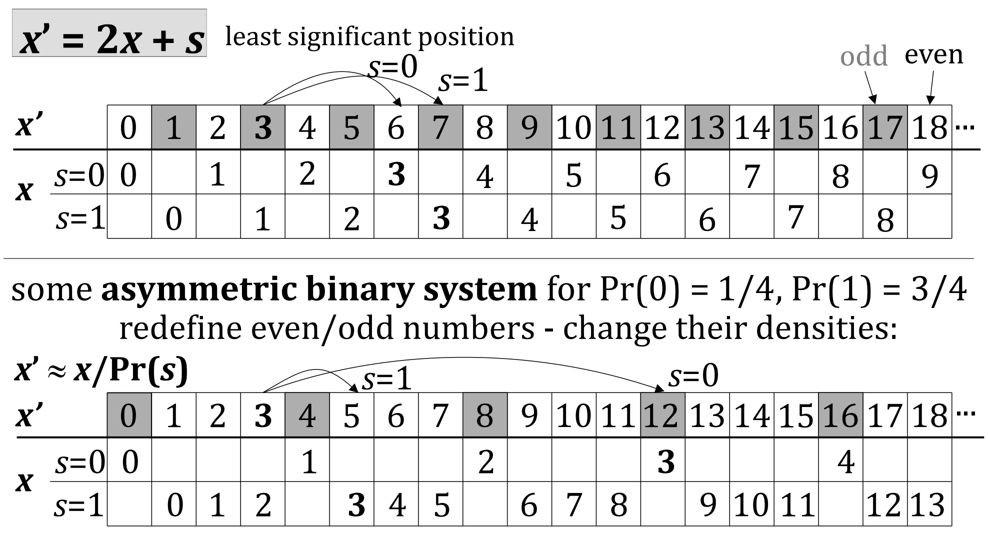
x' = 2x + s
least significant position
s=0
s=1
odd
even
x'
x
s=0
s=1
some asymmetric binary system for Pr(0) = 1/4, Pr(1) = 3/4
redefine even/odd numbers - change their densities:
x' ≈ x/Pr(s)

Figure 3.1: Adding information to state x in standard binary numeral system (up) and ANS (bottom) [16]. The new state becomes x-th element of the s-th subset. In ANS subsets are defined so that their densities correspond to the assumed probability distribution[3].

As seen in this example, encoding the sequence using ANS produces smaller final state (number) than using the standard binary system, because it adapts the subsets to given probability distribution of input symbols. This difference would grow with the input size.

Decoding would work similarly by following the steps in opposite direction. As seen in the example figure, the previous state and symbol are unambiguous for each state. It is always possible to get previous state and symbol from the table, for instance for state $x = 17$, the previous state is 12 and the symbol is 1. This way, previous state would be retrieved until the initial state is reached, decoding the whole sequence in the process. The symbols are retrieved in reverse order.

In practice it is preferable to avoid operations on very large numbers (states), as such operations may be very demanding. When a state is larger than some maximum value during encoding, some of its least significant bits are transferred to the output stream. This way the state will always remain in some fixed range. Naturally, decoding has to be modified accordingly and it must be ensured that encoding and decoding steps are inverse of each other. The exact mechanisms to achieve that are described in [16].

[3]In the original figure, the probabilities of symbols are defined as $Pr(0) = 3/4$ and $Pr(1) = 1/4$, but here they were changed to correspond to the subsets.

Also, because encoding works in opposite direction than decoding, it might be preferable to encode backwards starting from the last symbol so that decoding would start from the beginning and proceed forward.

Concepts described in this section could also be generalized for numeral systems of different base than 2 (i.e. for more possible input symbols). Several variants of ANS exists. An abstract description of finite state entropy (tANS), variant used in LZFSE, may be found in the next section (see Chapter 4 for details of its implementation in LZFSE). Detailed description of asymmetric numeral systems and all their variants may be found in [13] and [16].

3.3 Finite state entropy

Finite state entropy is a variant of asymmetric numeral systems (described in previous section). It is also called *tANS* (or table ANS) because the entire behaviour is put into a single coding table, yielding finite state machine.

For the assumed probability distribution, given usually in form of symbol frequencies, symbols must be distributed into subsets (as shown in previous section). Densities of those subsets must correspond to given probability distribution. "*Unfortunately, finding the optimal symbol distribution is not an easy task.*" It could be done by checking all possible symbol distributions. However, in practice some simple heuristic is usually used instead [13]. A simple and fast method of pseudorandomly distributing symbols into subsets is described in detail in [16].

The table used for encoding is called *encoding table.* For each symbol and state, the next state is stored in it. So the table stores results of coding function $C(x, s)$ for all possible (x, s) pairs. This table is created based on symbol distribution into subsets described in previous paragraph. The encoding table may also be stored in one-dimensional array as it is better for memory handling efficiency [16].

The encoding process is similar to the ANS encoding described in Section 3.2, but instead of computing the coding function $C(s, x)$ in each step, the next state is obtained from the encoding table.

Additionally, the mechanism of ensuring that states remain in some fixed range, which was discussed in previous section, is usually employed. If a state is bigger than given upper bound, its least significant bits are written to the output and the state is shifted right until it fits into the range. The number of bits that will have to be transferred to the output stream may be computed beforehand and stored for each possible transition (or symbol) to avoid additional computations during encoding.

The table that is used for decoding is called *decoding table.* For decoding to work, it is necessary to be able to get this table, and so some information must be added to the compressed data when encoding. This may be table of symbol frequencies, which was also used to construct the encoding table. This information is usually part of some *header* and may itself be compressed.

As with the encoding step, the decoding step of FSE is similar to the decoding step of general ANS (see Section 3.2). Instead of computing the result of decoding function $D(x)$, the previous state and symbol are looked-up in the decoding table.

This section contains only abstract description of finite state entropy, as there are several different possibilities of how to implement this method. More thorough description of this method along with exact pseudocodes for initialization and encoding and decoding steps may be found in [16]. Implementation of finite state entropy in LZFSE is described in Chapter 4.

CHAPTER 4

LZFSE

LZFSE is a lossless compression algorithm that was developed by Apple Inc. and later released as open-source. The algorithm is based on Lemple-Ziv dictionary compression (described in Chapter 2) and uses finite state entropy, a variant of ANS described in Chapter 3, for entropy coding [17].

According to its authors, LZFSE matches the compression ratio of zlib[4] level 5, while being between two to three times faster for both compression and decompression and having better energy efficiency. LZFSE is a balanced compression method intended for situations when compression ratio and decompression speed are equally important [18].

A reference implementation of LZFSE in C language written by Eric Bainville was released in 2016 [14]. This reference implementation serves as a definition of the LZFSE method, as Apple did not publish any other detailed description of it (e.g. pseudocodes or format description).

This chapter describes the reference implementation of the LZFSE method. All details pertain to the version from 22 May 2017, which is the most recent version as of March 2018 [14]. The file structure of the reference implementation and its main functions are described in Appendix B.

4.1 Compression

LZFSE combines LZ-style dictionary compression algorithm with finite state entropy. The dictionary algorithm serves as a *frontend* and the finite state entropy serves as a *backend* and is used to encode the matches and literals produced (emitted) by the frontend.

[4]zlib is a compression library that provides an abstraction over the DEFLATE compression algorithm. zlib implementation of DEFLATE allows users to choose a compression level between 0 and 9. Lower level means preferring speed and higher level means prioritizing compression ratio. Level 5 provides a good balance between speed and compression ratio.

LZFSE compresses the input data and generates one or more *compressed blocks*. There are several types of blocks that may be produced by the reference implementation. Each block starts with a *header* containing information about this block. Headers for different types of blocks differ not only in constitution but also in size. Regardless of block type however, the first four bytes of a header consist of special value identifying the block type (referred to as a *block magic*). There is also a special value indicating the end of file.

The blocks produced by LZFSE compression are described at the end of Section 4.1.2. There is also an *uncompressed block* type for unencoded data. If the input is really small (the default threshold is 8 bytes) or if LZFSE compression fails for some reason, the input is just copied unaltered and saved as an uncompressed block.

The reference implementation also contains a heuristic that will fallback to a simpler compression algorithm called LZVN when the input is smaller than a given threshold. The reason is that on small data LZFSE does not perform as well according to the author [14]. Because this thesis focuses on LZFSE, this algorithm will not be covered here. LZVN compression also was not implemented as part of the LZFSE module into ExCom, as the goal was to test the performance of the LZFSE method.

4.1.1 Frontend

The LZFSE frontend works by searching for matches in the input. A *match* is a sequence of bytes that exactly matches other sequence encountered earlier in the input. The frontend scans the input for matches using a procedure described below. These matches are then *emitted* – sent to the backend, where they are encoded.

The frontend produces matches in form of triplets (L, M, D), where M is the length of the match, D is the distance to the previous occurrence (i.e. offset) and L is the number of literals emitted with this match. *Literals* are values from the original input stream. When a match is emitted, all literals between this match and the previous one are also emitted with it. Emitted literals are kept in concatenated form and then encoded separately in the backend. For details on how the emitted matches and literals are processed in the backend, see Section 4.1.2.

Triplets that represent the LZFSE matches are somewhat similar to triplets produced by the LZ77 algorithm (see Section 2.1). The difference is that while LZ77 triplets contain only one symbol following the match, in case of LZFSE the L symbols (literals) before the match are emitted with it.

Because LZFSE uses a 32-bit offset for encoding matches, if the input is large, it must be divided into smaller blocks and each of them processed separately. This is done for inputs significantly smaller than the 2 GiB upper bound, as it is better for algorithm efficiency [14].

4.1.1.1 History table

LZFSE uses a *history table* and hashing to search for matches. The history table resembles a hash table with separate chaining (see Section 1.6.3) to some extent. It is an array indexed by hash values. On each position in the array, there is a *history set* containing locations of four-byte sequences encountered earlier in the input that hash to the same value – the index of this history set in the array.

Apart from the position, the first four bytes at that position are also stored for each element in the history set to quickly eliminate false-positive matches caused by hash function collisions. By default, each history table set can hold up to four positions. When more positions of four-byte sequences with this hash are found, the oldest element from the set is removed and the new entry is added in the beginning of this set.

For the hash function, an implementation of multiplicative hash described in Section 1.6.2 is used. This implementation also allows to change the hash length (how many bits the hash function produces), which is controlled by a parameter. The hash length also determines the size of the history table.

To find match candidates for some position in the input, hash function is computed for the first four bytes on that position. The hash value is then used as an index into the history table, yielding a set of match candidate positions. Because entry for each candidate also contains the first four bytes on that position, collisions can be quickly eliminated by comparing these bytes with bytes on current position. Figure 4.1 depicts the history table and illustrates this process.

As already mentioned, the size of the history table depends on the hash length. If the hash function produces 14-bit values, which is the default, the table will contain $2^{14} = 16384$ elements (sets).

4.1.1.2 Searching for matches

Algorithm that searches for matches keeps a *pending match* – the best match found so far, which will be emitted unless a better one is found. Note that the matches are internally kept by the frontend in a slightly different form than how they are then emitted to the backend. Here the position where the match starts, the position where the previous occurrence starts (called *source*) and the length of the match are stored.

The matches are searched by repeating the following procedure for each position in the input. First, a hash value of first four bytes starting on current

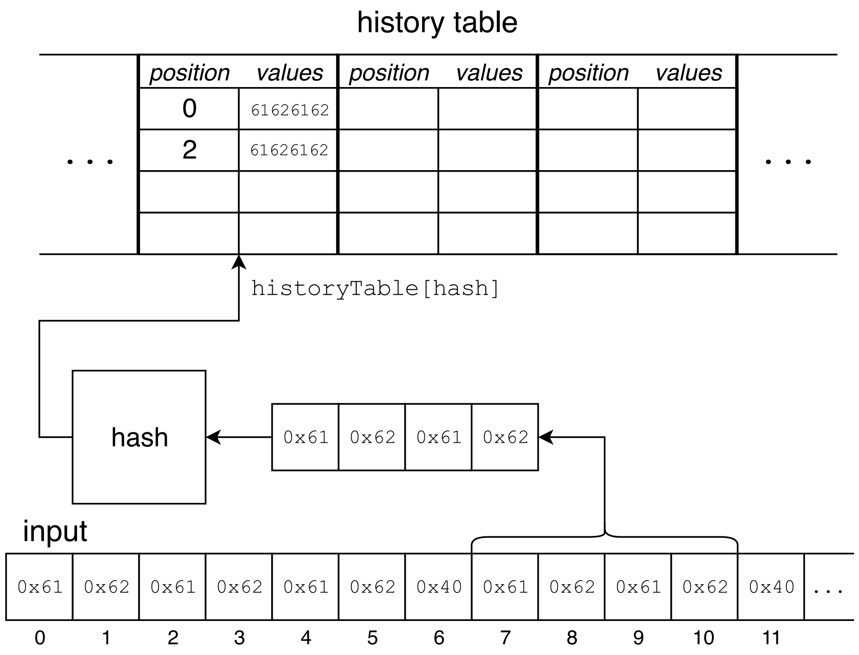

Figure 4.1: The history table used by LZFSE frontend to search for matches. A set on index i contains positions of previously seen four-byte sequences (and their values) with hash equal to i. To get match candidates for a position in the input, the hash of the four-byte sequence starting on that position is computed. This value is than used as an index into the history table.

position is computed. An entry for this position is later added into the correct set of the history table before moving on to the next position.

The current position may be inside some previous match. This can be detected by keeping track of the next unencoded literal (i.e. the first literal in the input that was not part of any previous match). If the current position is before this literal, it is inside a previously emitted match. In that case, the history table is just updated and the algorithm continues on the next position.

The computed hash value for current position is used to get match candidates from the history table as described above. The best match is then chosen from these candidates. This is done by counting how many bytes on each of the candidate positions match the bytes following the current position. The longest match is taken. Only matches of length at least 4 are considered [14].

The best match found is also expanded backwards if possible. While the bytes preceding the starting position of the match and the bytes before the source position are equal, the match length is incremented and both positions are moved back by one.

If the best match found for this position (the *current match*) has length greater than some given threshold (default is 40), it is immediately emitted. Otherwise, if there is no pending match, the current match will become the new pending match and the algorithm will continue on the next position. If there is already a pending match however, it should be emitted first and then the current match should become the new pending match. This may not be possible if the pending match overlaps with the current match (i.e. the current match starts inside the pending match). In that case, the longer of these two will be emitted.

If no match is found for the current position, it may still be desirable to emit some literals so that the current position is not too far ahead of the next unencoded literal. If the distance is larger than some given threshold, the pending match is emitted (if there is one) along with some literals. Otherwise, the algorithm just moves on to the next position.

This procedure is repeated until the end of input buffer is reached. Then the pending match and any remaining literals are also emitted and a method of backend is called to produce the final block and an end-of-file marker.

4.1.2 Backend

The backend of the LZFSE encoder uses an implementation of finite state entropy described in Section 3.3. It encodes the matches and literals emitted by the frontend.

The frontend emits matches and literals by calling the appropriate function of the backend. The matches are emitted as (L, M, D) triplets described in Section 4.1.1.When literals need to be emitted separately (i.e. not as a part of match), they are emitted by creating a fake match, a match of length 0, and emitting them along with this match.

The encoder backend keeps buffers for emitted matches and literals. For each element of the (L, M, D) triplet there is a separate buffer and there is also a buffer for emitted literals. Whenever a match is emitted, each of its L, M and D elements is pushed into its respective buffer and the L literals emitted as a part of this match are pushed into the literals buffer. When any of these buffers is full, all stored matches and literals are encoded using finite state entropy and a compressed block is produced by the backend.

Contents of each of the four buffers is encoded separately – the numbers of literals (L), the match lengths (M), the offsets (D) and the concatenated literals are each encoded separately using finite state entropy. How the finite state encoder is implemented in the reference implementation is described in Section 4.1.2.1 below.

Before encoding, the frequencies of symbols are counted. These frequencies are then normalized and used to create the coding tables for finite state entropy encoders. The exact process is described in Section 4.1.2.1. The frequency tables are also saved into the header to be used during decoding.

The backend internally uses two types of headers. A *v_1 header* contains frequency tables, final encoder states and all the other information in unencoded form. However, the header is written to output in a different form called *v_2 header*. This type of header contains all the information from *v_1 header*, but the frequency tables are compressed using a form of Huffman coding. Moreover, the other fields from *v_1 header* are "*packed*" – copied in compact form to shared continuous block of memory to save some space[5]. This header will be output in the beginning of the compressed block.

The literals are encoded first. They are encoded bytewise but four literal bytes are encoded in each step using one FSE encoder for each. So there are four FSE encoders for literals, which rotates in encoding the literal bytes. The four literal FSE encoders all use the same frequency tables and they also share the encoding tables but each of them has its own state. The reference implementation also uses an output stream that allows to output individual bits by accumulating them and outputting whole bytes when the accumulator is full. The four literal encoders share one such output stream.

The process described in previous paragraph requires that the number of literal bytes is divisible by four. This is ensured before the literals are encoded by adding up to three additional literals (to increase the number to nearest multiply of four). Note that even with the added literals, the original data can still be decoded unambiguously, because the number of preceding literals is kept for each match (as a L in the (L, M, D) triplet).

The FSE encoders for literals have $2^8 = 256$ possible input symbols (i.e. all possible byte values) and they use 1024 states in the reference implementation. The encoding also uses the procedure mentioned in Section 3.2: the literals are encoded backwards starting from the last literal. Because decoding works in the opposite direction, it will start from the first. When all literals are encoded, the final encoder state for each of the four FSE encoders is saved into the header.

The L, M and D elements of match triplets are encoded next. Three separate FSE encoders for each of the elements are used. In each step, the L, M, D elements of one match triplet are encoded and the algorithm then continues with the next match. Similar output stream as was used when encoding literals is also used here to allow outputting individual bits. One such stream is shared by the three FSE encoders.

The L, M and D values may be possibly very large, so the FSE encoders would need to have many symbols and the encoding tables would be huge.

[5]For instance the final state of FSE encoder for M elements is stored as a 16-bit integer in *v_1 header*. However, since it can only take values in range 0–63, such representation is inefficient. It is copied into fewer bits in *v_2 header*.

Instead, each encoder has a fixed number of possible symbols (it is 20 for L and M, 64 for D). There is also a limit on the values L, M and D may take. If either L or M is larger than its maximum value, the match must be split into multiple matches and each of them encoded separately. These maximums are still considerably larger than the number of FSE symbols, for instance it is 2359 for match length (M), while its FSE encoder has only 20 possible symbols. To map the values of L, M and D elements to symbols used by FSE, tables[6] are used. For each possible value, they contain its *base value*. The element is encoded as its base value and the extra bits representing the difference between value of the element and its base value are also written to output. In the reference implementation the extra value bits are written before the extra state bits (produced by FSE). Because lower values are more probable, they generally map to same base value and no extra bits need to be written. For each base value (i.e. symbol), the number of extra bits is stored in another table[7] to be used during decompression.

As in the case of literals, the match triplets are also encoded starting from the last, so that they can be decoded starting from the first. The final state for each of the FSE encoders is also saved into the header when all matches are encoded.

Compressed block structure

In summary, the LZFSE compressed block (with the *v_2 header*) has the following structure:

- It begins with a header that contains the following elements:
  - block magic identifying the type of this block (LZFSE compressed block)
  - number of decoded (output) bytes in block
  - number of literals[8] (i.e. number of bytes taken by literals in unencoded form, will be multiple of 4)
  - size of the encoded literals (i.e. number of bytes taken by literals in encoded form)
  - number of (L, M, D) matches

[6]These tables do not change with input data. They are hard-coded as constant static arrays in the reference implementation (see lzfse_encode_tables.h and lzfse_internal.h).

[7]This is also hard-coded in the reference implementation.

[8]Slightly different terminology is used in the reference implementation. There, all the bytes between the previous and the next emitted match are referred to as one literal. Therefore, as noted in a comment there, this number is not the number of literals defined in this way.

  - three bits representing internal state of literal FSE output stream
  - the final states of the four literal FSE encoders
  - number of bytes used to encode the (L, M, D) matches
  - three bits representing internal state of matches FSE output stream
  - size of the header
  - final states of the L, M and D FSE encoders
  - compressed frequency tables for FSE encoders

- Then it contains all the compressed literals.
- Then it contains the compressed (L, M, D) triplets representing matches.

All fields of the header starting with number of literals are kept in compact form. All fields between number of literals and final states of L, M, D encoders (inclusive) are stored into shared continuous block of memory. The frequency tables for L, M, D and literal encoders are compressed using a form of static Huffman coding.

4.1.2.1 Finite state entropy encoding

The finite state entropy encoding as implemented in the reference implementation of LZFSE is described here (see Section 3.3 for general description).

Before the actual encoding can take place, the encoding table must be initialized. This is done based on the symbol frequencies, which are counted in advance. Using the frequencies of symbols, the states[9] are distributed into subsets (corresponding to symbols). Each symbol is assigned a number of states based on its frequency. Symbols with higher frequency of occurrence will get more states but every symbol that has non-zero frequency is given at least one. For instance, if a symbol has frequency of 15% (i.e. 15% of input symbols are equal to this symbol), then approximately 15% of possible states will be assigned to it.

In this implementation of FSE, the encoder table contains an entry for each possible *symbol*. Each entry contains a value called *delta*. This value is relative increment used to obtain the next state [14]. When a symbol is encoded, the next state is computed by adding *delta* (from the table entry for this symbol) to the current state. The value of *delta* for each symbol depends on the number of states that were assigned to this symbol. The more states the symbol has the lower the *delta*. Thus, symbols with larger frequencies are assigned more states and so they have lower *delta*.

[9]The number of states for FSE encoders is fixed. It is hard-coded for each encoder type (L, M, D and literals) in the reference implementation.

The mechanism to ensure that states remain in a fixed range that was described in Chapter 3 is also used here. When a state would be larger than the maximum state, its k least significant bits are *shifted* – transferred to the output. This k is computed before encoding and saved for each symbol. However, this number also depends on the current state, larger states may require to transfer more bits to the output to keep in the range. Because of this, the first state requiring more bits to be shifted (denoted s_0) is also computed and saved for each symbol in the table. If the current state is smaller than s_0, only its $k-1$ bits are shifted. Otherwise, if the state is larger or equal, its k bits are shifted. Naturally, for both of these variants a different *delta* will be used to get to the next state. Both variants are stored in the table, denoted $delta_0$ and $delta_1$ respectively.

The encoding of a symbol using FSE as it is done in the LZFSE reference implementation is summarised in the following pseudocode:

Algorithm 5: FSE symbol encoding

```
  input  : $symbol$, current state $s$, encoding table $t$
  output: new state $s\prime$
1 if $s < t[symbol].s_0$:
2     $nBits := t[symbol].k - 1$
3     $delta := t[symbol].delta_0$
4 else:
5     $nBits := t[symbol].k$
6     $delta := t[symbol].delta_1$
7 output $nBits$ least significant bits of $s$
8 $s\prime := (s >> nBits) + delta$
```

4.2 Decompression

The decompression is done block by block. The header for the current block is loaded first. As described in Section 4.1, the first four bytes of each block contain a special value identifying type of that block (called block magic). If the block magic is equal to the special value denoting the end of file, the decompression is finished. Otherwise, the header is decoded in a way specific to the particular block type. Information about the current block is obtained from it and saved to be used during decoding.

If the current block is of uncompressed block type, it is decoded simply by copying the block content to the output unchanged. The decompression algorithm then moves on to the next block.

As mentioned in the compression section, there is a fallback algorithm called LZVN, which is used for compressing small files in the reference implementation (for speed reasons). It also has its own block type and its own decompression algorithm, which is used if a block of this type is encountered. As stated before, this algorithm will not be covered here, because this work focuses on the LZFSE algorithm.

There are two types of headers for blocks compressed by LZFSE, which were described in Section 4.1.2. As also mentioned there, only the *v_2 header* (the header with compressed frequency tables) is actually used in practice. The other header type (*v_1*) is only used internally, but the decoder also supports blocks starting with header of that type.

When the compressed header is encountered, it is decoded first and all the information, including frequency tables for FSE, is obtained. Four FSE encoders were used to encode the literals, there also was one encoder for each of the (L, M, D) elements (see Section 4.1.2). For each of these FSE encoders, a corresponding FSE decoder is used during decoding. The final states of all encoders were saved inside the header, from which they are now read and used to initialize all the FSE decoders. As in the case of encoding, one shared decoding table is used by all four literal decoders.

The literals were encoded first and so they are present in the beginning of the block immediately following the header. Thus, all the literals are decoded first using the four FSE decoders, which rotates in decoding the literals. Decoded literals are stored in a buffer to be used later when decoding the matches. Thanks to the literals being encoded in reverse order (starting from the last), they are now decoded starting from the first[10]. This also applies to the L, M, D values decoded later. The finite state entropy decoding is described in a separate section below, see Section 4.2.1.

The matches produced by the encoder in a form of (L, M, D) triplets are present next in the block. The matches are decoded successively in the process described below. The output is not written directly but it is rather kept in a buffer so that the previous output can be copied when decoding the matches. This buffer is referred to as the *output buffer*. The current position in the output buffer is kept and updated when executing the matches (as described further in the text).

For each match, the L, M, D values are decoded first using their respective FSE decoders. As described in Section 4.1.2, the values were encoded as their base value and bits representing the difference between their actual value and base value were written to the output. Therefore, additional bits must be read from the input and added to the base value yielded by FSE decoding step. This is done in the FSE decoder, the number of additional bits for each symbol is known in advance[11] and so it is saved in the decoder entries during the initialization.

[10]As described in Section 3.2, ANS decoding works in opposite direction than encoding.

[11]As described in the backend section of encoder, the tables used to convert values to their base values are hard-coded in the reference implementation.

When the L, M, D values of match are decoded, the match is *executed*. Firstly the L literals from the literal buffer are copied on the current position in the output buffer. The pointer to the literal buffer is then moved by L positions (to point to the next literal that was not yet written to output). Then the M bytes starting D positions before the current position in the output buffer are copied on the current position in the output.

The process of executing matches is similar to how LZ77 decodes the matches. A portion of previous output pointed to by the current match is copied on the current position. In case of LZ77, the previous output is kept in a sliding window, from which symbols are removed and new ones added every step. In case of LZFSE, all output is accumulated in the output buffer. The values are copied within the output buffer and only the pointer to current position needs to be updated, so no costly operation (as is the sliding window management in case of LZ77) needs to be done.

The process of decoding one match is formalized in Algorithm 6, *dst* is pointer to the current position in the output buffer and *lit* is pointer to the literal buffer. Note that the literal buffer contains all literals, which were decoded during initialization. The L, M, D symbols are decoded using their FSE decoder (the FSE decoding is described in Section 4.2.1 below).

This is repeated until all matches are decoded. If the output buffer runs out of space, the decoding is interrupted. When all matches are decoded, the output buffer contains the whole decoded block. Subsequently, the decoder state is updated and the decompression continues with the next block.

Algorithm 6: LZFSE match decoding step

```
1 decode the next $L, M, D$ symbols (using FSE decoders)
2 copy $L$ bytes (literals) from $lit$ to $dst$
3 $dst := dst + L$
4 $lit := lit + L$
5 copy $M$ bytes from $dst$ - $D$ to $dst$
6 $dst := dst + M$
```

4.2.1 Finite state entropy decoding

4.2.1.1 Decoding literals

The decoder for literals is simpler than the L, M, D values decoder (described further in the text). It is an implementation of the finite state entropy decoding. The decoding table is initialized before the decoding may take place. The initialization uses the same frequency table as was used for initializing the encoding table.

The decoding table contains an entry for each state. Each entry must contain an information on how to get to the previous state[12] and the decoded symbol[13]. The decoded *symbol* is simply stored in each table entry.

The decoder table entry also contains values called *delta* and k. The value of k is the number of bits that were transferred to output during encoding to ensure the state is lower than given maximum (see Section 4.1.2.1). During decoding, these k bits must be read from input and added to the *delta* value to get to the previous state. The decoding step is described in the following pseudocode:

Algorithm 7: Decoding of a literal using FSE

```
  input : current state $s$, decoding table $t$
  output: decoded symbol (literal), new state $s\prime$
1 $n$ := read $t[s].k$ bits from input
2 $s\prime := t[s].delta + n$
3 return $t[s].symbol$
```

4.2.1.2 Decoding L, M, D values

The L, M, D values are decoded in a similar way as the literals. The values were encoded as their base values (this was described in Section 4.1.2) and the difference was written to the output. When they are decoded, this difference must be read from input before reading the bits that are added to *delta* to obtain the previous state.

Each entry of the decoding table contains: *delta* (as in the case of literals), the value base (*vbase*), the number of bits that will be read from input (called *total_bits*) and a number indicating how many of these bits represent the difference from the base value (called *value_bits*).

The following pseudocode shows how a L, M or D value is decoded[14]:

Algorithm 8: Decoding of one L, M or D value using FSE

```
  input : current state $s$, decoding table $t$
  output: decoded value ($L$, $M$ or $D$), new state $s\prime$
1 $value\_extra$ := read $t[s].value\_bits$ bits from input
2 $state\_extra$ := read $(t[s].total\_bits - t[s].value\_bits)$ bits from input
3 $s\prime := t[s].delta + state\_extra$
4 return $t[s].vbase + value\_extra$
```

[12]From the decoding viewpoint, this state is actually the next state, but it will be referred to as the previous state to conform to the description of asymmetric numeral systems in Chapter 3.

[13]This is the symbol that was encoded by transitioning to this state during encoding.

[14]Note that in the reference implementation, all the extra bits are read together. They are then separated on state and value bits using bit operations. However, they are read separately in the pseudocode for simplicity reasons.

Chapter **5**

Implementation

The goal of this thesis was to integrate LZFSE method into the ExCom library. This chapter describes the steps taken to achieve it. LZFSE was implemented as a new ExCom module. The process of adding new methods into ExCom is described in detail in [2], only changes from this procedure and important steps will be listed here.

5.1 Implementation of LZFSE module

The LZFSE module was implemented as a new class `CompLZFSE` extending the `CompModule` class from the ExCom library. The source code for this class is present in `lzfse.hpp` and `lzfse.cpp` files.

As there exists an open source reference implementation of LZFSE written in C language (described in previous chapter), it was possible to use this implementation and integrate it into ExCom by writing a wrapper over it. The reference implementation provides an interface for block compression. There are two methods `lzfse_encode_buffer` and `lzfse_decode_buffer` that performs compression and decompression respectively. They take a source buffer as an input and write their output to a given output buffer.

The LZFSE module was implemented as a wrapper over these methods. It performs the required operation (i.e. either compression or decompression) by executing the following steps:

1. Load the input into a source buffer.
2. Allocate an output buffer[15].

[15]Because it is not known in advance how big the output of encode/decode operation will be, it is assumed that for compression the output will not be larger than the input. For decompression, the output buffer size is chosen to be four times the input size. If the output buffer is too small, the operation fails. In such case, the output buffer is enlarged and the operation is run again.

3. Call the appropriate method that will process data from the input buffer and write the result into the output buffer.

4. Write contents of the output buffer as an output of the operation.

The ExCom library API supports reading/writing by bytes or by whole blocks. Initially, the first variant was used and the input was loaded byte by byte using a loop and also written in similar way. However, this proved to be a significant bottleneck. By rewriting the I/O operations using the block functions from the API (`readBlock()` and `writeBlock()`), the compression time was improved (lowered) by nearly 15% in average. The improvement of decompression time was even higher, using block I/O operations decreased it by approximately 40% in average on Prague Corpus files.

The modified LZFSE reference implementation is placed in a subfolder named `lzfse` inside the LZFSE module. The changes made to the code of the reference implementation are listed below in Section 5.1.2.

Furthermore, unit tests were created for the LZFSE module to validate the implementation. They test the compression and decompression operations and other functions of the module.

5.1.1 Implemented parameters

The following adjustable parameters were implemented for the LZFSE module (for details of their implementation see Section 5.1.2):

- *good match (g)* – This parameter sets the threshold for immediately emitting matches. As described in Section 4.1.1 when a found match is bigger than this threshold, it is immediately emitted. Default value of this parameter is 40. Only values equal or larger than 4 are accepted.

- *hash bits (h)* – This parameter controls the number of bits produced by hash function that is used in frontend to search for matches (see Section 4.1.1). Integer values between 10 and 16 (inclusively) are possible, 14 is the default.

5.1.2 Changes in reference implementation

All changes made to the reference implementation are described here. The version of LZFSE reference implementation from 22 May 2017 as published at [14] was used.

As mentioned in Chapter 4, the reference implementation contains a heuristic that uses a different algorithm called LZVN for small inputs. Because the goal was to implement and measure performance of the LZFSE algorithm, the

LZVN compression was removed here by commenting out the corresponding part of the code. While the LZVN compression was removed from the encoding function, the ability to decompress LZVN blocks was preserved in the decoding function to maintain compatibility with the reference implementation[16].

Further changes were necessary in order to implement the adjustable parameters described above. Since these parameters affect only the compression operation, only the encoder was needed to be modified.

The reference implementation contains four adjustable parameters in the `lzfse_tunables.h` header. However, they are implemented as *preprocessor constants* there, so it is not possible to change their value at runtime.

The `LZFSE_ENCODE_GOOD_MATCH` and `LZFSE_ENCODE_HASH_BITS` preprocessor constants were replaced by parameters that are part of structure defined in `lzfse_params.h`. This structure is kept by the LZFSE module and its fields (the parameters) are updated when parameter change is requested through the module API. When the compression operation is run, this structure is passed to the appropriate method and saved inside the compression state. Its fields are then used instead of the constants.

Because the `LZFSE_ENCODE_HASH_BITS` affects the size of the history table, it was also necessary to modify the history table allocation. It is now allocated dynamically with size depending on the parameter value.

The `LZFSE_ENCODE_LZVN_THRESHOLD` constant was used to control the LZVN compression mentioned before. But since LZVN is not used here, this constant is unnecessary and was removed.

The last constant called `LZFSE_ENCODE_HASH_WIDTH` controls the number of entries stored in each set of the history table. Implementing this parameter would require even more complex allocation of the history table and various additional changes in code where the history table is manipulated with. Furthermore, this parameter may take only two values: 4 or 8. Because of this, it was left as a preprocessor constant and may be modified by changing its definition in the `lzfse_tunables.h` header file and recompiling the module.

5.1.3 Testing

Unit tests were implemented for the LZFSE module. The unit tests are similar to unit tests for other modules in ExCom. The functionality of the module is tested: input/output operations, setting of parameters, and finally the compression and decompression operations. The *cxxtest* framework is used for unit tests in ExCom, as described in [2].

[16]Therefore, LZVN compressed block will never be produced by the LZFSE module, but the module is able to decode them (so it can decode files compressed by the LZFSE reference implementation).

Additionally, when running benchmarks, all files from Prague Corpus were always tested if they were decompressed correctly by the method (i.e. whether the decompressed file was identical to the original file).

5.1.4 Other changes

Additional minor changes outside of the LZFSE module were necessary in order to compile the ExCom library using a modern g++ compiler.

Also the explicit call to search buffer object destructor present in LZ77 and LZSS modules was removed, because it was causing double delete (as the object is deleted using `delete` later) and the methods could not be run.

Chapter 6

Benchmarks

The results of benchmarks made on the newly added LZFSE module and other ExCom modules are presented in this chapter. The benchmarks were made using the ExCom library benchmarking and time measurement capabilities as described in [2].

6.1 Methodology

The Prague Corpus files were used for benchmarking, see Appendix C for list of files and file details. Every method was run 50 times for both compression and decompression operations on each file. The minimal time from those 50 runs was taken for each operation to eliminate bias caused by other running processes in the system.

ExCom supports multiple runs of a method on one file, but it does not support computing minimum time or compression ratio [2]. Two *bash* scripts were written to automate the process of measuring performance of methods. Both scripts are part of this work, they are described in Appendix E.

6.2 Testing platform specifications

The ExCom library was built and the benchmarks were made on a system with the following specifications:

Operating system	Fedora 26 (Linux kernel version 4.14.18)
Processor	Intel® Core$^{\mathrm{TM}}$2 Duo CPU T6570 @ 2.10GHz $\times$ 2
Architecture	64 bit
L2 cache size	2048 kB
Memory size (RAM)	4 GB
Compiler	g++ 7.3.1

6.3 Results

6.3.1 Comparison with other compression methods

Because LZFSE was designed as a balanced method and intended for general use cases, it was expected to be among above average methods both in terms of compression ration and speed. Furthermore, its compression ratio would likely benefit from the ANS entropy coder. The benchmarks proved those assumptions.

For all benchmarks presented in this subsection, default values were used for all adjustable LZFSE parameters.

6.3.1.1 Dictionary methods

LZFSE was compared to LZ77, LZ78, LZAP, LZMW, LZW and LZY dictionary methods implemented in ExCom. Unfortunately, most files from Prague Corpus were not decompressed correctly using the LZSS method (i.e. the decompressed file was not identical to the original), and so it was not used in comparisons.

Compression time
Because LZFSE compression operation is relatively complex (as described in Chapter 4) and it is considerably more complex than its decompression operation, it was not expected to be the fastest method in this regard. Nevertheless, LZFSE had the lowest compression time on 23 from 30 tested files and so on most files, it was the fastest from tested methods. On each of the `abbot`, `firewrks` and `nightsht` files, LZFSE had around 50% lower compression time than LZW – the second fastest method on most files. LZFSE generally performed well on larger files, especially on audio and graphic files. On the largest tested file `nightsht`, LZFSE had nearly 53% lower compression time than LZW, the absolute difference being more than 642 milliseconds.

However, LZFSE struggled on very small files, where compression time was negatively impacted by its complex compression algorithm. On the three smallest files `collapse`, `xmlevent` and `usstate`, which are all source codes, LZFSE had significantly worse compression time than LZW. The largest relative difference occurred on the smallest file `collapse`, where LZFSE was almost 2 times slower than LZW (0.313 ms versus 0.167 ms) and was the fourth slowest method. On `xmlevent` and `usstate` files, LZFSE had around 20% larger compression time than LZW. However, apart from `collapse` and `usstate` files, LZFSE was always at worst the second fastest method.

Based on these observations, it seems only logical to use a simpler algorithm instead of LZFSE for compressing very small files. Still, in absolute values, LZFSE was at most by 1.3 milliseconds slower than LZW (on file `modern`, where relative difference was only approx. 6%).

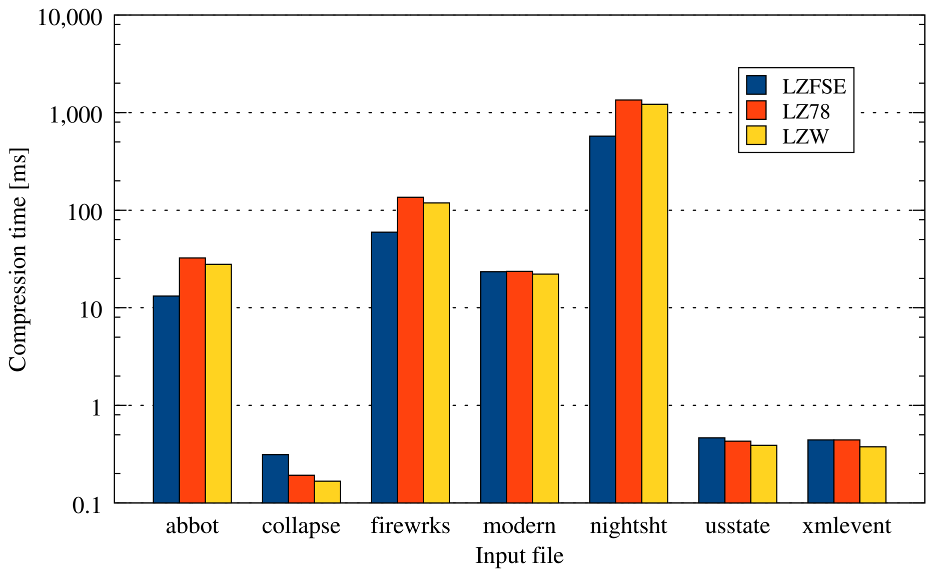

Figure 6.1: Comparison of compression time of LZFSE, LZ78 and LZW dictionary methods

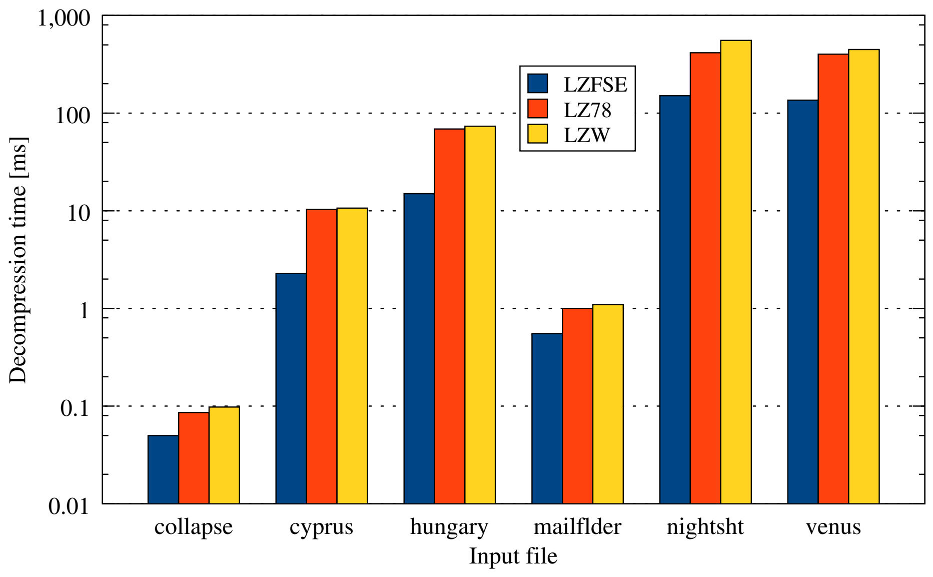

Figure 6.2: Comparison of decompression time of LZFSE, LZ78 and LZW dictionary methods

Decompression time
For decompression operation, LZFSE proved to be significantly faster than all dictionary methods implemented in ExCom that it was compared to. It had the lowest decompression time on all files and on most of them (27 from 30 files) it was more than 60% faster than any other dictionary method.

LZ78 was the dictionary method with fastest decompression so far. LZFSE achieved approximately 65% lower decompression time than LZ78 in average. The only two files where the relative difference was significantly lower than 60% were `collapse` and `mailfldr` text files, where LZFSE had only 42% and 45% respectively lower decompression time than LZ78.

The largest relative difference occurred on `cyprus` and `hungary` files, which contain data in XML format. There, LZFSE had approximately 78% better (lower) decompression time than LZ78. The largest absolute difference was on `nightsht` and `venus` image files. These files took LZFSE approximately 264 milliseconds less than LZ78 to decompress (151.06 ms versus 415.21 ms on `nightsht`), having more than 66% lower decompression time on both.

Compression ratio
In terms of compression ratio, LZFSE proved to be the best from all tested dictionary methods. Thanks to its efficient compressor and robust entropy encoder, LZFSE achieved more than 20% lower compression ratio than the second best method in average. It had the lowest compression ratio on all but one files from corpus and its compression ratio was always lower than 1 (i.e. negative compression never occurred using LZFSE). On `hungary` XML file, LZFSE even had 55% lower compression ratio than the second best dictionary method LZMW. The only file where LZFSE ranked second was `wnvcrdt` database file, where it had 6.31% ratio and LZMW had 5.74%.

The strength of LZFSE compressor was demonstrated on `abbot`, `firewrks` and `nightsht` files, where it was the only method to have compression ratio below 1. On `nightsht` and `firewrks` graphical files, all other tested dictionary methods produced negative compression, as it is hard for them to find matches in these files. However, thanks to its ANS entropy coder, LZFSE had compression ratio of 92% on both files.

The file `abbot` is a ZIP archive and contains data compressed using the DEFLATE algorithm. The contents of this file was originally compressed with 54.3% compression ratio using the said algorithm. While other dictionary methods failed to compress this archive further (each of them had at least 110% compression ratio), LZFSE achieved compression ratio of 91.4%. This indicates that the LZFSE compression method is slightly more sophisticated than the version of DEFLATE used to compress this ZIP archive.

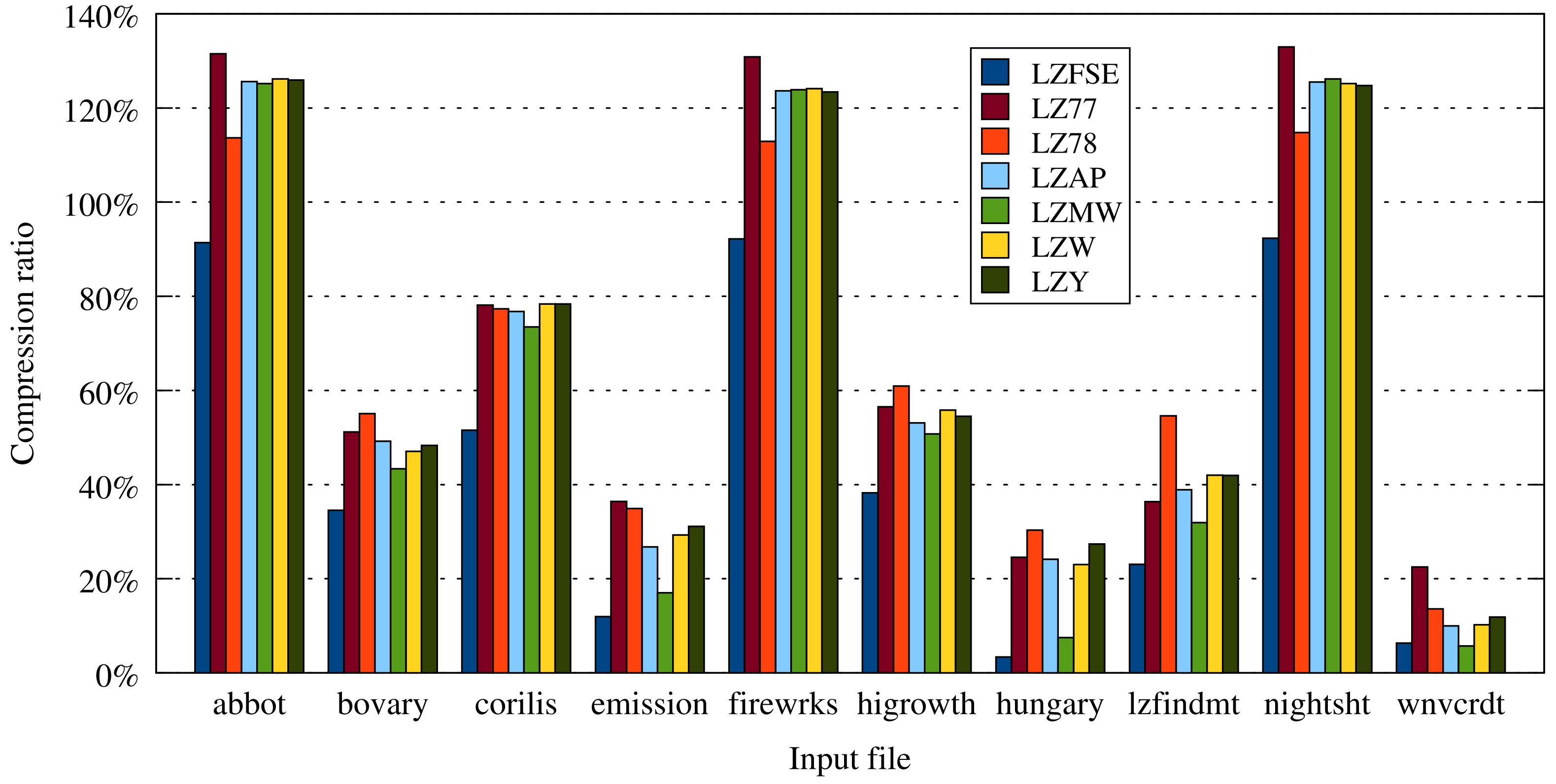

Figure 6.3: Comparison of compression ratio of LZFSE and other dictionary methods

6.3.1.2 Statistical methods

Compression time
In terms of compression speed, the fastest statistical method was static Huffman coding. It was also faster than LZFSE on more than half of corpus files. However, LZFSE was always at worst the second fastest method and it had the lowest compression time on 11 from 30 files from the Prague Corpus.

LZFSE achieved comparable compression time to static Huffman coding in most cases, as the average relative difference was fewer than 5 percent. The biggest relative difference was on `modern` file, where compression time of LZFSE was 64% larger. The biggest absolute difference was, unsurprisingly, on large graphic files. On `flower` and `venus` the difference was substantial – it was more than 200 milliseconds on `flower` and nearly 170 milliseconds for `venus`. In contrast, on `nightsht`, LZFSE had comparable compression time to static Huffman coding and was only approximately one millisecond slower.

LZFSE performed best mainly on database and XML files. It did exceptionally well on `cyprus` and `hungary` XML files, where it had around 40% lower compression time than static Huffman coding.

Decompression time
LZFSE decompression was decisively faster than that of any tested statistical method. It outperformed statistical methods on all files and always had at least 90% lower decompression time than the second fastest method (usually static Huffman coding, Shannon-Fano coding on some files). LZFSE did especially well on `hungary` and `cyprus` XML files, where it was around 36 times faster than the closest contender. Additionally, on `corilis` graphic file, it was more than 33 times faster than the second best static Huffman coding.

This difference is particularly noticeable on large graphic files. On each of the `nightsht`, `flower` and `venus` images, LZFSE had at least 95% lower decompression time than the second fastest method. In case of `nightsht` file, this difference was more than 3 seconds (0.15 s versus 3.48 s static Huffman).

Compression ratio
LZFSE achieved better compression ratio than any statistical method on 29 from 30 files. The only exception was `nightsht` file, where it had compression ratio of 92.3% while arithmetic coding had 91.8%.

On 7 files from corpus, the LZFSE method had at least 70% lower compression ratio than the second best compressing method on that file. On `hungary` XML file, it had 3.38% compression ratio while the closest arithmetic coding had 56.82%. Similarly, on `cyprus` XML file, LZFSE had ratio of 3.98% and arithmetic coding 56.27%, which is more than 14 times larger.

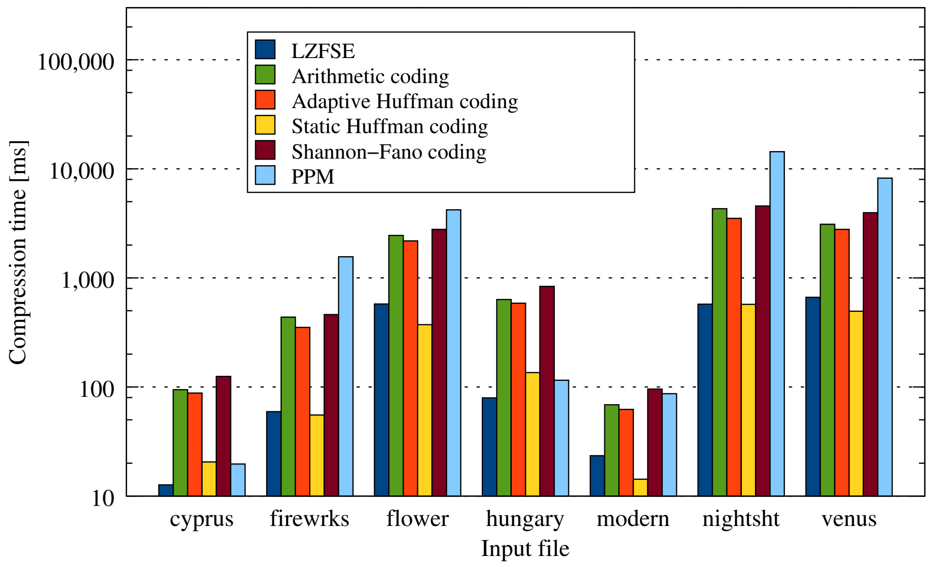

Figure 6.4: Comparison of compression time of LZFSE, statistical methods and contextual method PPM

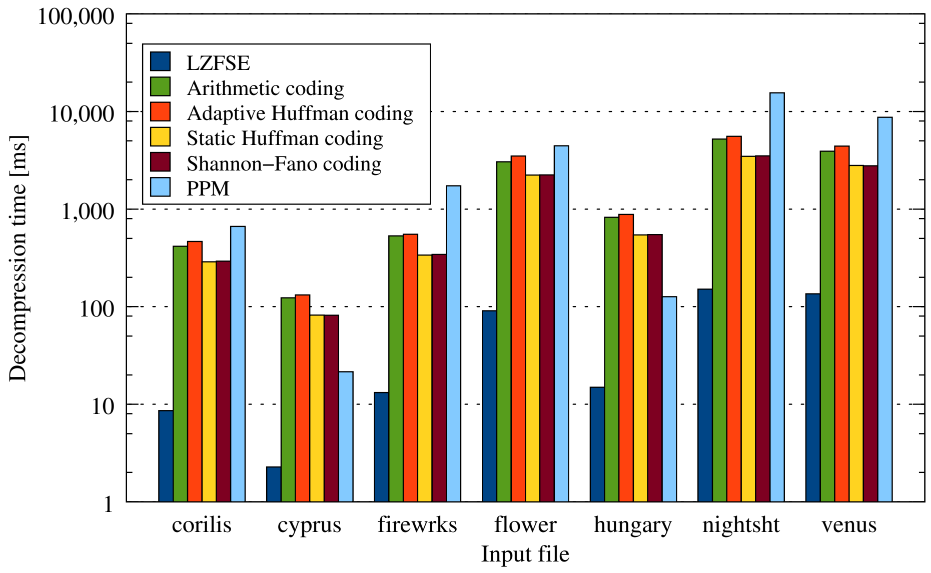

Figure 6.5: Comparison of decompression time of LZFSE, statistical methods and contextual method PPM

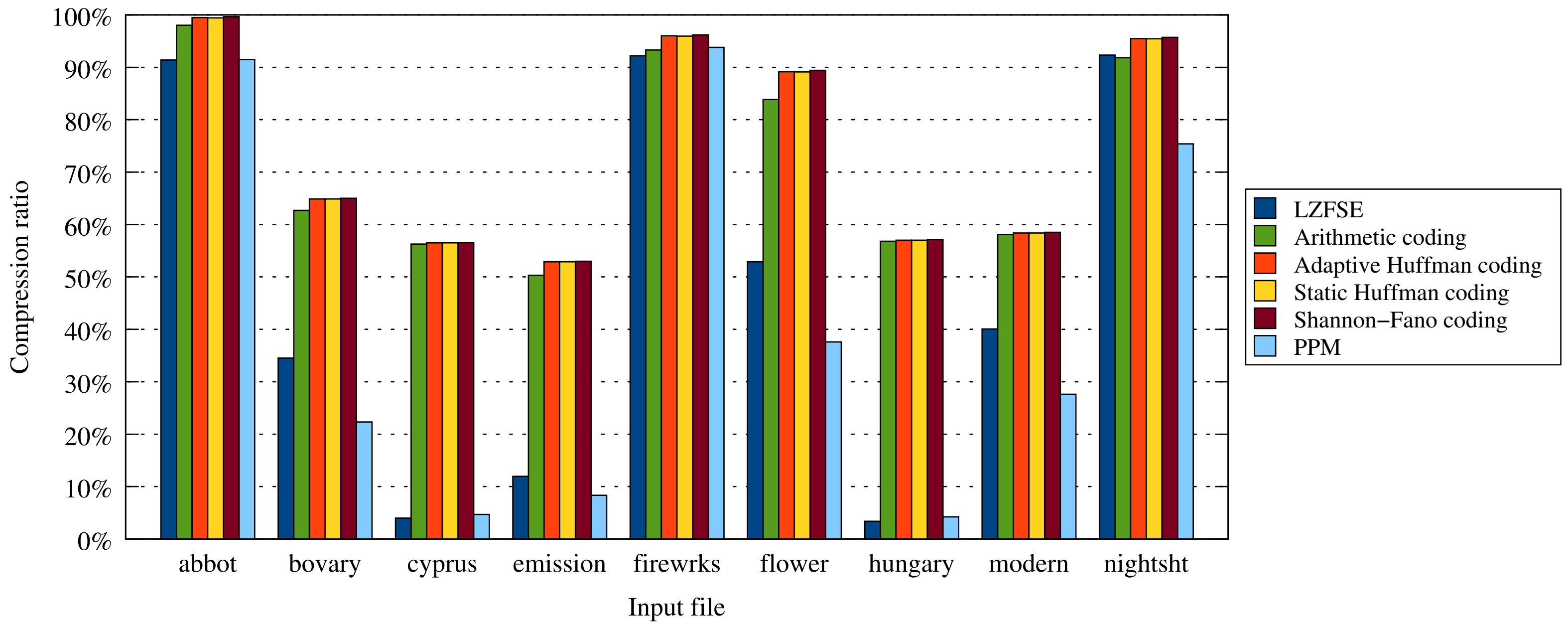

Figure 6.6: Comparison of compression ratio of LZFSE, statistical methods and contextual method PPM

6.3.1.3 Prediction by partial matching

Prediction by partial matching (PPM) from the contextual methods category was generally the best compressing method from all those tested. Excluding LZFSE, it had lower compression ratio than other tested methods on all files except one.

Because compression ratio is not the main focus of the LZFSE method, but rather it seeks balance between compression quality and speed, it could be expected to be outperformed by PPM in this regard. PPM had better compression ratio than LZFSE on 26 from 30 corpus files. The largest difference was on `bovary` PDF document, where LZFSE had almost 55% larger compression ratio than PPM (34.53% ratio versus 22.34% ratio) – and so produced 1.55 times larger compressed file. Also on `emission`, `flower` and `modern` files, LZFSE had more than 1.4 times worse compression ratio. However, apart from these four files, the relative difference between LZFSE and PPM ratios was always lower than 35%. On 22 from 30 corpus files, LZFSE had at most 25% larger compression ratio than PPM.

On four files from the Prague Corpus, LZFSE even achieved better compression ratio than PPM. When compressing the `hungary` file, LZFSE achieved ratio of 3.38%, which is by 20% lower than the 4.23% ratio produced by PPM. Also on `cyprus` file, LZFSE had approximately 15% lower compression ratio (3.98% versus 4.68%). On `abbot` and `firewrks` files, LZFSE had only slightly better compression ratio (by no more than 2%).

However, in terms of compression and decompression speed, LZFSE was substantially better than PPM. LZFSE had lower both compression and decompression time on all files from corpus. On 25 files, LZFSE had at least 65% lower compression time than PPM. The largest difference was on `firewrks` audio file, where LZFSE had more than 26 times smaller compression time than PPM (59 ms versus 1564 ms). On `nightsht` image, the largest file from corpus, PPM compression took 13.68 seconds longer and was 25 times slower (574 ms LZFSE versus 14372 ms PPM).

In case of decompression time, the difference between these two methods was even greater. On all files from corpus, LZFSE had more than 85% lower decompression time than PPM, the average relative difference being approximately 95%. The largest difference was again on the `firewrks` file, where LZFSE had more than 131 times lower decompression time (13 ms versus 1733 ms). The `nightsht` file was decompressed by LZFSE in 151 ms, while the same operation took PPM 15546 ms, which is more than 100 times longer.

In summary, LZFSE usually achieved slightly worse compression ratio but with significantly better speed than PPM. So unless compression ratio is greatly important, LZFSE would almost always be better choice. Figures 6.7 and 6.8 contain comparison of relative performance of all tested methods. From these graphs, LZFSE seems to be best choice for general applications.

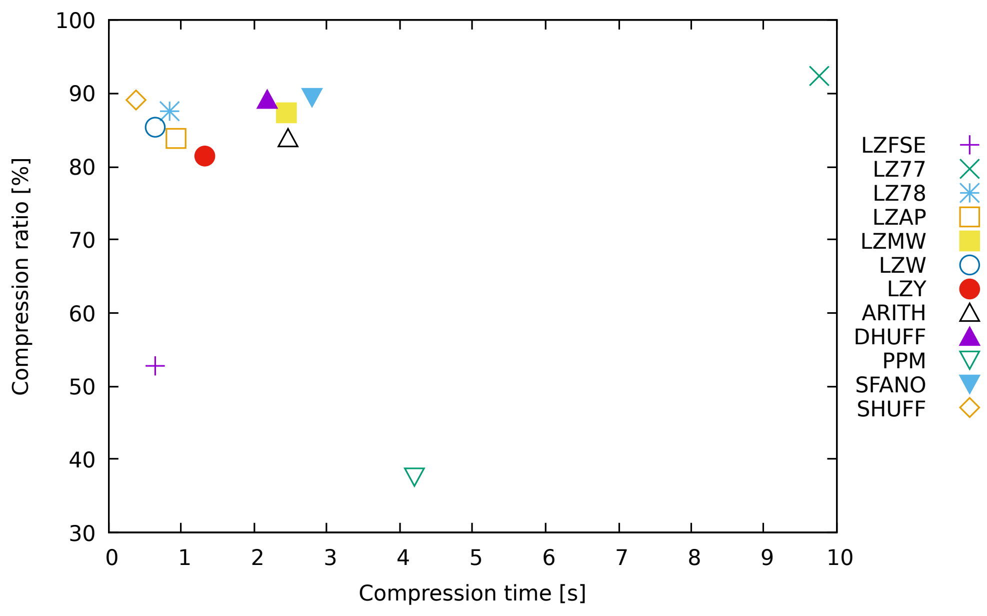

Figure 6.7: Comparison of compression time and compression ratio of all tested methods on `flower` file. This graph shows the tradeoff between compression ratio and speed. Lower values are better for both axis, so methods closer to the origin have better performance.

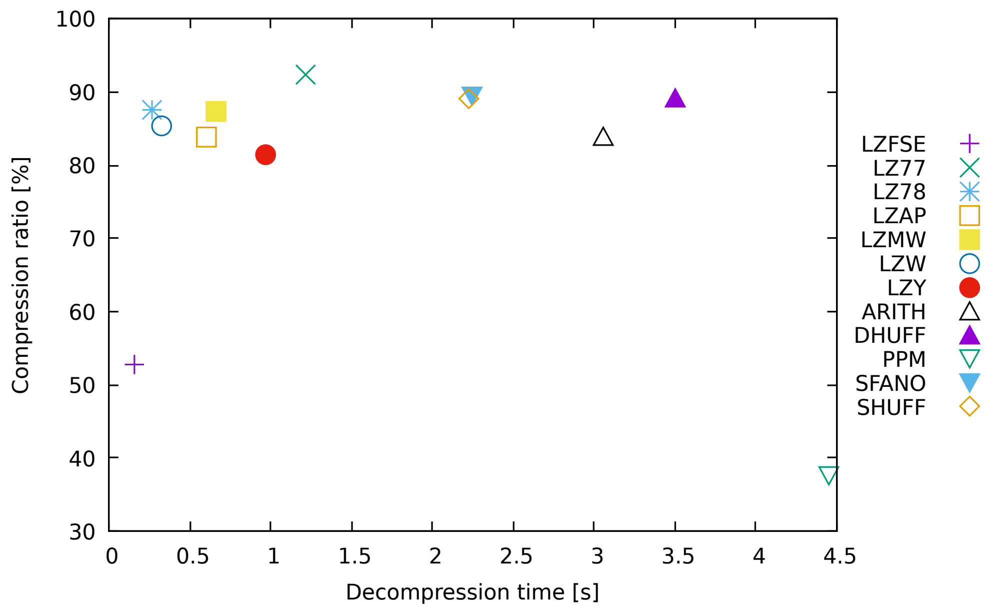

Figure 6.8: Comparison of decompression time and compression ratio of all tested methods on `flower` file

6.3.2 Adjustable parameters

6.3.2.1 The *good match* parameter

As stated before, this parameter controls the minimum size for matches to be emitted immediately in the frontend (see sections 5.1.1 and 4.1.1). Value of this parameter has an impact on compression ratio and speed. Setting it to higher value may yield better compression ratio as the matches are not emitted prematurely and longer matches may be found. However, this also increases the compression time.

Increasing the value of this parameter always slightly increased the compression time, because it prolonged the search for matches in the frontend. However, larger values of this parameter did not always improve the achieved compression ratio. On `nightsht` file, in which it is hard to find matches, the compression ratio was exactly the same for 10 and 100 values of this parameter. Similarly, on `venus` and `flower` image files, only very small values (smaller than 20) of the *good match* parameter produced different compression ratios.

The largest impact on compression ratio was on the `emission` database file, as shown in the graph below. This file contains many matches and some of them are very long. Therefore, when the parameter had lower values, the long matches were not found and the compression ratio was worse (higher).

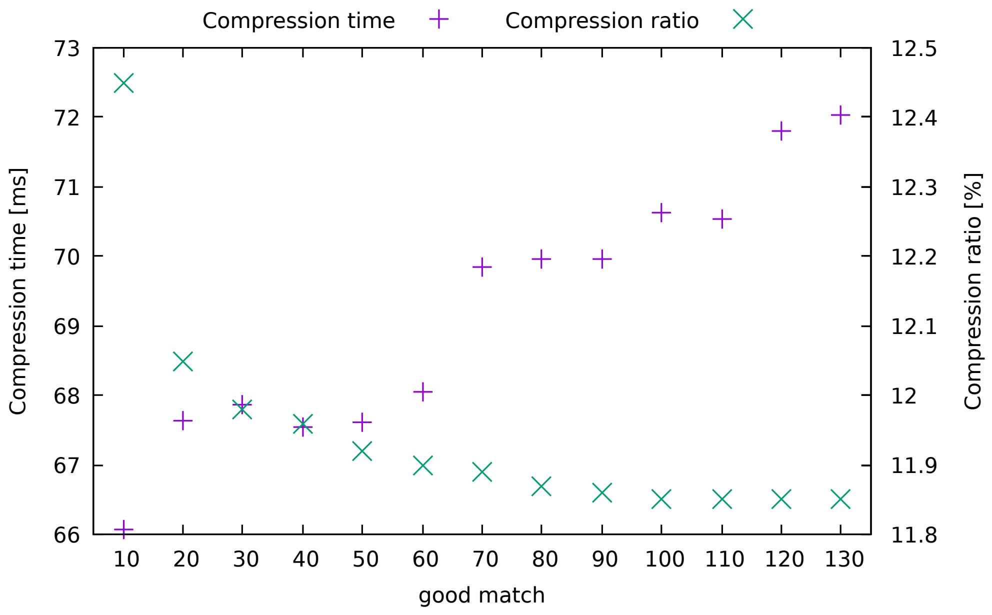

Figure 6.9: The impact of *good match* parameter on compression time and compression ratio on the `emission` file

6.3.2.2 The *hash bits* parameter

This parameter controls the number of bits produced by hash function when searching for matches (see sections 5.1.1 and 4.1.1). Larger values reduce the probability of hash collisions and increase the size of the history table. Because the history table contains a history set for each possible hash value, the size grows exponentially. Increased size allows more match candidates to be present in the history table and potentially improves achieved compression ratio. However, it also causes more cache misses and subsequently larger compression time.

Increasing this parameter always caused an increase in compression time. In case of `nightsht` and `flower` graphic files, the time increased significantly. The impact on compression ratio differed for each file. On `nightsht`, compression ratio remained basically the same (92.15% for 10 vs. 92.23% for 16). In contrast, the compression ratio changed significantly on the `flower` image (see the graph below).

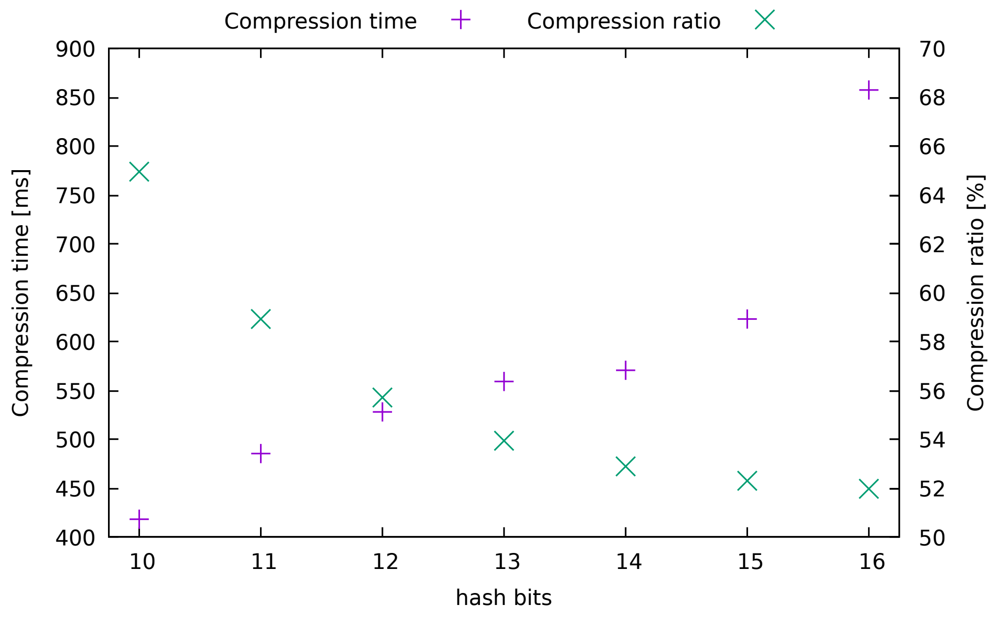

Figure 6.10: The impact of *hash bits* parameter on compression time and compression ratio on the `flower` file

6.3.2.3 The *hash width* parameter

The *hash width* parameter controls the number of entries for each set of the history table. Its value is defined in the `lzfse_tunables.h` header file as a preprocessor constant called `LZFSE_ENCODE_HASH_WIDTH`. This parameter was not implemented for the LZFSE module, so its value can not be controlled through the ExCom API. To change the value of this parameter, its definition was modified and the LZFSE module was recompiled.

Possible values for this parameter are 4 and 8, the default is 4. If it is set to 8, the history table size doubles. Having more match candidates for each position increases the chance of finding (longer) matches [14]. However, similarly to the *hash bits* parameter, this results in more cache misses and slower compression speed.

Benchmarks showed that the impact on compression time is significant. Compression time for *hash bits* set to 8 was in average about 70% larger than for value of 4. Biggest impact was on `collapse` file, where the compression time was 86% larger.

The impact on compression ratio was barely noticeable on most files. Setting the parameter value to 8 decreased compression ratio by approximately 1.4% in average. The largest impact occurred on `hungary` and `cyprus` files[17], where the ratio decreased by 7% and 6% respectively.

6.3.2.4 Impact on decompression time

Only compression time and compression ratio were discussed for all three parameters. All three parameters apply only to the compression operation. Therefore, decompression time is only affected indirectly by parameter settings. Compressing with different parameter values may produce slightly different output.

However, the actual measured impact on decompression time was negligible. It changed by no more than 5% in average for various settings of the three tested parameters.

[17]Both these files are XML files containing air quality monitoring data (see Appendix C).

Conclusion

As part of this thesis, the LZFSE compression method was analysed and then implemented into the ExCom library as a new module. This was done by using the reference LZFSE implementation as a base for the module and modifying it to work inside ExCom and to allow two adjustable LZFSE parameters to be controlled through the ExCom library API.

Benchmarks were then made using the files from the Prague Corpus and utilizing the ExCom library time measurement capabilities. The newly added LZFSE method was compared with other methods implemented in ExCom in terms of speed and compression ratio. The effects that adjustable LZFSE parameters have were also examined.

LZFSE did exceptionally well both in terms of decompression speed and compression ratio and was the best from tested methods in these two categories.

In terms of compression speed, LZFSE was not the fastest method because of its more complex compressor. It was surpassed by static Huffman coding in this regard. However, it was the fastest from dictionary methods on most tested files and it always placed among faster methods.

The decompression of LZFSE was the fastest from all tested methods on all files from corpus. It had considerably lower decompression time than other dictionary methods on most files, usually being substantially faster than any of these methods.

Furthermore, LZFSE achieved the lowest compression ratio on all but one files when compared with dictionary methods and usually outperformed them significantly in this regard. It also surpassed statistical methods in similar way. Moreover, negative compression never occurred when compressing corpus files with LZFSE. The only method that had better compression ratio on most files was a contextual method PPM. However, this method was incredibly slower than LZFSE for both compression and decompression operations.

In summary, LZFSE proved to be a balanced method providing both decent compression speed and excellent speed of decompression. It also achieved very good compression ratio. As such, this method is appropriate for common usage, when the speed and the compression ratio have similar priority.

LZFSE achieves its outstanding performance by combining a fast match-searching scheme, which employs hashing to find matches, and an entropy coder based on the finite state entropy, a variant of asymmetric numeral systems.

Future work

The strength of entropy coding based on the asymmetric numeral systems was demonstrated in the benchmarks. Thanks to it, LZFSE achieved size reduction even on files, where other dictionary methods produced negative compression. It would be interesting to add the different variants of ANS into the ExCom library and compare them with each other to identify the strengths of each variant. And also compare them with other entropy coding methods already implemented in the library.

Furthermore, another lossless algorithm called Zstandard (or Zstd), which uses the finite state entropy variant of ANS, was developed recently by Yann Collet at Facebook. This method is gaining popularity (it is used by Facebook and it is, for instance, already supported in Linux kernel) and it would probably be a valuable addition into the ExCom library. It would be particularly interesting to compare performance of this method with LZFSE and determine which one is better for which situations.

Bibliography

[1] Shannon, C. E. A Mathematical Theory of Communication. *The Bell System Technical Journal*, volume 27, no. 3, July 1948: pp. 379–423, 623–656, ISSN 0005-8580, doi:10.1002/j.1538-7305.1948.tb01338.x.

[2] Šimek, F. *Data compression library.* Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Prague, May 2009.

[3] Wang, R. Shannon's source coding theorem. 2012, accessed: 2018-03-28. Available from: `http://fourier.eng.hmc.edu/e161/lectures/compression/node7.html`

[4] Sayood, K. *Introduction to Data Compression.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., third edition, Dec. 2005, ISBN 0-12-620862-X.

[5] Salomon, D.; Motta, G.; et al. *Data compression: the complete reference.* London: Springer, fourth edition, Dec. 2006, ISBN 1-84628-602-6.

[6] Holub, J.; Řezníček, J.; et al. Lossless Data Compression Testbed: ExCom and Prague Corpus. In *2011 Data Compression Conference*, March 2011, ISSN 1068-0314, pp. 457–457, doi:10.1109/DCC.2011.61.

[7] The ExCom Library. Accessed: 2018-02-12. Available from: `http://www.stringology.org/projects/ExCom/`

[8] Cormen, T. H.; Leiserson, C. E.; et al. *Introduction to Algorithms, Third Edition.* The MIT Press, third edition, 2009, ISBN 0262033844, 9780262033848.

[9] Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, volume 23, no. 3, May 1977: pp. 337–343, ISSN 0018-9448, doi:10.1109/TIT.1977.1055714.

[10] Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, volume 24, no. 5, September 1978: pp. 530–536, ISSN 0018-9448, doi:10.1109/TIT.1978.1055934.

[11] Data Compression - LZ-77. Accessed: 2018-03-30. Available from: `http://www.stringology.org/DataCompression/lz77/index_en.html`

[12] Senthil, S.; Robert, L. Text Compression Algorithms - A Comparative Study. *ICTACT Journal on Communication Technology*, volume 2, Dec. 2011: pp. 444–451, doi:10.21917/ijct.2011.0062.

[13] Duda, J. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *ArXiv e-prints*, Nov. 2013, `arXiv:1311.2540`. Available from: `https://arxiv.org/abs/1311.2540`

[14] Bainville, E. LZFSE compression library and command line tool. June 2016, accessed: 2018-03-30. Available from: `https://github.com/lzfse/lzfse`

[15] 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`

[16] Duda, J.; Tahboub, K.; et al. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium (PCS)*, May 2015, pp. 65–69, doi:10.1109/PCS.2015.7170048.

[17] De Simone, S. Apple Open-Sources its New Compression Algorithm LZFSE. July 2016, accessed: 2018-03-31. Available from: `https://www.infoq.com/news/2016/07/apple-lzfse-lossless-opensource`

[18] Apple Inc. Data Compression — Apple Developer Documentation. Accessed: 2018-03-31. Available from: `https://developer.apple.com/documentation/compression/data_compression`

[19] Řezníček, J. *Corpus for comparing compression methods and an extension of a ExCom library.* Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, May 2010. Available from: `http://www.stringology.org/projects/PragueCorpus/data/papers/Reznicek-MSc_thesis-2010.pdf`

Appendix A

Acronyms

ANS	Asymmetric Numeral Systems
API	Application Programming Interface
DCA	Data Compression using Antidictionaries
ExCom	Extensible Compression Library
FSE	Finite State Entropy
I/O	Input/Output
LZ	Lempel-Ziv
LZAP	Modification of LZMW, AP stands for “all prefixes”
LZFSE	Lempel-Ziv Finite State Entropy
LZMW	Lempel-Ziv-Miller-Wegman
LZSS	Lempel-Ziv-Storer-Szymanski
LZW	Lempel-Ziv-Welch
LZY	Lempel-Ziv-Yabba
PDF	Portable Document Format
PPM	Prediction by Partial Matching
XML	Extensible Markup Language

Appendix B

Reference implementation

The reference implementation of LZFSE is available from GitHub at [14]. It is licensed under the open source 3-Clause BSD License. The file structure of the reference implementation and important functions are described here. All details pertain to the version from 22 May 2017.

B.1 Source files

The source codes are present in the `src` subfolder of the reference implementation. It contains the following files (listed alphabetically):

- `lzfse.h` – This header file contains the declarations of the API functions for both the encode and decode operations. See Section B.2 for details of the API functions.
- `lzfse_decode.c` – The implementation of the decode function declared in the `lzfse.h` file is present here. The actual LZFSE decompression is not implemented in this file. Here, it is only assured that an auxiliary buffer is allocated and an initialization is performed. An internal function called `lzfse_decode` is then invoked to perform the actual decoding.
- `lzfse_decode_base.c` – The `lzfse_decode` function is implemented here. The decoding of the compressed blocks is done here. Functions implemented in the `lzfse_fse.h` and `lzfse_fse.c` files are used for the FSE decoding.
- `lzfse_encode.c` – Similarly to `lzfse_decode.c`, this file contains an implementation of the API function that performs compression. Same as in the case of decompression, initialization is done here and another function is called to do the actual encoding. However, the heuristic that

chooses which function will be called to encode the input is implemented here. If the input is very small, it is copied uncompressed. Otherwise, if it is smaller than a given threshold, it is encoded using LZVN. If it is not, it is encoded using LZFSE by calling the appropriate function implemented in `lzfse_encode_base.c` file.

- `lzfse_encode_base.c` – This source file contains an implementation of the LZFSE compression. It contains both the implementation of the frontend and the backend, both were described in Chapter 4. For the FSE encoding, functions from `lzfse_fse.h` and `lzfse_fse.c` are called.

- `lzfse_encode_tables.h` – This file contains the tables used to map values to their base values in a technique described in Section 4.1.2, which is used in the LZFSE compression backend.

- `lzfse_fse.c` – This file contains an implementation of the finite state entropy.

- `lzfse_fse.h` – This header file contains declarations of the finite state entropy functions (and also an inline implementation of some of them).

- `lzfse_internal.h` – This header file contains definitions for the functions implemented in `lzfse_decode_base.c` and `lzfse_encode_base.c`. It also contains definitions of structures used in other files and preprocessor constants. Unlike the constants in `lzfse_tunables.h`, values defined here should not be changed.

- `lzfse_main.c` – An implementation of the LZFSE command line tool.

- `lzfse_tunables.h` – Adjustable LZFSE parameters are defined in this header file (as preprocessor constants).

- `lzvn_decode_base.c`, `lzvn_decode_base.h`, `lzvn_encode_base.c` and `lzvn_encode_base.h` – These files contain an implementation of the LZVN algorithm, which is used as a fallback for small files.

B.2 The API functions

B.2.1 Encoding

To compress a block of data using LZFSE, a function called `lzfse_encode_buffer` has to be called. This function is declared as:

```
LZFSE_API size_t lzfse_encode_buffer(uint8_t * dst_buffer,
                                     size_t dst_size,
                                     const uint8_t * src_buffer,
                                     size_t src_size,
                                     void * scratch_buffer);
```

The `dst_buffer` is a destination buffer, which must be provided to this function and which will contain the result of the compression operation. The `src_buffer` is an array containing the input block.

The `scratch_buffer` is is an auxiliary buffer, it is a pointer to memory that should be allocated by the caller and passed to the `lzfse_encode_buffer` function. This will be used during compression for storing the encoder state. If it is not provided (i.e. a `NULL` is given), then the function will allocate the needed memory itself. The function `lzfse_encode_scratch_size()` should be called to get the required `scratch_buffer` (i.e. the auxiliary buffer) size.

If the compression succeeds, `lzfse_encode_buffer` function returns the number of bytes written to the destination buffer. If the provided destination buffer is too small or if the compression fails from other reason, zero is returned. The contents of the destination buffer is undefined in such case.

B.2.2 Decoding

LZFSE decompression has a similar interface as the compression described above. The function is declared as:

```
LZFSE_API size_t lzfse_decode_buffer(uint8_t * dst_buffer,
                                     size_t dst_size,
                                     const uint8_t * src_buffer,
                                     size_t src_size,
                                     void * scratch_buffer);
```

The `src_buffer` is the input block and the `dst_buffer` is the output buffer. A scratch buffer should also be passed to this function. To find out the required size, the `lzfse_decode_scratch_size()` function should be called.

Unlike `lzfse_encode_buffer`, this function always returns the number of bytes written to the output. Even if the entire output will not fit into the output buffer, it will contain a portion of decoded data.

Appendix C

Prague Corpus files

Table C.1: The Prague Corpus files [19]

File	Size [Bytes]	Description	Type
firewrks	1,440,054	Sound of fireworks	Audio
thunder	3,172,048	Sound of thunder	Audio
drkonqi	111,056	KDE crash handler	Binary
libc06	48,120	A dynamic-link library	Binary
mirror	90,968	A part of the software package	Binary
abbot	349,055	Part of interior design application	Binary
gtkprint	37,560	A shared object	Binary
wnvcrdt	328,550	A database file	Database
w01vett	1,381,141	A database file	Database
emission	2,498,560	Waterbase emissions data	Database
bovary	2,202,291	Gustave Flaubert: *Madame Bovary*, in German	Documents
modern	388,909	Axel Lundegård, Ernst Ahlgren: *Modern. En berättelse*, in Swedish	Documents
ultima	1,073,079	Mack Reynolds: *Ultima Thule*, in English	Documents
lusiadas	625,664	Lúıs Vaz de Camões: *Os Luśıadas*, in Portuguese	Documents
venus	13,432,142	Ultraviolet image of Venus' clouds	Graphics
nightsht	14,751,763	A photo of a city at night	Graphics
flower	10,287,665	A photo of a flower	Graphics
corilis	1,262,483	CORILIS land cover data	Graphics
cyprus	555,986	Air Quality Monitoring in Cyprus	Markup languages
hungary	3,705,107	Air Quality Monitoring in Hungary	Markup languages
compress	111,646	Wikipedia page about data compression	Markup languages
lzfindmt	22,922	C source code from a file archiver	Scripts
render	15,984	C++ source code from an action game	Scripts
handler	11,873	Java source code from the GPS tracking system	Scripts
usstate	8,251	Java source code from the GPS tracking system	Scripts
collapse	2,871	JavaScript source code from the project management framework	Scripts
xmlevent	7,542	PHP source code from the calenedar generator	Scripts
mailflder	43,732	Python source code from the ECM framework	Scripts
age	137,216	Age structure in the world	Spreadsheets
higrowth	129,536	Financial calculations	Spreadsheets
Total	58,233,774		

Appendix D

Building the ExCom library

The process of building the ExCom library with all its modules is described in detail in [2, Appendix E].

To build the ExCom library on Unix system, execute the following commands in shell from the `excom` directory:

```
autoreconf -i -s -f
mkdir bin
cd bin
../configure -C --enable-perf-measure
make
```

Additionally, to generate the Doxygen documentation, run:

```
make doxygen-doc
```

The compiled testing application is located in `bin/src/app` directory, execute `./app -h` for its usage.

Unit tests

Unit tests require the Python interpreter to be installed. To enable unit tests for the library and its modules, add the `--enable-test-cases` parameter to the configure command when building the library:

```
../configure -C --enable-perf-measure --enable-test-cases
```

The tests may then be run by invoking:

```
make tests-run
```

Appendix E

Scripts used for benchmarking

Two *bash* scripts were developed to automate the process of running multiple methods on given corpus files and measuring their performance.

The first script called `benchmarks.sh` was used to compare performance of implemented compression methods with each other. The second script called `benchmarks_params.sh` was used to measure impact of different settings of LZFSE method adjustable parameters.

These two scripts perform compression and decompression multiple times on all given files and measure the minimal time taken by each operation. Compression ratio is also computed for each file. Usage of these scripts is described below. Both scripts are present on the CD attached to this thesis.

Script `benchmarks.sh`

This script will run compression and decompression of given methods on all corpus files and measure the time taken by these operations and resulting compression ratio. Each operation is run multiple times (default is 50) and the minimum from measured times is taken.

This script produces three files in the folder from which it is run. The `benchmarks_comp.csv` and `benchmarks_decomp.csv` contain compression and decompression times measured in microseconds. Compression ratios are computed as percentage (by default) and output into `benchmarks_ratio.csv` file. All three files are *csv* files containing data in form of comma-separated values. If files with these names already exist in the folder, they will be overwritten.

The script should be called from a directory containing both the ExCom testing application (`app`) and a folder containing corpus files (`PragueCorpus` by default). If it is run elsewhere, locations of these files must be given as arguments (see below).

Usage

`./benchmarks.sh [options] [-l method...]`

Possible options are:

-h	Show help.
-r T	Each operation will be repeated *T* times. Default is 50.
-f	Do not exit on failure. All files are tested for equality with the original when they are decompressed. Whenever a method fails to compress and decompress a file correctly, the benchmarks are stopped by default. This option disables this behaviour, a '?' is written into results instead of compression ratio and the benchmarking continues.
-s, -b	These flags control how much information is output during the process. The `-b` flag means *brief* and disables messages for individual files. The `-s` flag stands for *silent* and disables all output.
-d DIR	The benchmarks will be performed on all files from `DIR` directory. Default is `PragueCorpus`. If the corpus files are on different location, it must be given with this parameter.
-c APP	Relative path to the ExCom testing application. Default is `app`.
-n	Disables using percentage for compression ratio.
-l	If this option is used, a list of methods must be given as script arguments following the options. The benchmarks will be run for given methods. Methods must be given by the names ExCom uses for them, use `./app -m?` for list of possible methods.

Example

```
./benchmarks.sh -r 30 -d ~/PragueCorpus -l lzfse lz78 shuff
```

This command will perform benchmarks on LZFSE, LZ78 and static Huffman coding methods. It will run each operation 30 times for each file from the PragueCorpus folder in user home directory and save results in the folder from where it is run.

Script `benchmarks_params.sh`

This script is similar to `benchmarks.sh` but measures the impact of different parameter settings on method speed and compression ratio. This script produces same files as `benchmarks.sh` script.

Usage

```
./benchmarks_params.sh [options] method param min max
```

Four arguments must be given after the optional arguments:

method The name of the method to run benchmarks for. Use `./app -m?` for list.

param Name of the parameter. Use `./app -m method -p?` for list.

min, max These two arguments define the range of tested values of the parameter. By default all parameter values from $[min, max]$ range (inclusive) are tested.

All options available for `benchmarks.sh` (with the exception of `-l` and `-f`) can also be used here with the same effect. Additional options are:

-i INC Parameter value increment. First tested value of parameter is defined by *min*, then the value is incremented by `INC` every step until it is larger than *max*.

-a Apply parameters to both compression and decompression operations. The parameter is used *only* for compression by default.

Example

```
./benchmarks_params.sh -i 10 lzfse g 10 130
```

This command will perform benchmarks on LZFSE and measure the impact of *good match* parameter on compression/decompression times and compression ratio. Values $10, 20, \ldots, 130$ will be tested.

Appendix F

Detailed benchmark results

F.1 Tables

Table F.1: The measured compression time of dictionary methods on the files from the Prague Corpus

	Compression time [ms]						
File	LZFSE	LZ77	LZ78	LZAP	LZMW	LZW	LZY
abbot	13.20	210.09	32.44	42.71	90.94	27.86	46.18
age	6.97	343.09	7.84	9.99	30.56	7.40	16.35
bovary	109.28	3145.75	120.02	163.26	474.50	111.81	251.18
collapse	0.31	1.25	0.19	0.23	0.63	0.17	0.35
compress	4.04	73.49	5.74	7.02	22.32	5.21	11.21
corilis	45.86	765.06	86.05	104.68	284.98	71.49	130.23
cyprus	12.65	219.23	18.67	25.73	96.67	17.53	42.40
drkonqi	5.09	243.70	6.22	7.49	23.41	5.52	12.49
emission	68.70	1463.37	98.45	118.85	456.74	92.19	209.19
firewrks	59.45	1604.06	135.44	181.85	384.62	118.65	209.53
flower	576.02	9766.73	834.11	922.71	2435.40	638.22	1316.52
gtkprint	1.69	100.95	1.85	2.25	7.75	1.63	3.77
handler	0.60	9.31	0.63	0.75	2.44	0.57	1.18
higrowth	5.95	338.03	7.80	9.37	28.31	7.15	15.42
hungary	79.52	1563.44	130.30	176.64	643.19	123.30	289.45
libc06	2.34	113.08	3.09	4.37	11.57	2.97	7.55
lusiadas	28.33	1305.54	30.17	41.38	132.27	28.78	67.74
lzfindmt	1.05	17.12	1.35	1.46	4.76	1.13	2.34
mailflder	1.89	70.45	2.24	2.75	9.04	2.03	4.46
mirror	4.33	113.42	5.39	6.43	19.58	4.68	10.36
modern	23.44	648.67	23.62	31.82	89.29	22.11	51.30
nightsht	573.53	13360.66	1348.75	1897.11	3970.51	1215.88	2158.22
render	0.80	13.95	0.93	1.05	3.41	0.79	1.70
thunder	143.46	13411.37	241.70	334.75	806.45	216.63	464.75
ultima	46.15	797.51	79.88	108.78	258.19	71.01	135.04
usstate	0.47	7.57	0.43	0.51	1.72	0.39	0.80
venus	663.14	19600.65	1056.07	1395.64	3371.38	926.30	1803.15
wnvcrdt	7.69	473.05	8.00	12.20	57.50	8.37	20.11
w01vett	30.21	2069.62	36.00	51.54	241.25	36.10	81.51
xmlevent	0.44	4.03	0.44	0.50	1.57	0.38	0.77

Table F.2: The measured decompression time of dictionary methods on the files from the Prague Corpus

	Decompression time [ms]						
File	LZFSE	LZ77	LZ78	LZAP	LZMW	LZW	LZY
abbot	2.95	35.55	9.78	26.87	29.76	13.41	30.48
age	1.22	17.55	3.17	7.02	7.93	3.98	12.41
bovary	16.06	284.30	54.21	102.63	115.64	59.17	192.07
collapse	0.05	0.38	0.09	0.15	0.17	0.10	0.26
compress	0.83	14.87	2.65	4.74	5.24	2.94	8.70
corilis	8.60	151.69	30.34	68.17	76.98	36.90	96.30
cyprus	2.28	75.25	10.35	18.24	18.71	10.64	34.16
drkonqi	0.99	14.53	2.60	5.12	5.94	3.09	9.83
emission	17.50	332.99	47.88	82.20	87.27	53.35	165.89
firewrks	13.19	145.45	40.79	112.87	124.26	53.88	140.66
flower	90.74	1213.86	264.15	599.39	667.27	330.64	972.41
gtkprint	0.26	5.05	0.84	1.56	1.79	0.98	3.12
handler	0.09	1.64	0.30	0.50	0.59	0.31	0.89
higrowth	0.89	16.76	3.28	6.38	7.19	3.77	11.69
hungary	14.93	502.07	68.83	121.20	116.57	73.31	229.62
libc06	0.36	6.21	1.11	2.77	3.24	1.31	5.80
lusiadas	4.02	81.38	12.48	26.83	31.86	14.63	52.35
lzfindmt	0.25	3.17	0.60	0.97	1.12	0.63	1.83
mailflder	0.55	5.90	1.00	1.84	2.22	1.09	3.45
mirror	0.81	11.79	2.21	4.41	5.21	2.60	8.19
modern	2.98	49.66	9.31	20.23	23.18	10.74	39.31
nightsht	151.06	1504.05	415.21	1168.21	1285.17	554.34	1446.11
render	0.12	2.15	0.41	0.70	0.80	0.43	1.33
thunder	28.99	360.54	81.41	209.96	234.00	100.61	342.98
ultima	8.43	123.64	27.75	69.58	75.82	35.95	94.76
usstate	0.08	1.14	0.21	0.35	0.40	0.23	0.63
venus	135.47	1503.16	400.34	874.96	992.97	447.70	1285.37
wnvcrdt	1.45	45.07	4.94	8.87	11.64	5.14	16.62
w01vett	7.76	188.18	21.09	36.87	41.34	22.35	66.88
xmlevent	0.07	1.03	0.20	0.33	0.36	0.21	0.60

Table F.3: The measured compression ratio of dictionary methods on the files from the Prague Corpus

	Compression ratio [%]						
File	LZFSE	LZ77	LZ78	LZAP	LZMW	LZW	LZY
abbot	91.41	131.52	113.68	125.60	125.20	126.16	125.93
age	43.89	58.40	54.99	53.46	51.20	55.15	54.64
bovary	34.53	51.19	55.07	49.21	43.34	47.08	48.31
collapse	43.08	58.72	70.32	50.71	51.89	56.67	51.51
compress	19.31	37.14	51.77	40.49	30.40	42.19	43.70
corilis	51.58	78.11	77.35	76.76	73.51	78.36	78.36
cyprus	3.98	23.67	27.80	23.08	6.36	20.78	26.30
drkonqi	36.03	48.58	54.68	46.07	44.19	48.37	47.33
emission	11.96	36.44	34.91	26.75	17.02	29.28	31.10
firewrks	92.21	130.87	112.91	123.66	123.87	124.09	123.39
flower	52.89	92.31	87.45	83.91	87.36	85.41	81.46
gtkprint	31.66	44.92	47.30	36.95	36.93	40.83	38.49
handler	25.32	40.35	50.13	38.37	31.71	39.50	40.73
higrowth	38.24	56.52	60.95	53.09	50.75	55.82	54.51
hungary	3.38	24.58	30.34	24.13	7.51	23.03	27.38
libc06	35.06	62.56	51.99	46.73	45.13	47.81	46.85
lusiadas	32.49	50.90	47.54	42.15	41.17	41.27	41.23
lzfindmt	23.07	36.37	54.62	38.89	31.93	42.01	41.97
mailflder	23.13	38.03	46.76	38.54	30.74	36.96	40.36
mirror	40.18	55.98	58.46	51.05	48.38	53.19	51.64
modern	40.08	61.27	64.10	56.70	53.62	52.83	54.94
nightsht	92.33	132.99	114.77	125.52	126.19	125.20	124.78
render	25.81	39.73	54.04	42.64	32.93	43.53	44.63
thunder	75.87	98.48	91.06	96.35	96.14	93.61	94.32
ultima	65.72	96.03	87.24	91.28	87.47	90.05	90.79
usstate	26.28	40.86	48.83	37.52	31.97	39.30	39.43
venus	75.13	106.17	92.70	95.99	97.20	94.15	93.66
wnvcrdt	6.31	22.49	13.60	9.99	5.74	10.21	11.87
w01vett	5.75	22.45	16.11	11.04	7.78	12.20	13.19
xmlevent	29.66	45.30	57.33	40.05	36.75	45.33	42.76

Table F.4: The measured compression time of statistical methods and PPM on the files from the Prague Corpus

	Compression time [ms]					
File	LZFSE	Arithmetic coding	Adaptive Huffman coding	Static Huffman coding	Shannon-Fano coding	PPM
abbot	13.20	114.69	93.11	13.12	112.22	327.53
age	6.97	26.54	28.34	6.05	34.62	58.59
bovary	109.28	410.21	391.19	83.41	589.91	396.59
collapse	0.31	0.62	0.69	0.22	0.97	0.76
compress	4.04	21.32	20.48	4.68	30.36	13.38
corilis	45.86	331.49	294.55	48.28	403.55	609.80
cyprus	12.65	94.46	87.86	20.52	125.18	19.67
drkonqi	5.09	21.76	24.72	4.89	29.95	30.82
emission	68.70	398.25	400.56	89.31	514.02	230.40
firewrks	59.45	436.39	351.40	55.32	461.67	1564.29
flower	576.02	2453.48	2184.70	372.31	2789.61	4206.57
gtkprint	1.69	6.58	7.76	1.79	10.12	8.68
handler	0.60	2.15	2.29	0.61	3.07	1.92
higrowth	5.95	26.39	28.51	5.67	35.16	42.01
hungary	79.52	632.66	585.16	135.53	836.26	115.12
libc06	2.34	8.82	10.08	2.29	14.74	13.52
lusiadas	28.33	96.96	103.25	24.05	145.48	144.15
lzfindmt	1.05	4.44	4.69	1.06	6.18	3.10
mailflder	1.89	7.16	6.99	1.88	9.77	5.80
mirror	4.33	18.07	20.79	4.11	24.95	28.90
modern	23.44	68.65	62.37	14.25	95.79	86.69
nightsht	573.53	4312.98	3512.18	572.33	4561.62	14371.82
render	0.80	3.06	3.24	0.77	4.26	2.51
thunder	143.46	818.57	690.55	124.30	907.97	2957.50
ultima	46.15	285.58	248.74	41.93	326.58	632.79
usstate	0.47	1.50	1.58	0.48	2.15	1.40
venus	663.14	3102.40	2782.17	494.11	3957.87	8231.30
wnvcrdt	7.69	32.44	25.87	11.08	41.30	12.60
w01vett	30.21	150.49	126.29	45.82	206.96	59.56
xmlevent	0.44	1.49	1.67	0.44	2.28	1.47

Table F.5: The measured decompression time of statistical methods and PPM on the files from the Prague Corpus

File	Decompression time [ms]					
	LZFSE	Arithmetic coding	Adaptive Huffman coding	Static Huffman coding	Shannon-Fano coding	PPM
abbot	2.95	140.40	145.33	82.44	84.24	368.65
age	1.22	34.06	40.19	22.32	22.39	64.06
bovary	16.06	529.34	596.27	369.10	368.35	423.12
collapse	0.05	0.78	0.96	0.54	0.57	0.85
compress	0.83	27.48	31.30	19.27	19.41	14.63
corilis	8.60	416.10	466.36	288.37	292.29	664.85
cyprus	2.28	123.21	131.90	81.88	81.58	21.53
drkonqi	0.99	27.66	35.36	19.37	19.67	33.85
emission	17.50	548.80	583.52	353.17	355.52	248.29
firewrks	13.19	531.71	552.90	338.11	343.15	1732.61
flower	90.74	3053.32	3499.23	2228.08	2241.07	4454.61
gtkprint	0.26	8.55	10.73	5.45	5.61	9.65
handler	0.09	2.75	3.36	1.91	1.93	2.15
higrowth	0.89	33.72	41.22	23.06	23.58	46.27
hungary	14.93	822.54	880.18	542.89	548.21	126.35
libc06	0.36	11.37	14.56	8.18	8.24	15.00
lusiadas	4.02	129.29	153.67	92.85	94.58	157.51
lzfindmt	0.25	5.77	6.75	4.08	4.05	3.52
mailflder	0.55	9.35	10.26	6.05	6.21	6.36
mirror	0.81	23.37	29.64	16.13	16.15	32.20
modern	2.98	89.52	95.16	59.82	60.07	92.71
nightsht	151.06	5227.51	5558.35	3477.31	3515.17	15545.83
render	0.12	3.89	4.69	2.71	2.67	2.83
thunder	28.99	1018.62	1081.85	643.89	657.24	3169.79
ultima	8.43	354.91	393.15	238.56	236.84	715.67
usstate	0.08	1.90	2.30	1.30	1.32	1.56
venus	135.47	3910.76	4419.54	2798.94	2779.44	8736.13
wnvcrdt	1.45	48.84	37.31	21.98	21.68	14.29
w01vett	7.76	216.64	188.68	112.84	114.05	65.95
xmlevent	0.07	1.94	2.36	1.35	1.38	1.63

Table F.6: The measured compression ratio of statistical methods and PPM on the files from the Prague Corpus

	Compression ratio [%]					
File	LZFSE	Arithmetic coding	Adaptive Huffman coding	Static Huffman coding	Shannon-Fano coding	PPM
abbot	91.41	98.03	99.50	99.43	99.66	91.50
age	43.89	59.43	62.75	62.69	63.19	36.98
bovary	34.53	62.70	64.88	64.87	65.03	22.34
collapse	43.08	67.53	66.00	66.07	66.38	34.06
compress	19.31	66.30	67.19	67.16	67.30	15.76
corilis	51.58	89.11	94.18	94.17	94.63	45.30
cyprus	3.98	56.27	56.52	56.51	56.55	4.68
drkonqi	36.03	64.45	69.38	69.30	69.62	30.54
emission	11.96	50.33	52.91	52.90	53.00	8.32
firewrks	92.21	93.30	96.00	95.95	96.17	93.81
flower	52.89	83.87	89.14	89.13	89.42	37.61
gtkprint	31.66	54.10	55.67	55.54	55.62	26.49
handler	25.32	60.16	59.93	59.82	59.93	20.72
higrowth	38.24	65.55	69.51	69.44	69.89	34.31
hungary	3.38	56.82	57.00	57.00	57.11	4.23
libc06	35.06	59.74	64.80	64.78	64.85	30.43
lusiadas	32.49	45.73	58.07	58.05	58.22	25.66
lzfindmt	23.07	66.22	66.39	66.33	66.45	17.62
mailflder	23.13	52.33	53.05	53.00	53.20	18.00
mirror	40.18	65.03	69.31	69.22	69.67	34.76
modern	40.08	58.07	58.37	58.37	58.53	27.63
nightsht	92.33	91.84	95.48	95.46	95.73	75.40
render	25.81	62.82	62.67	62.58	62.85	20.96
thunder	75.87	76.49	79.98	79.96	80.36	67.07
ultima	65.72	84.26	90.35	90.32	90.43	59.98
usstate	26.28	57.02	56.74	56.62	56.74	22.17
venus	75.13	77.75	84.37	84.37	84.59	62.66
wnvcrdt	6.31	21.54	23.39	23.38	23.41	5.51
w01vett	5.75	27.91	29.69	29.68	29.88	5.26
xmlevent	29.66	66.50	66.13	66.05	66.20	23.78

F.2 Additional graphs

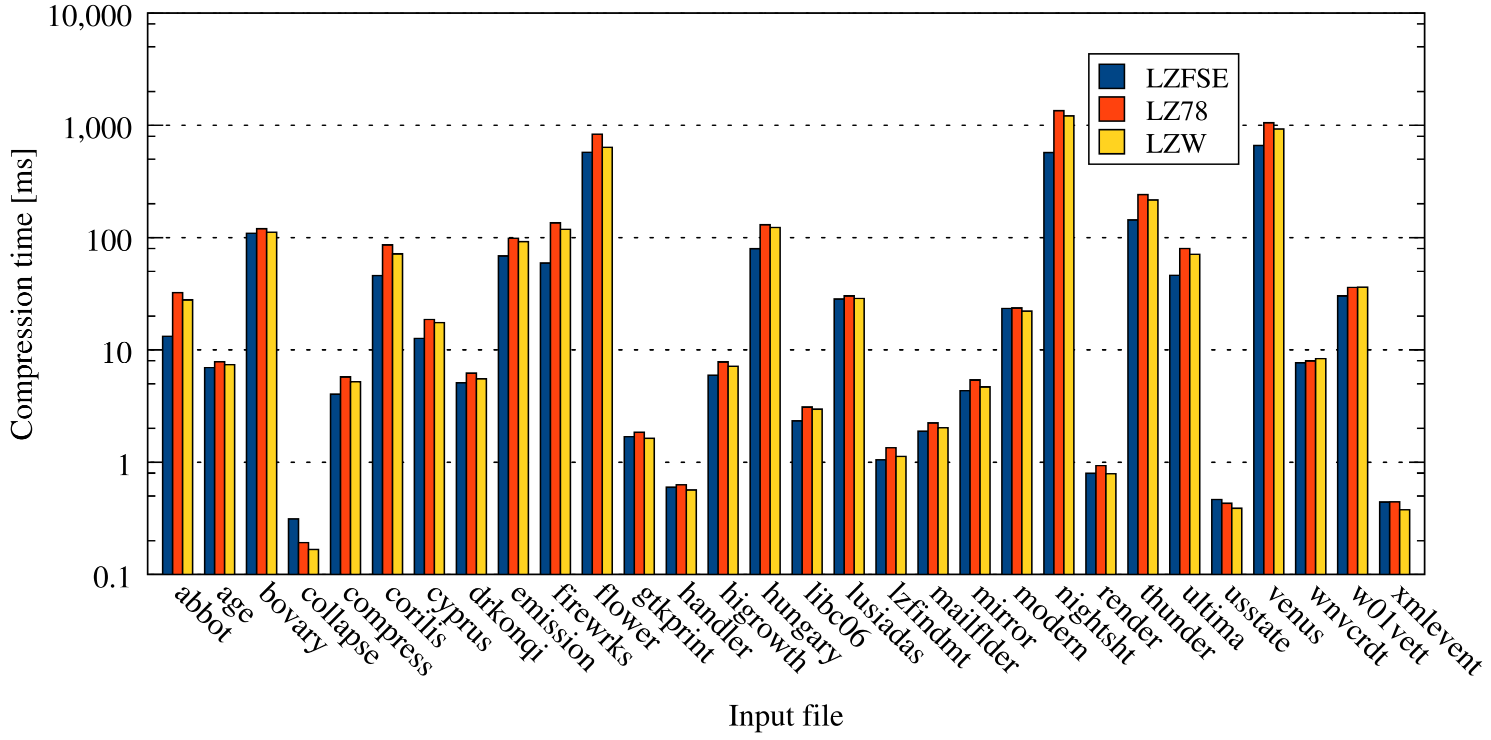

Figure F.1: Comparison of compression time of LZFSE, LZ78 and LZW dictionary methods on all files from the Prague Corpus

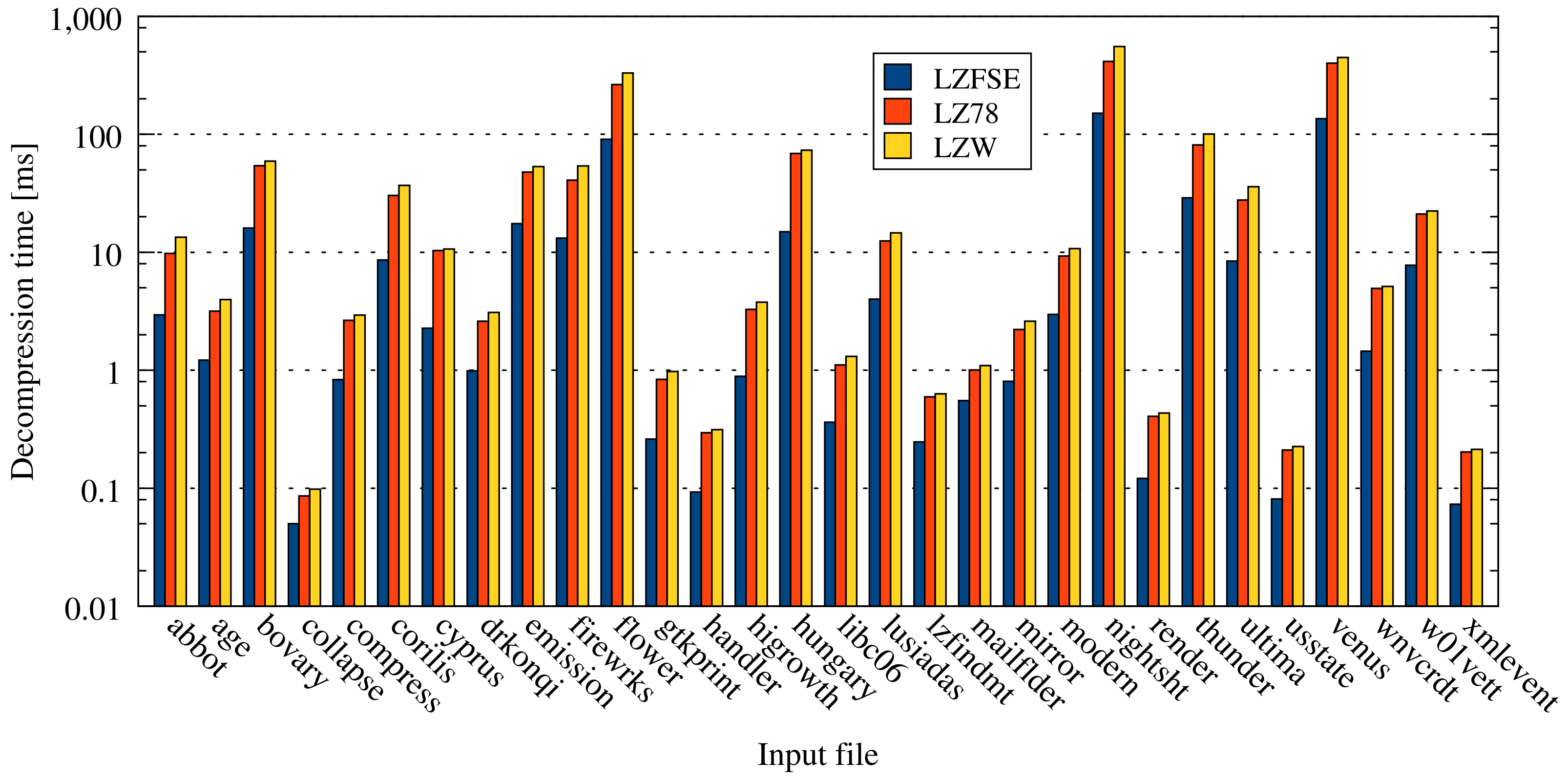

Figure F.2: Comparison of decompression time of LZFSE, LZ78 and LZW dictionary methods on all files from the Prague Corpus

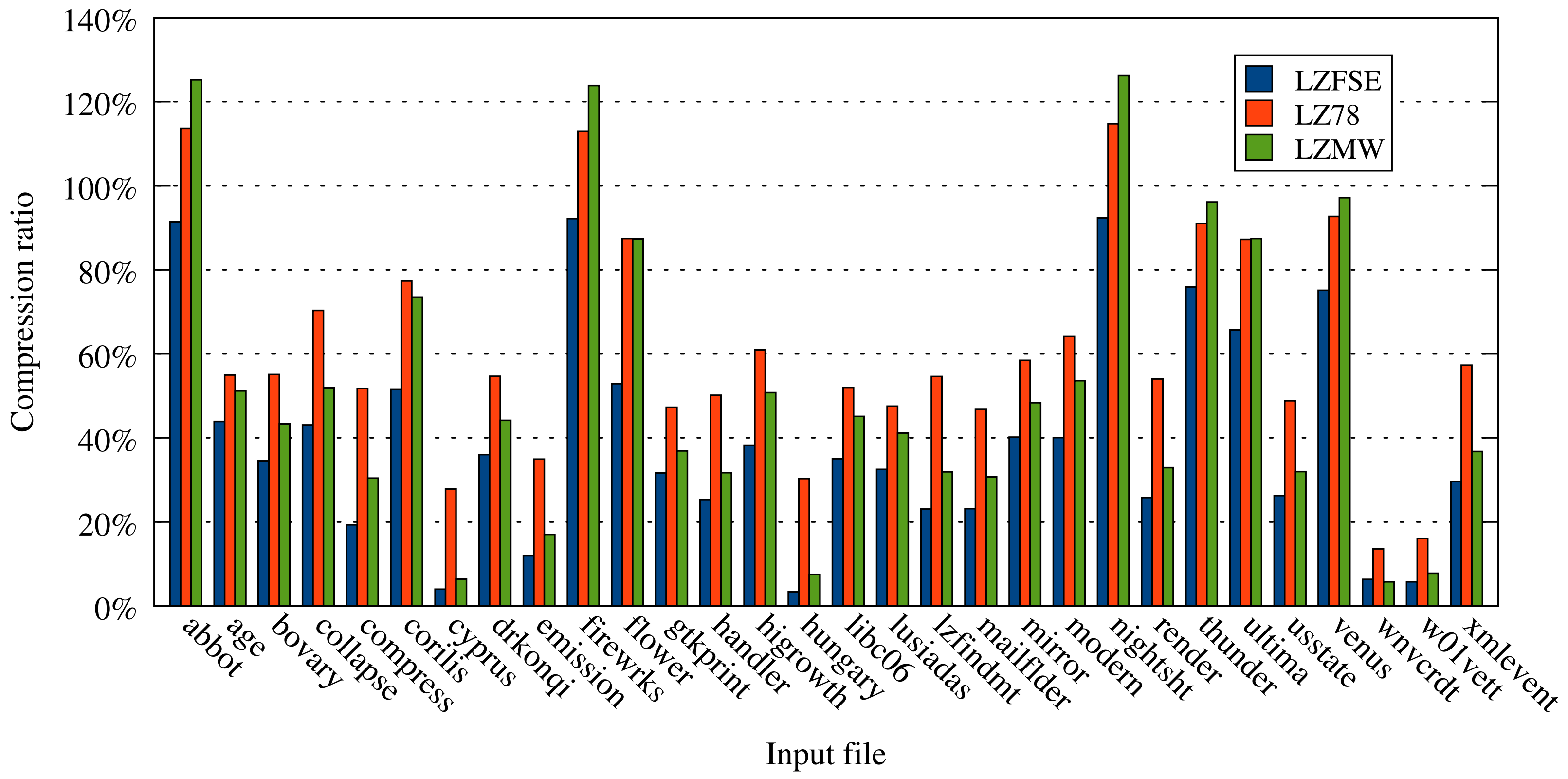

Figure F.3: Comparison of compression ratio of LZFSE, LZ78 and LZMW dictionary methods on all files from the Prague Corpus

Appendix G

Contents of enclosed CD

```
/
├── excom ......... sources of the ExCom library with the LZFSE module
├── PragueCorpus ..... the directory containing files of the Prague Corpus
├── readme.txt .................... the file with CD contents description
├── scripts .......... directory containing scripts used for benchmarking
└── text ..................................... the thesis text directory
    ├── src .............. the directory of LaTeX source codes of the thesis
    └── thesis.pdf ....................... the thesis text in PDF format
```