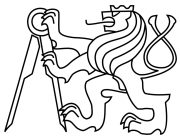
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Compression of natural Czech text
Student:	Bc. Jan Navara
Supervisor:	prof. Ing. Jan Holub, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Design an algorithm for lossless compression of natural Czech text using lemmatiser, morphological tagger and morphological generator. Make a survey of suitable open-source tools and select one. Implement the algorithm in the C++ programming language. Perform experiments with various ways of utilization of lemmas and tags in the compression process and evaluate the results of the experiments.

References

Will be provided by the supervisor.

L.S.

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

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 17, 2016

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

Master's thesis

Compression of natural Czech text

Bc. Jan Navara

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

30th June 2016

Acknowledgements

I would like to thank my friends and family for all support, my supervisor, prof. Ing. Jan Holub, Ph.D., for frequent consultations and all guidance, and also my employer for giving me as many days off as I needed to finish this thesis.

Declaration

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

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 30th June 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Navara, Jan. *Compression of natural Czech text.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

V rámci práce byly navrženy a implementovány 4 algoritmy pro bezeztrátovou kompresi přirozeného českého textu využívající lemmatizátor, morfologický tagger a morfologický generátor obsažený v softwaru MorphoDiTa. První z algoritmů je založen na ukládání lemmat jednotlivých slov a generování slovních tvarů, druhý modeluje pravděpodobnosti jednotlivých slov na základě uložené informace o jejich slovním druhu, třetí modeluje pravděpodobnosti slov na základě slovního druhu předchozího slova a čtvrtý je rozšířením prvního, kdy jsou společně s lemmaty ukládány i některé části tagu. Výsledné kompresní poměry byly porovnány s kompresními poměry dosaženými referenčním algoritmem základní slovní komprese. Z porovnání vyplynulo, že druhý a třetí popsaný algoritmus zlepšení nepřináší, zatímco první a čtvrtý algoritmus dokáže kompresní poměr při vhodných konfiguracích zlepšit, typicky v řádu desetin procenta. Hlavním přínosem práce je důkaz, že použití lingvistických nástrojů může být při kompresi českých textů z hlediska kompresního poměru výhodné. Významným vedlejším produktem práce je široce použitelná implementace adaptivní verze kompresního algoritmu PPM, na níž jsou výše popsané algoritmy založeny.

Klíčová slova komprese textu, lemmatizace, morfologické tagování, morfologické generování, PPM, aritmetické kódování, MorphoDiTa

Abstract

In scope of this thesis, 4 algorithms for lossless compression of natural Czech text have been designed and implemented. The algorithms utilize the lemmatiser, morphological tagger and morphological generator contained in open-source software MorphoDiTa. First of the algorithms is based on storing lemmas of individual words and generating the word forms, the second one uses stored part-of-speech tags to estimate probability of words, the third one estimates probability of a word using part-of-speech tag of the previous word and the fourth one is an extension of the first algorithm, storing some parts of the tag alongside the lemma. The achieved compression ratios have been compared to compression ratios achieved by a basic word-based reference algorithm. The comparison has shown that the second and the third algorithm are not better than the reference algorithm in terms of compression ratio, while the first and the fourth algorithm are able to achieve better compression ratio than the reference algorithm when using an appropriate configuration (they typically improve the compression ratio by several tenths of percent). This thesis thus proved that using linguistic tools in compression of natural Czech texts may be beneficial in terms of compression ratio. An important by-product of this thesis is a highly universal implementation of adaptive PPM compression algorithm, which has been used as the core element of each of the above-mentioned algorithms.

Keywords text compression, lemmatisation, morphological tagging, morphological generation, PPM, arithmetic coding, MorphoDiTa

Contents

List of Figures

List of Tables

Introduction

To compress a natural text with a good compression ratio, the compression algorithm can utilize the fact that a lot of information contained in the text is determined by characteristics of the language. From all possible character sequences of length N, some of the sequences occur much more frequently than some other sequences in the natural text written in a specific language. The same can be said about the individual characters. Of course, we can also see the text as a sequence of words, not just as a sequence of characters; we can again state that some words or sequences of words occur more frequently than other words or sequences of words, respectively. Words can also be classified into several groups called *parts of speech*, where each part of speech has a different role in the sentence. Moreover, words typically have various grammatical categories; especially in highly inflective languages, the grammatical categories determine the specific form of a word which appears in the text, and again, there are more or less strong rules for where a grammatical category (or a specific value of grammatical category) occurs more likely and where it occurs less likely.

In our compression algorithms designed and implemented within this thesis, we are utilizing all of these features. However, our main task is to focus on parts of speech and grammatical categories; we want to show that the compression of natural Czech text can be improved using these properties of words.

From all the characteristics of Czech language which may influence the compression ratio of our algorithms, we want to point out the following two:

- Czech is a highly inflective language. Especially for nouns, adjectives, pronouns, numbers and verbs it holds true that a specific word can appear in many different forms, depending on the values of grammatical categories. This means that by a naive compression algorithm, different forms of the same words may be recognized as different words, which may negatively influence the quality of the compression model.

- The word order is relatively loose in Czech language; this may be a challenge for compression algorithms which are utilizing word order, order of part-of-speech tags in the sentence etc.

Linguistic tools (lemmatiser, morphological tagger and morphological generator) help us to process the text and to get the necessary linguistic information about the individual words. We are focusing on compression ratio, compression time is not our priority. We are testing several ways of using the linguistic information to show which ways are worth developing further and which are probably not helpful. According to our best knowledge, most of the algorithms implemented in this thesis have not yet been implemented by anyone else or the results have not been published (at least for texts written in Czech language).

Chapter 1

Preliminaries

1.1 Linguistic definitions

This section explains some key linguistic terms which are used in this thesis. Examples are shown later (when presenting the selected linguistic tool). As the definitions of some of these terms vary depending on the source of the definition, some definitions have been adjusted a little bit to match our needs in this thesis (and to match the real functionality of the selected linguistic tool); this includes the definition of morphological tag which is valid at least as long as we are using Czech morphological system by Jan Hajič [1].

Definition 1. Lemma of a given word is a canonical form of this word.[1] The process of converting a word form to its lemma is called **lemmatisation** in the rest of this thesis.

Definition 2. In scope of this thesis, a **morphological tag** (or simply a **tag**) is an information about part-of-speech and possibly other grammatical categories of a word form. The process of assigning a tag to a given word form is called **tagging** in the rest of this thesis.

Definition 3. In scope of this thesis, **morphological generation** is the process of generating all possible word forms from a given lemma or from a pair lemma + tag.

We analogously use terms **lemmatiser**, **morphological tagger** (or simply **tagger**) and **morphological generator** (or simply **generator**) in the rest of the thesis.

1.2 Survey to find a suitable linguistic tool

This section describes the initial survey which has been done to find a suitable open-source tool as required by the assignment. We are searching for a tool

or a combination of tools which works with natural Czech text and is capable of lemmatisation, morphological tagging and morphological generation. We prefer tools with high quality of lemmatisation and tagging since we suppose that more exact information about the text will give us chance to achieve better compression ratio. We also want that it is easy to use the tool from within our C++ code.

Sources [2], [3] and [4] proved to be good starting points for this survey, though not all of the discovered tools have been found using these sources.

1.2.1 MorphoDiTa

MorphoDiTa [5] is an open-source tool capable of morphological analysis, morphological generation, tagging and tokenization. It uses linguistic models which are distributed under CC-BY-NC-SA license.

The tool is available as a standalone tool, as a library (including a C++ version of the library) or as a web service.[6] An online demo version is available in [7].

The publication [6] compares the tool to other Czech taggers, Featurama and Morče, and states that "MorphoDiTa reaches state-of-the-art results for Czech and nearly state-of-the-art results for English." and that "The results are very similar for the three Czech systems, Morče, Featurama and MorphoDiTa, ... However, MorphoDiTa is the first end-to-end application released under a free license." The publication also states that MorphoDiTa is efficient and lightweight, showing comparison of speed and resource consumption against the other two tools. That makes MorphoDiTa a suitable tool for our natural language compression task; however, the survey continues.

1.2.2 Czech Morphological Analyzer

This is an online tool by Jan Hajič, available in [8], capable of lemmatisation and tagging. The results of lemmatisation and tagging are displayed on a separate web page. This tool doesn't seem to contain any morphological generator and doesn't seem to be open-source, it could be used only via the online interface.

1.2.3 Flect

A tool available from [9]. It is only a morphological generator, written in Python.

1.2.4 Morče

Morče is a software for tagging a Czech text.[10] The software is available under GPL license (both binaries and source code).[11] However, we have already

found a slightly better tool which, unlike Morče, contains all functionality we need, so we are not going to analyse Morče deeper.

1.2.5 Featurama

Featurama is a C++/Perl open-source tagging software available from [12]. We already know that MorphoDiTa's tagging and lemmatisation results are the same (if not better) as for Featurama and that MorphoDiTa is significantly faster and more space-efficient,[6] so we provide no detailed information.

1.2.6 LemmaGen

LemmaGen is a fast and open-source lemmatisation tool available in C++.[13] This tool is not capable of tagging and morphological generation and the accuracy of lemmatisation (using an online service available in [14]) is obviously not very good.

1.2.7 Ajka and Majka

Ajka is a program which, for a given word form, determines its lemma, part-of-speech and other grammatical categories.[15] Its newer version is called Majka.[15] According to [16], Majka is a more accurate and faster tool than Ajka. According to download page [17], Majka is written in C++, the source code of Majka is licensed under GPL and the tool is very fast (it is able to process approximately one million words per second — faster than MorphoDiTa, where the fastest models allow to process approx. 200 000 words per second[1]). The tags used by Majka [18] are different from those which are used in MorphoDiTa [19].

1.2.8 Conclusion

We decided to use MorphoDiTa, especially for the following reasons:

- We know that is has high lemmatisation and tagging accuracy.
- It is the only found open-source tool combining all three functionalities we need. We could still combine two tools instead of using just this one but that seems impractical (we could, e.g., try to combine Majka's lemmatiser and tagger with MorphoDiTa's generator, but we would have to deal with differences between those two tools — some reasonable conversion between different tag definitions of these two projects would be required etc.).
- It is one of the only two found tools capable of morphological generation, the other one is Flect. If there was a good reason to do so, we probably would be able[20] to use Python code of Flect in our C++ project and

thus use the generator from Flect instead of the one from MorphoDiTa; however, working with C++ code of MorphoDiTa should be simpler and again, merging two different tools could be challenging.

1.3 About MorphoDiTa

The main features of MorphoDiTa have already been presented, now we take a closer look at what MorphoDiTa functionality could be useful for us. We use MorphoDiTa version 1.3.0, which was the latest version of the software at the start of this thesis. This MorphoDiTa version can be downloaded from [21] (binaries or source files). Since the online manual [1] and API reference [22] are changing with time (probably to match the most recent version of MorphoDiTa), we usually refer to manual [23] downloaded with the version 1.3.0.

1.3.1 MorphoDiTa models

As already mentioned, MorphoDiTa works with trained linguistic models which are available separately from the tool. The newest models (version 160310) are available in [24], older models (version 131112) are available in [25]; the newest models have been released when this thesis was already in progress. As apparent after unpacking the downloaded archives (and after reading the attached readme files), both sets of models contain a part-of-speech-only variant (producing tags containing only part-of-speech info[1]) of the tagger. The newer version also contains a no-diacritical-marks variant (working with text not containing diacritics[1]). The older version contains two versions of the main tagger — a version focusing on accuracy and a version focusing on speed.

The proposed tagger speeds and accuracies are summarized in Table 1.1, all info taken from the attached readme files; it is necessary to mention that version 160310 was trained and tested on different data than version 131112. In this thesis, we use only "full" taggers (131112-best_accuracy, 131112-fast and 160310-main) since we work with texts containing diacritics and we are not really focusing on program speed. Moreover, if not stated otherwise, we are using model 131112-fast.

1.3.2 Lemmas

This section summarizes important details about lemmas used in MorphoDiTa. An explicit example of lemmas is shown later in this thesis (together with a sample tagger output).

A lemma in MorphoDiTa is a string consisting of the following three parts:[23]

Table 1.1: Summary of MorphoDiTa taggers

version	tagger name	tag accuracy	lemma accuracy	overall accuracy	speed [words/s]
131112	best_accuracy	95.67 %	97.78 %	94.97 %	10k
131112	fast	94.70 %	97.64 %	93.94 %	60k
131112	pos_only	99.20 %	97.64 %	97.60 %	200k
160310	main	95.57 %	97.75 %	94.93 %	10k
160310	pos_only	99.04 %	97.62 %	97.56 %	200k
160310	no_dia	94.74 %	97.05 %	93.83 %	5k
160310	no_dia-pos_only	98.59 %	97.04 %	96.96 %	130k

- *raw lemma* — shortest text form of the lemma;
- *lemma id* — raw lemma and lemma id together provide a unique identifier of the lemma;
- *lemma comments* — some additional comments, not needed to identify the lemma.

An important fact is that MorphoDiTa contains methods which allow us to determine the boundaries between the three parts of a lemma and thus possibly cut the lemma to a shorter form.[23] For practical reasons, in the rest of this thesis, we are using the following terms:

- *full lemma* is a lemma containing all three parts (raw lemma + lemma id + lemma comments),
- the term *raw lemma* keeps its original meaning,
- *lemma id* is a full lemma without lemma comments (raw lemma + lemma id according to the overridden definition).

A lemma returned by a tagger (which, in MorphoDiTa, serves as lemmatiser as well) is always a full lemma. For morphological generation, we can use either raw lemma or lemma id (lemma comments are ignored).[23]

1.3.3 Tags

This section summarizes important details about tags used in MorphoDiTa. An explicit example of tags is shown in Section 1.3.4, together with a sample tagger output.

As already mentioned, MorphoDiTa uses Czech morphological system by Jan Hajič where the tags are positional with 15 positions representing e.g. part of speech and various grammatical categories.[23] The following detailed info is taken from [19] and [26].

Table 1.2: Brief summary of MorphoDiTa tags

position	description
1	Part of speech
2	Detailed part of speech
3	Gender
4	Number
5	Case
6	Possessor's gender
7	Possessor's number
8	Person
9	Tense
10	Degree of comparison
11	Negation
12	Voice
13	Reserve
14	Reserve
15	Variant, style

Each of the 15 positions has (typically) several possible values. Every value is encoded using one ASCII character and every position contains only one value per tag, thus the tag is a string of 15 ASCII characters. Some values represent a set of other values and not all combinations of values are possible within one tag. Table 1.2 shows a brief summary of what each position represents. Of course, not all grammatical categories are applicable for all words; in such cases, value "-" appears on the corresponding position in the tag to mark that the grammatical category is not applicable.

1.3.4 Tagging (includes lemmatisation)

In this section, we first show how tagging works using the downloaded MorphoDiTa executable. This is just for simplicity; a short presentation of corresponding methods from MorphoDiTa API follows, since we would like to use MorphoDiTa rather as a library. We use a similar approach in the next section as well.

1.3.4.1 Tagger executable

The input of the tagger is an UTF-8-encoded text (there is no need to preprocess the text, MorphoDiTa is able to tokenize it).[23] Tagger requires path to a tagger model as a parameter[23] (tagger models have been described in one of the previous sections). There are also other options which can be used, but they are not described here.

Here we present a simple example of tagger output. We run here a downloaded MorphoDiTa binary `bin-linux64/run_tagger` which can be found in the downloaded binaries archive. One of the previously described models (taggers) is used as parameter:

command:

```
echo "Mé vznášedlo je plné úhořů."
   | ./run_tagger czech-morfflex-pdt-131112.tagger-fast
```

output (line breaks added):

```
Loading tagger:  done
<sentence>
<token lemma="můj_^(přivlast.)" tag="PSNS1-S1------1">Mé</token>
<token lemma="vznášedlo" tag="NNNS1-----A----">vznášedlo</token>
<token lemma="být" tag="VB-S---3P-AA---">je</token>
<token lemma="plný" tag="AANS1----1A----">plné</token>
<token lemma="úhoř" tag="NNMP2-----A----">úhořů</token>
<token lemma="." tag="Z:-------------">.</token>
</sentence>
Tagging done, in 0.000 seconds.
```

We can see that the tagger assigned a lemma and a tag to each word (token). An important detail is that the capital starting letter in the first word has been lost during lemmatisation; we have to be aware of this in our lossless compression algorithm since we cannot alter the case of any letter.

1.3.4.2 Tagger API

The tagger is represented by class `tagger`.[23] An instance of tagger can be created by method

```
static tagger* tagger::load (const char* fname)
```

where `fname` is name of the model, or by method

```
static tagger* tagger::load (FILE* f)
```

where `f` is C file pointer to an opened file with the model.[23] Tagging can be performed using method

```
virtual void tagger::tag (
   const std::vector<string_piece>& forms,
   std::vector<tagged_lemma>& tags
) const
```

where `forms` is a vector of input tokens and `tags` is a vector holding the result

of tagging.[23] There is also a method returning tokenizer instance, which works with UTF-8 encoding and which we can use to tokenize an untokenized text.[23] It's obvious that we can use here all functionality that has been shown with the executable example.

1.3.5 Morphological generation

1.3.5.1 Generator executable

The input of morphological generation is again in UTF-8 encoding.[23] Each line of the input must contain a lemma, optionally followed by a tab and a tag (the tag may contain wildcards).[23] Tag wildcard is a special replacement of a value in the tag which can be used to filter the results of morphological generation; `?` matches any character applicable for the corresponding position in the tag, `[chars]` matches any of the characters listed and `[^chars]` matches any of the characters not listed.[23]

The morphological generator requires the following parameters:[23]

- path to a morphology model; we are allowed to use a tagger model here if option `--from_tagger` is specified;
- an integer value (0 or 1) specifying whether guesser mode should be used or not (according to API reference, when guesser mode is set to 1, generator tries to guess unknown words[23]).

Here we present a simple example of morphological generation output. We use our previously generated output of the tagger as input of the generator and we use the morphology model associated with the tagger:

command:

```
printf
"můj_^(přivlast.)\tPSNS1-S1------1\n
vznášedlo\tNNNS1-----A----\n
být\tVB-S---3P-AA---\n
plný\tAANS1----1A----\n
úhoř\tNNMP2-----A----\n
.\tZ:-------------"
| ./run_morpho_generate czech-morfflex-pdt-131112.tagger-fast
1 --from_tagger
```

output (whitespaces edited):

```
Loading dictionary from tagger:  done
mé          můj_^(přivlast.)  PSNS1-S1------1
vznášedlo   vznášedlo         NNNS1-----A----
```

```
je          být                 VB-S---3P-AA---
plné        plný                AANS1y----1A----
úhořů       úhoř                NNMP2-----A----
```

We can see that the generator assigned a correct form to each lemma+tag pair except the last one. It is, of course, not guaranteed that the generator generates some form from an arbitrary lemma+tag pair (at least because not all tags are valid, as mentioned before), but this case might be a little bit surprising — we were able to convert the original form (".") to lemma+tag but the generator cannot make it the other way round using the same model; we have to handle this somehow in our compression algorithm.

1.3.5.2 Generator API

There is a class called `morpho` which represents a morphological dictionary, this class can be used to perform morphological generation.[23] We can use method

```
virtual const morpho* tagger::get_morpho () const
```

to get an instance of `morpho` associated with a specific `tagger` instance.[23] For morphological generation, we can use method

```
virtual int morpho::generate (
   string_piece lemmma,
   const char* tag_wildcard,
   guesser_mode guesser,
   std::vector<tagged_lemma_forms>& forms
) const
```

where `lemmma` and `tag_wildcard` are input parameters with intuitive meaning, `guesser` is an enumeration with two possible values (whether to allow using the guesser or not) and `forms` is a vector holding the result of generation (each element of the vector contains a lemma + forms generated from the lemma with the respective tags).[23] Again, it's obvious that we can use here all functionality that has been shown with the executable example.

1.4 Introduction into data compression

Data compression (further just "compression") is the process of converting (encoding) data into some less space-consuming form.[27] Generally, data compression is achieved by removing redundancy from the data.[27] Below we define a few important terms that are used in the following sections and chapters.

Definition 4. Compression is the process of transforming original data into compressed data (encoding). **Compressor** (encoder) is thus a program

performing compression.[27]

Definition 5. Decompression is the process of transforming compressed data into original data (decoding). **Decompressor** (decoder) is thus a program performing decompression.[27]

Definition 6. Lossless compression is a compression method which involves no loss of information (when the compressed data is decompressed, the result is identical to the original data).[27]

Definition 7. A **nonadaptive** or **static** compression method is a compression method that does not modify its operations and parameters in response to the data being compressed.[27]

Definition 8. A **semi-adaptive** compression method is a 2-pass compression method that in the first pass examines the original data and sets parameters for the compression, which is then performed in the second pass.[27]

Definition 9. An **adaptive** compression method is a compression method that modifies its operations and parameters according to the original data during the compression process.[27]

The difference between adaptive and semi-adaptive methods is that adaptive methods are supposed to pass through the original data just once.

Definition 10. Compression ratio is a measure of compression performance. It can be computed by the following formula:[27]

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

It is obvious that lower compression ratio means better compression performance. In this thesis, we always express the compression ratio in percents.

Definition 11. This definition summarizes basic information about entropy (the theory has been developed by Claude Shannon[28]).

Consider a set of source units $S = \{x_1, x_2, \ldots, x_n\}$ with probabilities $P = \{p_1, p_2, \ldots, p_n\}$. Then average **entropy** (information content) of a source unit from S is equal to

$$H(S) = -\sum_{i=1}^{n} p_i \log_2 p_i \text{ bits} \quad [29]$$

The set of source units is referred to as an **alphabet** and source units are called **symbols** in this thesis.

Entropy of a source unit x_i is equal to

$$H_i = -\log_2 p_i \text{ bits} \quad [29]$$

Entropy of a message $X = x_{i_1} x_{i_2} \dots x_{i_k}, X \in S^+$ is equal to

$$H(X) = -\sum_{j=1}^{k} \log_2 p_{i_j} \text{ bits} \quad [29]$$

Entropy gives us a theoretical limit for data compression — given entropy H of the source units and the length n of the input message, we cannot compress the message further than to nH bits on average.[27] A good compressor should compress the input data close to its entropy. The term **entropy encoders** is used for encoders which are almost optimal.[27]

Definition 12. Statistical compression method is a compression method which assigns variable-size codes to the symbols depending on the probability of their occurrence (symbols which appear more often in the data have shorter codes than those which appear less often).[27]

We note here that arithmetic coding (mentioned further) is also considered a statistical compression method; it assigns a code to the whole input (based on the probabilities of individual symbols), however.

Definition 13. Context-based compression method is a compression method which assigns probability to a symbol according to context of that symbol (by context, we mean N preceding symbols).[27]

1.5 Arithmetic coding

All information about arithmetic coding contained in this section has been taken from [27] (direct citations are explicitly marked).

Arithmetic coding is a statistical method of compression which assigns a code to the entire input data. This is an advantage over statistical compression methods which assign codes of integer length to individual symbols; if we consider Huffman compression method, which produces best codes for individual data symbols, the average size of codes produced by Huffman coding equals the entropy of input data only if the probabilities of symbols are equal to negative powers of 2. Arithmetic coding overcomes this problem; it is an entropy encoder.

The basic idea of arithmetic encoding is narrowing an initial interval $[0, 1)$ with every consecutive input symbol; symbols with higher probability narrow the interval less than symbols with lower probability (this corresponds to the fact that the number of bits needed to specify the interval grows as the interval gets narrower). The output of arithmetic coding is a number from the range $[0, 1)$, thus we only need to store the decimal part of the resulting number (the integer part is always 0). In fact, the output can be any number from the final interval.

This was just a brief intro; further we show the principle in detail and together with an example taken directly from [27].

1.5.1 Encoding process

We summarize the whole encoding process by a direct citation of Salomon's book [27]:

1. Start by defining the "current interval" as $[0, 1)$.
2. Repeat the following two steps for each symbol s in the input stream:
   a) Divide the current interval into subintervals whose sizes are proportional to the symbols' probabilities.
   b) Select the subinterval for s and define it as the new current interval.
3. When the entire input stream has been processed in this way, the output should be any number that uniquely identifies the current interval (i.e., any number inside the current interval).

Salomon also states that the probabilities used in step 2a don't need to be rigid during the whole encoding process — they may change all the time. This also means that we can even encode symbols from different alphabet in every iteration of the encoding cycle (if we have a decoder that works the same way).

In our example, we are using semi-adaptive version of arithmetic coding (which first inspects the input data to determine probabilities of the symbols and then encodes using these probabilities); there is also an adaptive variant of arithmetic coding which updates the probabilities with every consecutive symbol read from the input, and for an alphabet of fixed size, we could also use some fixed probabilities (with no respect to the input data). For simplicity, we omit here the necessary step of storing the symbol ranges into the output (so that the decoder can read them as a necessary initial info).

The input data for encoding is string SWISS_MISS. Table 1.3 contains information which is used in the encoding process — for every input symbol, it contains its frequency in the input data, its probability (based on the number of its occurrences in the input data) and a corresponding subrange. The subrange is always relative to the current interval (see step 2a in the algorithm described above). In fact, the second and third column are there just for clarification of how subranges have been computed.

In the encoding algorithm, we can describe the current interval borders by variables `Low` and `High`. Then, in algorithm step 2b, we can calculate the new interval borders `NewLow` and `NewHigh` using the following formulas, where `LowRange(X)` and `HighRange(X)` are borders of the subrange for symbol X being encoded in the current iteration:

```
NewLow:=Low+(High-Low)*LowRange(X)
NewHigh:=Low+(High-Low)*HighRange(X)
```

Table 1.3: Table for arithmetic coding example 1

input symbol	frequency	probability	subrange
S	5	0.5	$[0.5, 1.0)$
W	1	0.1	$[0.4, 0.5)$
I	2	0.2	$[0.2, 0.4)$
M	1	0.1	$[0.1, 0.2)$
_	1	0.1	$[0.0, 0.1)$

To make the process clearer, we show how the variables `Low` and `High` are changing during encoding of our example input for the first three input symbols:

encoding symbol S:

$$Low = 0.0 + (1.0 - 0.0) * 0.5 = 0.5$$
$$High = 0.0 + (1.0 - 0.0) * 1.0 = 1.0$$

encoding symbol W:

$$Low = 0.5 + (1.0 - 0.5) * 0.4 = 0.70$$
$$High = 0.5 + (1.0 - 0.5) * 0.5 = 0.75$$

encoding symbol I:

$$Low = 0.7 + (0.75 - 0.70) * 0.2 = 0.71$$
$$High = 0.7 + (0.75 - 0.70) * 0.4 = 0.72$$

If we continued until the whole input string has been encoded, then after encoding the last symbol, `Low` would be equal to 0.71753375 and `High` would be equal to 0.717535. Then we could output any number from the interval $[71753375, 717535)$ as result of the compression (a good practice is to choose the number with shortest bit representation).

1.5.2 End-of-message problem

It's necessary to mention that we must also add some mark for the decoder to stop the decoding process (we don't consider this fact in our examples for simplicity). One possibility is to output the size of the input first (which can then be read and remembered by the decoder), another possibility is to add a special *EOF* symbol to our alphabet (this symbol should have a very small probability and should be encoded after the last symbol of the input message; when the decoder decodes this special symbol, it knows that the decoding should stop).

1.5.3 Decoding process

In our semi-adaptive example, the decoder first reads info on input symbols and their ranges (stored by the encoder). Then it reads the code representing the encoded data (described as `Code` later in this subsection). Afterwards, until the end of the message has been reached (this problem was discussed in the previous subsection), it repeats the following steps:

1. Using the info on symbols and their ranges, find the range to which the number `Code` belongs; this specifies the decoded symbol X (for example, if `Code` equals 0.47, then we have decoded symbol W since $0.47 \in [0.4, 0.5)$).

2. Compute the new value of `Code`:

   ```
   Code:=(Code-LowRange(X))/(HighRange(X)-LowRange(X))
   ```

Let's say that the code stored by encoder in our encoding example was 71753375, thus the initial value of `Code` equals 0.71753375. Here we show the decoding of first three symbols of our encoded message.

The first decoded symbol is S since $0.71753375 \in [0.5, 1.0)$. Let's recompute the value of `Code`:

$$Code = (0.71753375 - 0.5)/0.5 = 0.4350675$$

The second decoded symbol is W since $0.4350675 \in [0.4, 0.5)$. Recompute `Code` again:

$$Code = (0.4350675 - 0.4)/0.1 = 0.350675$$

The third decoded symbol is I since $0.350675 \in [0.2, 0.4)$. Recompute `Code` again:

$$Code = (0.350675 - 0.2)/0.2 = 0.753375$$

This way the decoder continues until whole message has been decoded. We have described here the theoretical variant of arithmetic coding, real implementations are a little bit different and more complicated. The differences are described in the following subsection.

1.5.4 Implementation of arithmetic coding

A big disadvantage of the previously described theoretical concept of arithmetic coding is that it requires unlimited precision of numbers `Low` and `High`. Another disadvantage is that the number `Code` can be very long and dividing such number would be slow. These problems with precision and complexity can be solved by using integers of fixed length for computation; this requires a slight modification of the previously described algorithm.

Table 1.4: Table for arithmetic coding example 2

input symbol	frequency	CumFreq
S	5	5
W	1	4
I	2	2
M	1	1
_	1	0

In the following text, we are explaining this modification using imaginary integers with capacity of 4 decimal digits to keep values of `Low`, `High` and `Code`. The largest digit here is thus 9; in practice, we would use for example 32-bit integers and we would work with binary digits — in that case, the largest digit would be 1 and otherwise the approach would be the same as for decimal digits.

The values of `Low` and `High` are initialized to 0000 and 9999, respectively (for both encoding and decoding). When, in some moment, the leftmost digit of `Low` equals the leftmost digit of `High`, then the leftmost digit is output and both `Low` and `High` are shifted by one to the left (the new rightmost digit of `Low` is then set to 0 and the new rightmost digit of `High` is set to 9).

The described handling of `Low` and `High` fulfils both of the following requirements (we don't show the proof here, it can be found in [27]):

1. The initial value of `Low` and `High` is equal to 0 and 1, respectively.

2. The value of both `Low` and `High` can be interpreted as a fraction less than 1.

Now we show how the encoding and decoding process changes when using the fixed-length integers, as well as an example directly taken from [27]. The coder needs information contained in Table 1.4 which is quite similar to Table 1.3, though there is a new column containing values of `CumFreq(X)` (cumulative frequency) for each symbol X. We can then define `TotalFreq` as the sum of `CumFreq(X)` for all X (which equals 10 in our example), `LowCumFreq(X)` as `CumFreq(X)` and `HighCumFreq(X)` as `CumFreq` of the symbol above X in the table (or as `TotalFreq` if no such symbol exists).

The formulas for recomputing `Low` and `High` are slightly different here when compared with the theoretical variant:

```
NewLow:=Low+(High-Low+1)*LowCumFreq(X)/TotalFreq
NewHigh:=Low+(High-Low+1)*HighCumFreq(X)/TotalFreq - 1
```

1.5.4.1 Implementation of arithmetic encoding

We again want to encode string SWISS_MISS in our example. Here we show how the encoding proceeds for the first three input symbols:

encoding symbol S:

$$Low = 0 + (9999 - 0 + 1) * 5/10 = 5000$$
$$High = 0 + (9999 - 0 + 1) * 10/10 - 1 = 9999$$
Output: none

encoding symbol W:

$$Low = 5000 + (9999 - 5000 + 1) * 4/10 = 7000$$
$$High = 5000 + (9999 - 5000 + 1) * 5/10 - 1 = 7499$$
Output: 7 ($Low = 0000$, $High = 4999$)

encoding symbol I:

$$Low = 0 + (4999 - 0 + 1) * 2/10 = 1000$$
$$High = 0 + (4999 - 0 + 1) * 4/10 - 1 = 1999$$
Output: 1 ($Low = 0000$, $High = 9999$)

Now let's imagine that we have processed all symbols from the input string, not just the first three symbols. After encoding the last symbol, the values of `Low` and `High` equal 3750 and 4999, respectively. Then 3750 is output and the encoding is finished. We again omitted the fact that in practice, we would probably encode some special *EOF* symbol as last symbol of the message.

1.5.4.2 Implementation of arithmetic decoding

The values of `Low` and `High` are initialized and recomputed the same way as during encoding. The initial value of `Code` is set to first 4 digits from the compressed data (which is, in our shortened example, 7175).

When the values of `Low` and `High` are shifted (which would result in output of the first digit during encoding), `Code` is also shifted (to the left by one) and the rightmost digit of `Code` is set to the next digit from the compressed data.

The decoding process can be summarized as follows (partially a direct citation from [27]; we don't show an example calculation here):

1. Calculate `index:=((Code-Low+1)*TotalFreq-1)/(High-Low+1)` and truncate it to the nearest integer.
2. Use the value of `index` to find the next symbol by comparing it to the cumulative frequencies from Table 1.4.
3. Update `Low` and `High` (already described).
4. Update `Code` if `Low` and `High` have been shifted (already described).

1.5.4.3 Underflow problem

The shifting of `Low` and `High` prevents the values of both variables from getting too close to each other when the first digits of `Low` and `High` are equal. However, it could happen that they get close to each other anyway (for example when both values converge to 500000, then `Low` can reach the value of 499999 and `High` can reach the value of 500000; then the algorithm will not output anything for the rest of iterations, even if it should). We need to detect such cases early and rescale the variables to avoid this situation.

Salomon's book [27] shows a solution for case when `Low` has reached the value of $49xxxx$ and `High` has reached the value of $50yyyy$. Then we should set `Low` to $4xxxx0$ an `High` to $5yyyy9$. If we do this n times before the most significant digits of `Low` and `High` become equal, then if the most significant digit equals 4, we output n zeros, else the most significant digit equals 5 and we output n nines.

1.6 PPM

All information about PPM contained in this section has been taken from [27] (direct citations are explicitly marked).

PPM is a state-of-art, context-based compression method where the coder maintains a statistical model and uses an arithmetic coder as a subprocedure for the actual coding. We show here the most important motivation for using this method for text compression (although the use of this method is, of course, not limited to compression of texts).

- In English text, when we encounter a 't' letter, there is about 30% probability that the next symbol is 'h'. After letter 'q', there is about 99% probability that the next symbol is 'u' etc. This means that we can use the context to reduce the number of bits required to encode 'h' in context 't' (similarly for 'u' in context 'q' etc.), since we already know that high probability symbols make the arithmetic coder output less bits than low probability symbols.

An N-order PPM considers last N symbols when estimating the probability that the next symbol is S. N shouldn't be too large for the following reasons:

- We must somehow encode the first N symbols of the input before we can encode any symbol in context of length N; encoding the first N symbols might be a problem reducing the overall compression.
- For a given fixed-sized alphabet, the number of possible contexts grows exponentially with N, which may make the statistical model unacceptably large (too much memory-consuming).

- As we go through the input data, its nature may be changing significantly. For a better compression, the context shouldn't retain too much information about old data.

An adaptive version of PPM builds a statistical model which is updated with every incoming symbol. PPM may also use a static model, but we deal only with the adaptive version in this thesis. The adaptive version is slower and more complex, but it is generally better than the static version because it adapts well to the nature of the input data.

Probably the fastest way to explain how PPM works is to show an actual example of PPM coding. This example has been taken directly from [27].

1.6.1 PPM example

We have a 2-order PPM model here. Table 1.5 shows what the model looks like at the moment when string "assanissimassa" has been encoded (notice that the current context is then "sa").

The model is being built and updated the same way during both encoding and decoding; we show an encoding example here. The column f (frequency) shows how many times we have encountered symbol S in the given context and p is the estimated probability that symbol S appears in the given context (the probability estimate is based on f). The probability is used when encoding the symbols using the arithmetic encoder.

Even though we have a 2-order PPM, we keep also statistics for contexts shorter than 2. One of the most important ideas of PPM, which stands for "prediction with partial matching", is to shorten the context if the model doesn't have any info about S in the given context.

There appears a special symbol $\perp$ in the table, which is called *escape symbol*. This symbol is encoded when a not-yet-known symbol (not-yet-known in the given context) is encountered; this is necessary so that the decoder knows when to switch to a shorter context.

If the incoming symbol S has not yet been encountered, the encoder doesn't find it in any order in the model (including order 0). In such case, the encoder switches to a special order -1 where each of the possible input symbols is assigned a fixed probability equal to 1/(size of the alphabet).

Column called "Order 0" in fact keeps statistics about how many times each of the non-escape symbols has appeared so far (regardless of any context), thus a 0-order PPM is in fact "just a type of adaptive arithmetic coder".

Let's take a look at info about context "ss" in Table 1.5. The info says that only symbols 'a' and 'i' have been encountered in this context so far; 'a' has been encountered twice and 'i' once. The escape symbol has a frequency of 2 since the coder had to switch to a shorter context from context "ss" twice so far (once when symbol 'a' was first encountered in this context and once when symbol 'i' was first encountered in this context). Now the meaning of

Table 1.5: PPM encoding example

Order 2				Order 1				Order 0			
Context	S	f	p	Context	S	f	p	Context	S	f	p
as	s	2	2/3	a	s	2	2/5	ε	a	4	4/19
as	$\perp$	1	1/3	a	n	1	1/5	ε	s	6	6/19
				a	$\perp$	2	2/5	ε	n	1	1/19
ss	a	2	2/5					ε	i	2	2/19
ss	i	1	1/5	s	s	3	3/9	ε	m	1	1/19
ss	$\perp$	2	2/5	s	a	2	2/9	ε	$\perp$	5	5/19
				s	i	1	1/9				
sa	n	1	1/2	s	$\perp$	3	3/9				
sa	$\perp$	1	1/2								
				n	i	1	1/2				
an	i	1	1/2	n	$\perp$	1	1/2				
an	$\perp$	1	1/2								
				i	s	1	1/4				
ni	s	1	1/2	i	m	1	1/4				
ni	$\perp$	1	1/2	i	$\perp$	2	2/4				
is	s	1	1/2	m	a	1	1/2				
is	$\perp$	1	1/2	m	$\perp$	1	1/2				
si	m	1	1/2								
si	$\perp$	1	1/2								
im	a	1	1/2								
im	$\perp$	1	1/2								
ma	s	1	1/2								
ma	$\perp$	1	1/2								

all info in the table should be clear. Here we show how the next symbol S will be encoded — there are 4 possible situations depending on the value of S (the current context is "sa", as already mentioned):

1. Imagine that S equals 'n'. The model does contain info about symbol 'n' in context "sa" — the probability of 'n' in context "sa" is 1/2. The info about each context in Table 1.5 can be used similarly as the info in Table 1.3, the arithmetic coder thus encodes range $[0.5, 1)$.

2. Imagine that S equals 's'. There is no info about symbol 's' in context "sa", so the escape symbol is encoded in context "sa" (the arithmetic coder thus encodes range $[0, 0.5)$). The context is then shortened to "a"

and the encoder is in order 1; in context "a", the probability of 's' is 2/5 (the arithmetic encoder thus encodes $[3/5, 1)$).

3. Imagine that S equals 'm'. The encoder will have to switch to order 1 and then to order 0 where 'm' is finally found, thus the arithmetic coder will encode ranges $[0, 0.5)$, $[0, 2/5)$ and $[5/19, 6/19)$.

4. Imagine that S equals 'd'. The encoder will have to switch to order 1, then to order 0 and then to order -1 since 'd' never appeared before. The arithmetic coder will encode ranges $[0, 0.5)$, $[0, 2/5)$, $[0, 5/19)$ and then a range in context -1 (now not citing Salomon's book: if we have an alphabet of size 28 including escape symbol, then this range could be [4/28, 5/28) if we treat 'd' as the fifth letter of our alphabet; encoding an escape symbol in -1 order is used to mark the end of coding, by the way).

Here is a short information on how the model is updated with an incoming symbol— the frequencies and probabilities are updated for every order in the model; if S equals 's', then we increment frequency of 's' in contexts "sa", "a" and ε and the probabilities of symbols in these three contexts are updated accordingly. We take a deeper look on this when presenting the data structure for the model in Section 1.6.4.

1.6.2 Exclusion

Exclusion is a technique which improves the compression ratio achieved by PPM.

Consider the situation when the encoder was about to encode 's' in context "sa" in the previous subsection. According to Table 1.5, there was no 's' encountered in context "sa" so far, so the encoder encodes an escape symbol and switches to order 1. Now imagine what the decoder would do here when decoding — it would decode the escape symbol and switch to order 1 as well. However, the decoder would now know that the encoded symbol is not 'n' since otherwise it wouldn't have been necessary to switch to shorter context ('n' is already present in the table for context "sa"). This can be utilized here when encoding in order 1 — we know that symbol 'n' now has zero probability in context "a", so we temporarily *exclude* it from statistics for context "a" and thus 's' can be encoded in order 1 using range $[2/4, 1)$. If we didn't use exclusion, this range would have been $[3/5, 1)$ which is a narrower range than $[2/4, 1)$; we see that, thanks to exclusion, the arithmetic coder will output less bits.

The exclusion principle can, of course, be used in any order, including -1 order. When encoding an incoming symbol, the encoder just remembers all excludable symbols on its way to shorter contexts and excludes them appropri-

ately as described. For the next incoming symbol, the encoder does the same (but it builds the list of excludable symbols again from scratch, of course).

1.6.3 PPM variants

There are several variants of PPM which differ in the way of assigning probability to the escape symbol. Let us recall that the estimated probabilities of symbols in Table 1.5 are based on their observed frequency. In our example, we used PPMC, which, for a context X, sets the frequency of the escape symbol to a value equal to number of distinct symbols encountered in this context so far; this approach works good in practice since it keeps the frequency of the escape symbol relatively high when the encoder is encountering a lot of symbols not yet seen in the given context (in such case, it's quite likely that another not-yet-seen symbol will appear) and relatively low when not many not-yet-seen symbols are being encountered.

There are another common variants of PPM: PPMA always assigns the escape symbol a frequency of 1. PPMB is similar to PPMC, but when a symbol is encountered for the first time in the given context, PPMB keeps the symbol frequency on zero until it is encountered for the second time (since the second occurrence, the frequency of the symbol is incremented as in PPMC). Variants PPMP and PPMX treat the appearance of each symbol as a separate Poisson process and they estimate the probability of escape symbol using this assumption.

In practice, PPMC is slightly better than PPMA and PPMB but slightly worse than PPMP and PPMX.

1.6.4 PPM data structure

The most important requirement on data structure holding the PPM model is that updating the model and searching for the appropriate context should be fast. We present here a tree-like data structure called *trie*. For an N-order PPM, it reaches maximum depth of $N + 1$.

On Figure 1.1 we show how the trie changes during encoding of the first 4 symbols of string "assanissimassa" with an 2-order PPM. There is always a *root node* which represents -1 order. The nodes in depth 1 then represent order 0, the nodes in depth 2 represent order 1 etc. Each node except the root stands for one symbol in the given context and holds info on how many times we have seen this symbol in the given context; the context is specified by the shortest path from root to a specific node (when we look at the second trie diagram, the node $a; 1$ says that symbol 'a' has appeared once so far, the node below says that in context "a", symbol 's' has appeared once so far etc.). For each context, just one node is added effectively.

The grey nodes are the nodes added/updated with the current incoming symbol. The dotted lines are pointers which we call *suffix links*; they point to

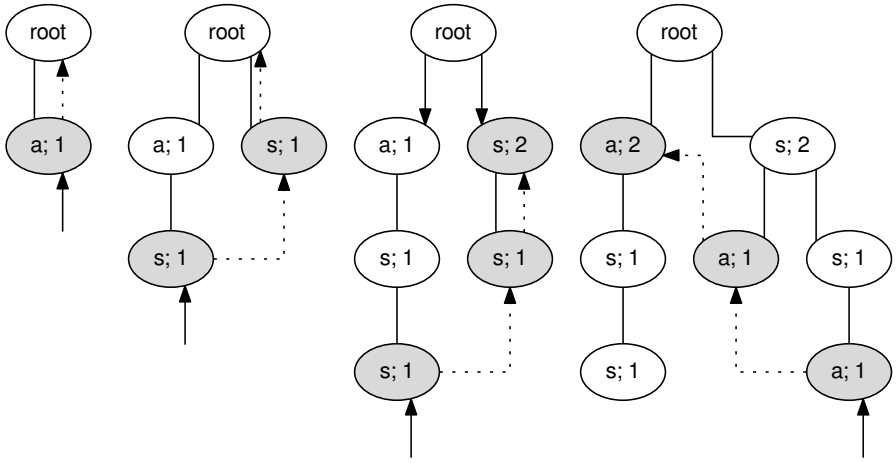


Figure 1.1: Example of PPM trie - encoding first 4 symbols of string "assanissimassa" with an 2-order PPM (created using Graphviz[30])

the node representing shorter context (thus, during encoding and decoding, the context can be shortened quickly and easily); for simplicity, each of the four diagrams shows only newly added suffix links, but the old ones still remain in the model. The last piece of diagram to explain is the isolated vertical arrow pointing to one of the nodes in each of the diagrams; this is a pointer called *base pointer* and it points to the deepest one of the nodes added/updated with the last incoming symbol (it tells us where to start when processing the next incoming symbol).

The first of the trie diagrams shows what the model looks like when the first symbol ('a') has been processed; we added one node saying that 'a' has been encountered once (in empty context) and we set a suffix link from context "a" to the shorter context, which, in this case, is empty context represented by root.

Then we encountered symbol 's'. We added a new node for symbol 's' in context "a" and a new node for symbol 's' in empty context; the suffix links have been set similarly as in the first case. Notice that the place where to start updating the model was specified with the base pointer (so we found it in constant time) and that after adding the first node, we used the existing suffix link from node "a;1" to find where to add the second node (again in constant time).

In the third diagram, we show how the trie is updated with the third

symbol, which is 's'. We added a new node for symbol 's' in context "as" and then a new node for symbol 's' in context "s"; the existing node representing symbol 's' in empty context has been just updated (the frequency of 's' in this context has been incremented). Again, we used base pointer and existing suffix links to update the model quickly.

In the fourth diagram, we updated the trie with the fourth symbol, which is 'a' — two nodes have been added and one node has been updated since it already existed. Notice that we didn't increase the depth of the trie above $N + 1$ — we used the base pointer to find where to start the update but we didn't add any child node there.

That's how updating of the trie works. In Table 1.5, we also had info on the frequencies of escape symbols; the frequency of escape symbol in the given context is (in case of PPMC) equal to the number of child nodes of the node representing the given context, so the number of child nodes is kept for each of the nodes in the trie. The probabilities of the symbols are then calculated the same way as in Table 1.5.

1.7 Text compression

This section contains a very brief intro into text compression which may be useful to read before we get further. Basically, the following approaches are used:

- encoding text as a sequence of characters,
- encoding text as a sequence of words,
- other, more "exotic" approaches, like syllable-oriented compression (this approach was used on Czech text in [31] with ambiguous results).

The first two approaches are the most common ones. Since we want to utilize word-related info (lemmas and tags) during the compression, it makes no sense to use some other method than word-based compression. The following subsection contains a very basic comparison of word-based and character-based compression and it also gives a very brief summary of some of the key properties of word-based compression.

1.7.1 Word-based compression properties

As already mentioned, we focus on compression ratio rather than on speed of the compression and decompression in this thesis. According to [32], "words reflect the true entropy of the text much better than characters", thus it makes sense to use word-based compression instead of character-based compression when compressing natural text. Another hint that the word-based approach is better is thesis [33] where word-based methods generally achieved better

compression ratio than comparable character-based methods on large text files.

Source [34] praises word-based compression as well and uses Heaps' law to justify two important statements about the word-based approach. We first present this law using source [35]:

Theorem 1. Heaps' law:

$$M = kT^b$$

where M is vocabulary size (number of distinct terms in the collection), T is number of tokens in the collection and k and b parameters with unspecified values (typically $30 \leq k \leq 100$ and $b \approx 0.5$).

The two important statements from [34] are:

1. We have to manage a large source alphabet (alphabet of words); however, the size of the alphabet grows slower and slower as the text collection gets larger.

2. The size of the source alphabet is relatively large for small documents, thus an adaptive word-based compression may not be useful for small documents (there is not enough data to build a good model).

With the information contained in this section, we can state that word-based compression is a good choice anyway. We should just keep in mind the two consequences of the Heaps' law mentioned above.

1.8 State of the art

This section contains info about what has been achieved in the field of lossless natural text compression using lemmatisation, tagging and morphological generation before this thesis has started. However, it seems that with Czech text, not much has been implemented (or published) so far and the same it is with other languages.

1.8.1 State of the art for Czech language

According to our best knowledge, there is only a paper by Ondřej Kazík and Jan Lánský [36] describing how part-of-speech tags can be used in compression of natural Czech text. According to this publication, the authors have shown that "separation of coding of part-of-speech tags of a sentence (so called sentence types) from the text and coding this sentence types separately can improve resulting compression ratio".

The main point in their compression algorithm is that they use sequences of consecutive part-of-speech tags to identify a "type of sentence" (though

they work here rather with pieces of sentences than with entire sentences); the encoded type of sentence then specifies the part-of-speech tags of the next N words. For each part-of-speech, they have a separate model of words. Last but not least, they use an initial model in their adaptive algorithm, so that the coder can use some pre-gathered info about natural Czech text before reading any single byte from the text to be compressed. It's necessary to say that their compression algorithm works well especially with small files; when compressing files larger than 100 kB, it is surpassed by the common `bzip2` compression algorithm.

1.8.2 State of the art for other languages

We didn't find any specific work here. There is a paper from 1992 by R. Nigel Horspool and Gordon V. Cormack [37] stating that "the rules of grammar strongly influence the probabilities of certain words appearing in certain contexts". The document proposes following method of using part-of-speech tags: If there is a word the part of speech of which equals X, then encode the next word using probability estimates associated with X. Actually, this is the idea of one of our experiments performed in scope of this thesis.

Chapter 2

Analysis

This chapter summarizes our ideas and our requirements on the final program before the design phase. All information about MorphoDiTa in this chapter is taken from [23] if not stated otherwise. See Section 1.3 for a brief intro into this tool.

We should mention here once more what our priorities are: We are trying to create a compressor with a good compression ratio and we don't really care about speed or memory consumption as long as speed and memory consumption are acceptable. That's why we occasionally ignore possible speed and memory consumption improvements if this ignorance doesn't negatively affect the compression ratio and if it simplifies the implementation.

We want our program to be compilable and runnable on GNU/Linux; a version for Windows may be added in the future but that is not our priority in this thesis, since our task is just proof of the concept (MorphoDiTa works under both GNU/Linux and Windows[5]).

2.1 Incorporating MorphoDiTa into our project

We already know that MorphoDiTa is available in three basic forms: as a standalone tool, as a library and as a web service.[6] Using MorphoDiTa as a library seems to be the most convenient option for our compression experiments. When we unpack the archive with MorphoDiTa 1.3.0 (downloaded from [21], as already mentioned), we find a makefile which can be used to easily create either static or dynamic library; using a static library should be easier and we don't have any special reason to use the dynamic version in our program.

The MorphoDiTa API is defined in header file `morphodita.h`, thus we should include this file into our code project to get the necessary declarations.

It seems useful to wrap all used MorphoDiTa functionality somehow so that we have an interface separating MorphoDiTa from the rest of our program (if the MorphoDiTa API changes a little bit in some of the future versions, it

shouldn't be that hard to incorporate the new version into our project this way).

2.2 Input of our compression program

In this section, we discuss how the text input of our compression program should be processed.

2.2.1 Input encoding

A Czech text is expected to contain diacritics. Moreover, a Czech text may contain characters from any existing alphabet (it may e.g. include some Chinese characters, when the topic is China-related). Thus it makes sense to use some Unicode encoding. Since all strings sent to MorphoDiTa should be UTF-8-encoded, we decided that our compression program will only work with UTF-8-encoded texts.

2.2.2 Input processing

2.2.2.1 Tokenizing the text input

As already mentioned, our compression algorithms will be word-based, so we would like to tokenize the text into words. We use term *white token* for tokens separating the words and term *black token* for the words themselves, thus we can describe the input text as a stream of black and white tokens.

We can use MorphoDiTa class `tokenizer` to split the text into tokens (an instance of `tokenizer` can be acquired from an instance of `tagger` or `morpho`). Class `tokenizer` offers the following important methods:

- `void set_text (string_piece text, bool make_copy = false)` which accepts text to be processed by the tokenizer,
- ```
  bool next_sentence (
     std::vector<string_piece>* forms,
     std::vector<token_range>* tokens
  )
  ```
  which tokenizes the next sentence if there's some text left to process (`forms` holds info about token ranges in bytes, `tokens` holds info about token ranges in Unicode characters — we can set either of the parameters to `NULL` to prevent getting info we don't need); we just note that it is not clear what exactly is considered a sentence here.

The tokenizer only gives us borders of black tokens; since our compression must be lossless, we need to encode the white tokens too, thus we have to calculate the borders of white tokens on our own (knowing the borders of black tokens, it is algorithmically easy though).
  ```

2.2.2.2 Input reading

We need to handle the file reading before passing the text to MorphoDiTa tokenizer since no method from MorphoDiTa API processes the input file directly. Our input reader should just be able to deliver the next piece of input text to tokenizer when needed.

A slight problem here is that when the tagger does tagging and lemmatisation on a piece of sentence, it may perform differently (and maybe worse) than when tagging the whole sentence (see experiments with "isolated tagging" later in this thesis). As already mentioned, only the `tokenizer` class knows what should be considered a sentence. We could use some heuristics here to estimate what a sentence is or we could just deliver a fixed number of lines to the tokenizer everytime and accept risk of a slight degradation of tagging. Another possibility is to deliver the whole input text at once; this approach sounds kind of dummy but we surely won't accidentally split sentences this way and the waste of memory is still acceptable here (we don't plan to work with extremely large files in this thesis). Last but not least, such a dummy input reader is very easy to implement.

2.2.2.3 Input stream for compression

We already know that once the text has been tokenized, it is very easy to do the tagging and lemmatisation (using just one method of MorphoDiTa class `tagger`). However, we don't want the compressor itself to deal with tagging all the time in our compression experiments; the compressor should receive already prepared tags and lemmas and just *use* them.

Similarly, we already know how to get stream of white and black tokens from the input text but the compressor should be separated from the process of tokenizing. If we look at a text, it is in fact a stream of regularly alternating black and white tokens, with some white tokens possibly being empty (having zero length): In text "Mé vznášedlo je plné úhořů.", there are six black tokens (according to MorphoDiTa tokenizer) and five white tokens between them (according to our definition), including an empty white token between "úhořů" and ".". We could somehow wrap the tokenizing process so that the compressor knows that after every black token, there is a white token and the other way around. Moreover, we can make every input text start with a black token to simplify things further (either the text really starts with a black token or we add there an "artificial" empty black token, which probably won't happen very often).

Summarizing the two paragraphs, the custom input stream for compression could look as follows:

- The stream starts with a black token and then the white and black tokens alternate regularly.

- For every black token, the stream contains its lemma and tag.

We may not need lemmas and/or tags in every compression experiment, so there could be an option to turn off automatic tagging and lemmatisation in our input module, but we won't implement such option unless the unneeded tagging and lemmatisation is a big performance problem.

2.3 Test files

It would have been optimal to test the compression performance on a compression corpus, which would ensure that the test set is balanced and that we can easily compare our algorithms against other compression methods. However, as far as we know, there is no such corpus of files containing natural Czech text, thus we had to create our own test set.

Our final test set consists of the following files:

- genesis.txt (124.2 kB) — first of the five books of *Thorah*, translated by Isidor Hirsch; rather an archaic text with verse-like structure and frequent repetitions of words (available in [38]),
- komunikace.txt (49 kB) — *O digitální komunikaci*, a technical essay about digital communication by Tomáš Svoboda; this text is relatively informal, using modern language (available in [39]),
- mloci.txt (456.8 kB) — *Válka s mloky* (*War with the Newts*), a novel by Karel Čapek; a piece of literary art containing an extensive vocabulary (available in [40]),
- zakonik.txt (1.3 MB) — civil code of the Czech Republic (*Občanský zákoník*) released in 2012; a structured text with frequent repetitions of words (available in [41]).

This test set seems to be diverse enough to give us basic estimate of how successful our compression algorithms are, since it contains texts of various style, structure and length (the set contains no very small file though since it makes no sense for adaptive compression methods; the selection of compression method is described further in this thesis). Information about licensing of the texts is available in a readme file enclosed in the test files folder.

2.4 Compression experiments

In this section, we present the main ideas of each of the experiments, discuss possible difficulties and suggest how to overcome the difficulties. Full and formal specification of each of the experiments is presented in Section 2.8.

We want to compress the text in a fully linear way, just as a stream of tokens without recognizing any deeper structures. The compression of a piece of the text will be influenced only by the preceding text. We will heavily focus on context-based compression approach since the probability of a word or its grammatical categories is often strongly influenced by the previous words and their grammatical categories. PPM, a state-of-art context-based compression method, seems to be a suitable compression method in this case when we are focusing on compression ratio. We want to use a fully adaptive version of PPM for the following reasons:

- A fully adaptive version of PPM adapts well on the input data.
- A static (fully nonadaptive) version of PPM would probably produce poor results for texts with unusual structure. Moreover, creating a static model containing all possible tokens for our word-based compression would be challenging.
- A fully adaptive version of PPM seems to be easier to implement than some semi-adaptive version of PPM which would first build the model according to input data and then compress using this model (we would have to pass through the input text twice and we would also have to store the model in the compressed file so that the decoder can use the same model for decompression); both versions should asymptotically produce similar results anyway.

We could also prepare some tiny static model (as universal for different texts as possible) and always start the adaptive compression with this initial model. This would very likely improve the compression ratio when compared with the fully adaptive method, but we won't do that in this thesis to keep things simple.

Our plan is to develop a basic word-based compression algorithm and then to try to improve this word-based compression using the linguistic tools. In fact, the word-based compression will be just a reference algorithm to evaluate if a specific way of using the linguistic tools is beneficial or not. Rather than developing a highly complex compression algorithm, we will focus on one-dimensional improvements of the basic word-based compression to show which approach is probably worth developing further.

2.4.1 Basic word-based compression

The proposed word-based compression algorithm surely could be improved in many ways (for example by cleaning the model from words which haven't been used for a long time to increase the probabilities of more frequent words) but we just want a reasonable reference algorithm, not a perfect one.

2.4.1.1 Handling the tokens

The main benefit of the word-based compression is that it takes entire words (tokens) as input symbols; this means that, practically, each unique word is assigned an identifying ID and the compression method then works with the IDs instead of the actual words. If we, during our adaptive compression, encounter a not-yet-known word, we must assign it a not-yet-used ID. Moreover, we have to encode the new word character by character so that the decoder knows what the not-yet-known word looks like (thus our word-based compression will use a character-based compression as a subprocedure).

Since the set of black tokens appearing in a text should logically have no intersection with the set of white tokens, we will use different models for black and white tokens. We already suggested an input module delivering stream of regularly alternating black and white tokens, which we can use here — the compressor doesn't have to encode whether the current token is black or white (on the other hand, it will sometimes have to encode tokens of zero length).

The last idea is about the model of white tokens. We don't expect having many different white tokens in an average text, thus the model of white tokens will usually be very tiny — the most frequent white tokens will be a space, a newline and a white token of zero length (empty token), other white tokens are expected to be much less frequent and rather unusual. Going deeper in this analysis, we can guess that the newline-token will probably appear frequently only after a few specific black tokens (like ".", "?" or "!"). We can also guess that after tokens like "," or ";", a space-token will appear far more frequently than any other white token. Also after token "(", an empty white token will follow almost always. Based on this observation, we suggest a simple heuristics of holding an individual model of white tokens for every character appearing at the end of a black token (thus we will have a model for white tokens appearing after character ',', another model for white tokens appearing after character 's' etc.). Because of small number of distinct white tokens in an average text, we suggest that all the models of white tokens share a single alphabet of white tokens, so that the few white tokens don't have to be encoded character by character again and again.

2.4.1.2 Handling the case of letters

Another question is how to encode case of letters; we are opening this question for the following reasons:

- Two words which differ only in case of some letter(s) are two different words for us. More different words generally means more bits spent to encode a word on average (because we have larger alphabet of words); also considering two words different just because of different letter case may break our context model pretty much (which may result in more bits required for encoding, again).

- We already saw that the MorphoDiTa tools may discard some information about letter case; since we are not allowed to alter the text in any way, we would like to have some clever way to get over this problem and store the information about letter case separately if we need it.

It probably makes no sense to store information about case of every letter in the text — this would mean storing a lot of mostly unuseful information since a very big majority of letters in a natural text is lower-case. We could apply a heuristics here — the first word after tokens ".", "?", "!" usually starts with an upper-case letter, so let's encode the letter case only for first letters of words appearing just after ".", "?", "!"; this will solve most of our lower case-upper case problems and at the same moment we won't spend much bits on storing information about the letter case.

After storing the case of the first letter as described, we can modify the word so that the first letter of the word is in expected case; this will reduce lower case-upper case duplicities of words in our alphabet of words. For example, we will change the word "Pes" ("dog") to "pes" since "pes" is the usual form of this word, but for word "Alice" (name), we will keep the first letter in upper case since if we encounter this word inside a sentence next time, it will almost surely start with the upper-case letter too.

The heuristics in fact switches between two states (*activated* and *deactivated*) depending on the current black token, as shown by Figure 2.1.

2.4.2 Basic experiment using morphological generator

We start with a short motivation: As already mentioned, Czech language is highly inflective, thus there may be many different forms of a single word. For example, let's consider word "malý" ("small") — according to MorphoDiTa, it may occur in any of the following forms:

malej malejch malejm malejma malou malym malá malé malého malém malému malí malý malých malýho malým malýma malými malýmu menším menší menších menšího menším menšíma menšími menšímu nejmenším nejmenší nejmenších nejmenšího nejmenším nejmenšíma nejmenšími nejmenšímu

If we also include forms with a negative prefix, we get twice as many forms. Our basic word-based compression has to treat all those forms as different words — this increases the total size of black token alphabet and it may also worsen our context model since the model doesn't know that "malý" and "malým" are in fact the same contexts, only with different grammatical case.

We have already shown how a morphological generator works: given a lemma and a tag (possibly containing wildcards), it generates all possible word forms for that lemma and tag. If the tag contains only '?' characters (such tag is called *empty tag* in the rest of this thesis), the generator generates

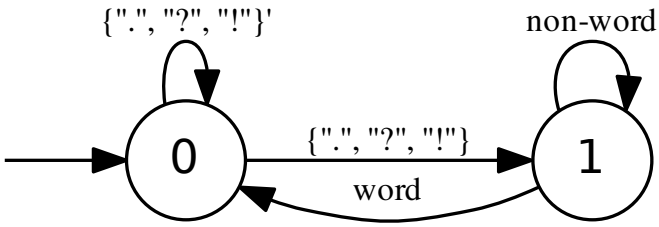


Figure 2.1: Diagram showing how the letter case heuristics switches between states (activated = 1, deactivated = 0) depending on the incoming black token. By "word" we mean a token starting with any letter of Czech alphabet. When the heuristics switches from activated to deactivated state, the size of the first letter of the token is encoded/decoded and the token may be modified as described in Section 2.4.1.2. Notice that the loop in state 1 allows us to skip quotation mark at the beginning of a sentence.

all possible forms of the given lemma; that's the basic idea of this compression experiment, the description of which follows.

We will alter our word-based compression (described in Section 2.4.1) so that instead of real black tokens appearing in the text, it stores the corresponding lemmas. As already indicated, it should bring us the following benefits (all resulting in less bits encoded):

- It will decrease the size of black token alphabet.
- By decreasing the total number of encoded tokens, it will also decrease the number of character-based encodings spent to encode a not-yet-known token.
- It will possibly also improve the context model of black tokens.

2.4.2.1 Index storing

To enable decoding of the original word form, we need to store not only the lemma; it is necessary to store an index into the list of forms generated by the morphological generator as well to specify the form (the decoder can then use the stored lemma and index to get the original form). One significant problem here is that the generation process isn't always symmetrical with the

lemmatisation process, as already shown earlier in this thesis. We must deal with the following special cases:

- For some lemmas, the generator doesn't generate any forms (like for lemma ".", as shown in Section 1.3.5).
- For some lemmas, the generator generates a non-empty list of forms not including the form we expect; we show two examples here which have been found during experiments with MorphoDiTa:
  - For form "PSI" ("DOGS"), we get lemma "pes" ("dog") but the generator doesn't generate form "PSI" from this lemma (problem with letter cases is partially solved by the letter case heuristics presented in Section 2.4.1.2, but not in this case).
  - For form "vyňal" ((he) "took out"), we get lemma "vyjmout" ("take out"), but then we don't find the form "vyňal" in the long list of forms generated from the lemma.

This means that the encoder must always examine the output of the morphological generator before encoding the lemma and index to make sure that the decoder is always able to decode the correct form. The encoder can use the following procedure when deciding what to encode:

- Case 1: The form is successfully generated from the lemma — then we simply store lemma + index (this case is expected to be the most typical one).
- Case 2: The form cannot be generated from the lemma — then we store the form itself as a "pseudolemma". We must keep in mind that the decoder won't know that we stored a pseudolemma instead of a lemma and it will trie to generate forms from it, thus the following is needed as well:
  - Case 2a: The generator generates no forms from the pseudolemma — then we don't have to encode any index (the decoder will know that if nothing can be generated, then the lemma equals the original form).
  - Case 2b: The generator generates some forms from the pseudolemma and the pseudolemma is among these forms — then we encode the corresponding index.
  - Case 2c: The generator generates some forms from the pseudolemma and the pseudolemma is not among these forms — now we have no other choice than to encode some special symbol to tell the decoder that the lemma equals the original form.

Note that, because of case 2c, we have to add one extra symbol into our alphabet of indexes for every lemma which is not contained among forms generated from itself.

Last thing to mention — for both lemmas and pseudolemmas, we don't have to encode the index if the generated list contains only one form and this form equals the lemma/pseudolemma used for generation.

The presented way how to overcome the problems with generation is just one of possible solutions. We could also solve these problems by storing a flag indicating whether the stored lemma is a real lemma or pseudolemma — then the decoder would know that it shouldn't apply morphological generation on the pseudolemma and we wouldn't have to add the extra index into the alphabet of indexes for many lemmas. However, we don't use this simpler solution since it may require to store up to one bit per every stored lemma, while the effect of the extra index in the alphabet of indexes should be not so significant (the increased size of the alphabet of indexes will affect the compression only when the PPM encodes in order -1 and usually the size of the alphabet is big anyway so one extra index doesn't change much).

2.4.2.2 Lemma ids vs. raw lemmas

We already know that the morphological generator ignores any lemma comments, thus it makes no sense to store full lemmas in this experiment. However, it may be interesting to inspect how the compression results will depend on whether we are storing just raw lemmas or lemma ids (let us recall that lemma ids uniquely identify the lemmas and may thus contain some technical info which makes them generally longer than raw lemmas).

Using of raw lemmas may worsen the context model (the different meanings of a word will now be no more distinguished) but it should bring the following benefits, all possibly resulting in a better compression ratio:

- Thanks to cutting off the technical info from the lemma and to decreasing the number of distinct lemmas, less character encodings will be required.

- The total number of distinct items in the model of black tokens will (slightly) decrease.

- The number of cases when we don't have to encode the index will increase (this is valid for cases when only one form can be generated from the raw lemma and this form equals the raw lemma while not being equal to lemma id, see Section 2.4.2.1).

2.4.3 Experiment with part-of-speech tags (1)

2.4.3.1 Motivation

It is intuitive that in a natural Czech text, just after an adjective, there often occurs a noun. Similarly, just after a preposition, there often appears a noun, pronoun or adjective. On our test files, we performed an experiment showing how the part-of-speech tag of one word in the text influences the part-of-speech tag of the next word (now we are talking about the first position in the tag produced by MorphoDiTa tagger, see [26] for a quick reference).

Table 2.1 shows the results of this experiment, which was run on files `komunikace.txt` and `mloci.txt`. For every value of the part-of-speech tag (except value X which is very rare), the table shows the most typical values of the following part-of-speech tag, together with their observed probability.

We can see that the results are very similar for both files. The table shows that for many values of POS tag, the distribution of values of the following POS tag is significantly biased in favour of one or a few values. We could utilize this in our next compression experiment, the description of which follows immediately.

2.4.3.2 Experiment description

We start with our basic word-based compression method described in Section 2.4.1. We extend this method with a model of part-of-speech tags — the value of POS tag will be stored for every black token. We expect that the values of POS tags will be highly compressible using a context model.

When the POS tag of every stored black token is specified, we don't need to hold all black tokens in a single model, we will thus have an individual model for every value of POS tag. These models will not share the alphabet of tokens since the number of words which can be correctly assigned multiple POS tags is insignificant. We expect that this will affect the compression in the following two ways:

- It will break our original context model of black tokens, which may worsen the compression (in individual models, we will have a context model for nouns only, then for adjectives only etc.).

- By dividing the black tokens into multiple models with zero or nearly-zero intersection, the average number of bits spent to encode a black token will decrease significantly, which could ultimately result in a better compression ratio than the ratio achieved by the basic word-based compression.

Notice that this way of using part-of-speech tags is very similar to the approach used in [36].

Table 2.1: POS (part-of-speech) dependency experiment — supposing that word X has POS tag "POS 1", the table shows which three POS tags "POS 2" are the most probable for word Y following X in the given text

POS 1	komunikace.txt		mloci.txt	
	POS 2	observed prob.	POS 2	observed prob.
A (adjective)	N	65.8 %	N	67.9 %
	Z	13.4 %	Z	12.8 %
	A	8.7 %	A	8.0 %
C (numeral)	N	52.4 %	N	54.5 %
	Z	17.7 %	Z	13.0 %
	A	11.2 %	A	8.8 %
D (adverb)	V	25.6 %	V	30.5 %
	A	17.7 %	Z	14.6 %
	Z	13.4 %	A	11.9 %
I (interjection)	n/a	n/a	Z	91.0 %
	n/a	n/a	N	3.6 %
	n/a	n/a	J	1.4 %
J (conjunction)	N	22.7 %	V	25.2 %
	V	17.5 %	N	18.2 %
	P	15.0 %	P	16.4 %
N (noun)	Z	38.3 %	Z	41.6 %
	V	14.5 %	N	14.3 %
	N	13.4 %	V	12.0 %
P (pronoun)	V	30.1 %	V	31.8 %
	N	27.7 %	N	22.8 %
	P	9.2 %	P	10.3 %
R (preposition)	N	44.6 %	N	49.8 %
	A	24.8 %	P	24.2 %
	P	21.3 %	A	20.5 %
T (particle)	R	24.5 %	Z	27.0 %
	P	20.4 %	P	14.7 %
	N, D	12.2 %	R	12.2 %
V (verb)	Z	17.1 %	Z	20.0 %
	N	16.2 %	P	16.1 %
	P	12.9 %	N	15.4 %
Z (punctuation)	J	21.8 %	Z	19.8 %
	N	16.2 %	N	15.9 %
	P	14.7 %	J	15.7 %

2.4.4 Experiment with part-of-speech tags (2)

Before detailed explanation of this experiment, we introduce the idea of *isolated tagging*. By *isolated tagging*, we mean using the tagger to tag individual black tokens separately (the input of the tagger is just a single black token here).

2.4.4.1 Isolated tagging vs. default tagging

Our default tagging approach is tagging of whole sentences at one time (we call it *sentence tagging* from now on). We suppose that the accuracy of tagging (and also the accuracy of lemmatisation) is lower when we use isolated tagging instead of sentence tagging since in case of isolated tagging, the tagger doesn't know the context of the word in the text. Since we are technically not able to inspect and evaluate the accuracy of tagging, we would like to know at least how often the results of both tagging approaches differ.

To see how big the difference is, we performed an experiment comparing lemma ids and tags acquired by sentence tagging with lemma ids and tags acquired by isolated tagging for each of the black tokens in files `komunikace.txt` and `mloci.txt`. This experiment is summarized in Table 2.2 (tag positions 12 and 13 are excluded since they have no meaning, see [26]).

Looking at the table, we can state that there really is some difference between results of sentence tagging and isolated tagging, as expected. However, for lemmas and also for many positions in the tag, the difference is not very significant. The most important fact here is that for part-of-speech tags (`tag[0]`), the difference is just about 2 %, thus isolated tagging is a suitable way how to estimate part-of-speech of an isolated black token.

The accuracy of isolated tagging in case of POS tags allows us to use POS tags during compression without storing them. However, not storing the POS tags means that the decoder will never know a POS tag of a token before it decodes the token.

2.4.4.2 Experiment description

We again start with our basic word-based compression method described in Section 2.4.1. We make the following changes:

- For every black token, we acquire its part-of-speech tag using isolated tagging (ignoring the tag delivered by the input module, which is acquired using sentence tagging).
- If the value of part-of-speech tag of a black token is X, then for encoding of the following black token, we will use a model for black tokens after POS X.

Table 2.2: All black tokens in the two test files have been lemmatised and tagged using sentence tagging and then using isolated tagging (if a specific token occurred n times in the text, it was included n times in this experiment). The table shows the percentage of black tokens where different results appeared.

	percentage of different results	
	komunikace.txt	mloci.txt
lemma	2.55 %	3.3 %
tag (whole)	23.4 %	19.3 %
tag[0]	1.95 %	2.45 %
tag[1]	2.02 %	2.61 %
tag[2]	8.09 %	7.67 %
tag[3]	6.62 %	5.53 %
tag[4]	19.6 %	15.6 %
tag[5]	0.013 %	0.03 %
tag[6]	0.013 %	0.009 %
tag[7]	0.16 %	0.29 %
tag[8]	0.21 %	0.31 %
tag[9]	0.36 %	0.66 %
tag[10]	0.76 %	1.02 %
tag[11]	0.21 %	0.31 %
tag[14]	0.53 %	0.76 %

The presented idea means that we will have a separate model of black tokens for every part of speech, but the individual models won't be part-of-speech-homogenous — the model for black tokens following adjectives, for example, will typically hold nouns, punctuations, adjectives and also other parts of speech, as estimated in Table 2.1. Since individual black tokens can appear in multiple models at the same time, all the models should share a single alphabet of black tokens.

We expect that this idea will affect the compression in following ways:

- It will break our original context model of black tokens, which may worsen the compression.
- The average number of bits spent to encode a black token may decrease anyway since the individual models of black tokens may reflect the text structure better than a single black token model used in Section 2.4.1 — some words may appear more often after nouns than after any other part of speech, for example.

Notice that this way of using part-of-speech tags is very similar to the approach proposed in [37].

2.4.5 Experiment with non-part-of-speech tags

2.4.5.1 Motivation

Let's see the list of forms which the MorphoDiTa generator generates from lemma "jarní" ("spring", adjective) and an empty tag:

jarnim jarní jarních jarního jarním jarníma jarními jarnímu jarnějším jarnější jarnějších jarnějšího jarnějším jarnějšíma jarnějšími jarnějšímu nejarnim nejarní nejarních nejarního nejarním nejarníma nejarními nejarnímu nejarnějšim nejarnější nejarnějších nejarnějšího nejarnějším nejarnějšíma nejarnějšími nejarnějšímu nejjarnějšim nejjarnější nejjarnějších nejjarnějšího nejjarnějším nejjarnějšíma nejjarnějšími nejjarnějšímu nejnejarnějšim nejnejarnější nejnejarnějších nejnejarnějšího nejnejarnějším nejnejarnějšíma nejnejarnějšími nejnejarnějšímu

We can see that the list contains a lot of rarely used forms. For example, 50% of the forms contain negative prefix "ne-" ("non-"); although these forms are fully valid in Czech, it is very unusual that someone uses the negative prefix together with adjective "jarní".

The problem is that in experiment described in Section 2.4.2, the compressor expects the forms with negation the same way as the forms without negation, thus in PPM order -1, the compressor assigns the negative forms a 50% probability in total, and the other 50% of probability are assigned to the non-negative forms in total. This is impractical since the negative forms here don't have a 50% probability in total in natural Czech texts.

The value of negation is specified by the 11th position in MorphoDiTa tags.[26] If we modify the experiment described in Section 2.4.2 so that together with a lemma of an adjective the value of negation is stored, then for generation of forms, we can use tag "??????????X????", where 'X' is the stored value of negation, instead of the empty tag "???????????????". Then, for lemma "jarní", the generator generates only 50% of the forms which would have been generated using an empty tag. Thus, according to the theory of entropy (see Definition 11), the compressor needs one bit less to store the index in PPM order -1 than if it used the empty tag. On the other hand, we suppose that for storing the value of negation, less than one bit per lemma will be needed since, according to our best estimates, the negative forms of adjectives are not as usual as the non-negative forms in a natural text. This may result in achieving better compression ratio than for the basic experiment described in Section 2.4.2.

Storing the value of negation may be beneficial also for verbs an adverbs since these parts of speech may contain the negative prefix "ne-" as well. Moreover, we can extend this experiment to any position in the tag and test which positions in the tag are worth storing and which are probably not.

2.4.5.2 Experiment description

As already mentioned, this experiment is based on experiment described in Section 2.4.2. The lemmas and indexes will be stored the same way as described there, we will just generate forms also from non-empty tags and we will have an alphabet of indexes for every pair lemma + tag value (the non-empty tags will be empty except exactly one position, as proposed). The input parameters of the experiment will be *selectedPOS* and *tagPosition*; the first one will be a list specifying for which parts of speech we want to store the tag (adjectives + verbs, for example), the latter will specify which position in the tag we want to store (e.g. position 10, which is the position where value of negation is stored if we use zero-based indexing).

To make the approach more clear, we show the whole idea using an example: Let's say that *selectedPOS* equals {A} (adjective) and *tagPosition* equals 10, which is the position where the value of negation is stored. Then, if we encounter an adjective during encoding (the part-of-speech should be estimated using isolated tagging so that we don't have to store it), we may store any of the values {-, A, N} depending on the value of negation extracted from the tag of the adjective (the tag is delivered by the input module). We may also store nothing if storing the value is not beneficial (this is explained further); in such case we are virtually (but not really) storing value '?' since this is the value used for generation if no tag value is stored. The morphological generation will thus always be performed using one of the following tags: "??????????-????", "??????????A????", "??????????N????", "????????????????". The model for encoding the index will thus be selected using lemma of the adjective + one of these four tags.

Now we need to discuss in which cases it is beneficial to store the tag value (for many tag positions, applicability of one value disqualifies the applicability of some other values and reserving non-zero probability for those values is thus useless). Before storing the value, we should try to generate forms from the lemma using each of the three tag values from set {-, A, N}. If a specific value decreases the number of generated forms (when compared with the number of forms generated using an empty tag) to some number greater than zero, then we say that storing this value is useful. Otherwise, the value is not useful (we don't need to store a value from which the original form can't be generated or a value which doesn't decrease the number of generated forms when compared with the empty tag). Regardless of which value we want to encode, we can do this check everytime and pre-exclude the not useful values.

When pre-excluding the values, it may happen (because of the already discussed asymmetry of lemmatisation and tagging vs. generation) that the value which we want to encode gets pre-excluded (we suppose that this will not happen often). In such case, we just store the first non-pre-excluded value from the set of possible values (the decoder does the pre-exclusion the same way as the encoder so we really cannot store any pre-excluded value here).

Another special case is when all values get pre-excluded — then we don't store the tag value and an empty tag is used for generation.

When we finally know which value from set {-, A, N, ?} should be stored, we are again facing the problem of index storing which is discussed in Section 2.4.2.1. Basically, the problem is solved the same way as described there, just the empty tag used for generation is updated with a value from set {-, A, N, ?}. If the original form can be generated using lemma + tag, it is case 1 (see Section 2.4.2.1), if not, it is the second case. In the second case, we use the original token instead of lemma and we must repeat the whole process of estimating POS, pre-excluding values and selecting which tag value should be stored.

2.5 Utilizing UTF-8 encoding

As already mentioned, our compression program will work only with UTF-8-encoded text. The MorphoDiTa class `tokenizer` shows us on which bytes in the text each black token starts and also the length of each black token in bytes, but that's all we can use here regarding the conversion between a sequence of bytes and a sequence of Unicode characters. If we want to do such conversion, we need to implement it on our own.

The conversions will be useful for the letter case heuristics presented in Section 2.4.1.2 so that we are able to recognize multibyte-characters and do the upper case-lower case conversion if needed. Another point of use is for models of white tokens — in Section 2.4.1.1 we proposed that there should be an individual model for each character appearing at the end of black tokens, thus we want to recognize the multibyte-characters at the end of black tokens as well.

However, we won't work with Unicode characters in the character by character encoding of not-yet-known tokens. Encoding the tokens byte by byte is very simple and it should work well in most cases for Czech texts. Encoding the tokens Unicode-character by Unicode-character would be more perfect but we would have to deal somehow with the large Unicode alphabet so that the size of the alphabet doesn't negatively affect the compression ratio. Since this is not the topic we should be focusing on in this thesis, we decided to use the simple byte by byte coding.

2.6 Compression method

As already mentioned, our experiments will use the PPM compression method. Summarizing the previous analysis, we have the following requirements on the implementation of PPM:

- The implementation must allow us to select a different trie in every coding step, the structure of the trie must thus be independent from the coder. In Section 2.4 describing the compression experiments, we are using the term *model* for what will be represented by an individual PPM trie in the implementation.

- We must be able to choose an individual size of the alphabet for every trie. The upper limit on the alphabet size should be at least several tens of thousands distinct symbols (among our test files, file `mloci.txt` has the most extensive vocabulary with about 17700 distinct black tokens; with respect to the Heaps' law, we don't expect that this number is much greater for any natural Czech text up to several megabytes long).

- There must be a possibility to use a dynamic-size alphabet since when compressing the natural text using an adaptive compression, we don't know the number of distinct tokens at the start of the compression (new symbols should be added on the fly during the compression).

- We would like to use the exclusion principle, extended by the possibility of pre-exclusion (manually selecting some symbols to be excluded as early as in the longest context of PPM, which is not a standard PPM functionality).

- We would like to be able to set the order of PPM to any small integer starting at 0. Zero-order PPM is in fact an arithmetic coder, thus by setting the order to 0, we can use the PPM as a statistical compression method instead of context-based compression method if needed.

We decided to take an existing implementation of PPM (including an arithmetic coder) and modify it to match our requirements instead of implementing the PPM from scratch. We chose implementation done by Jiří Krotil within his master's thesis [42], which was originally developed for ExCom [43], a library of compression algorithms maintained by the Prague Stringology Club. More exactly, we took what is called "basic implementation" in the thesis as our starting point. A brief summary of key features of the basic implementation together with a summary of required changes follows.

2.6.1 Krotil's basic implementation of PPM

The program is written in C programming language. The implementation includes four variants of PPM: PPMA, PPMB, PPMC and PPMD (PPMD is a version of PPM which increases the frequency of the escape symbol by $\frac{1}{2}$ for every new symbol in the given context, while PPMC increases the frequency by 1). The exclusion principle is not used in the basic implementation.

2.6.1.1 Trie structure

The trie is built from two types of nodes — *internal nodes* and *leaf nodes* which are represented by different C structures (internal nodes represent contexts and thus they don't appear in the deepest layer of the trie structure, while leaf nodes only appear in the deepest layer).

The main feature of the trie structure is that the list of child nodes of a node is represented by a linked list; this means that the time needed to search for a symbol in the given context is $O(n)$ where n is the length of the list.

If the sum of symbol frequencies in some context exceeds a specific limit, the stored frequencies are rescaled to lower values.

2.6.1.2 Memory management

To make the allocation (or deallocation) of nodes fast and easy, there is a special module called *memory manager* which allocates the nodes in groups; when a new node is needed, it is taken from the most recently allocated group if there is some not-yet-used node, otherwise a new group of nodes is allocated.

The memory manager also checks the size of the model; if the amount of allocated memory exceeds a specific limit, the model is deallocated and then reconstructed using last 2000 encoded (or decoded) symbols.

2.6.2 Required changes

We definitely need to modify the implementation to fulfil the requirements summarized at the beginning of Section 2.6. This includes:

- separating the trie structure from the coder,
- eliminating the fixed size of the alphabet (originally set to a fixed value of 256),
- implementing the exclusion and pre-exclusion,
- changing the trie structure so that the searching for symbols in a given context is faster (only needed if the implementation is unacceptably slow for large alphabets).

Other desirable changes include:

- changing the declaration of integer variables to use fixed-length integers where needed (this is especially important in the arithmetic coder so that the coding works exactly the same on every platform),
- converting the whole implementation to C++ (e.g. replacing C-like memory allocation by C++-like memory allocation),

- reorganizing the code and introducing an object-oriented architecture, so that the code is more readable and easily maintainable.

We plan to use only the PPMC version since it is the most common version of PPM with good compression results. Utilizing the model reallocation, which is done by memory manager to keep the memory consumption below a specific limit, is not our priority, thus this functionality will not be used (otherwise we would need to modify the memory manager so that it is able to work correctly with multiple tries at the same time).

Once the modification of Krotil's implementation is finished, the implementation of PPM can be incorporated into the ExCom library, thus it becomes an important by-product of this thesis.

2.7 Trie statistics

We are not able to analyze the performance of the compression experiments properly if we know only the achieved compression ratio. The proposed compression experiments always use multiple models and we already know that each of the models is represented by an individual trie in the PPM algorithm. We want that, at the end of the encoding process, a summary of compression performance for each trie is printed to the standard output. The summary should contain the following for each of the tries:

- name of the trie (we should assign a unique name to each trie for an easy identification),
- number of encodings carried out using this trie (in fact the number of symbols encoded using this trie; here we mean how many times the PPM encoder was asked to encode a symbol using this trie, thus this number shouldn't include escape symbols which are encoded by the PPM encoder when switching to shorter contexts),
- entropy of the encoded message in bits (see Definition 11; this number should be very close to the real size of the output since arithmetic coder encodes the message close to the entropy),
- bits per encoded symbol — this number should be acquired by dividing the entropy of the encoded message by the number of encodings,
- how many percents of the overall output have been output by this trie (we will first need to calculate a sum of entropy of the encoded message for all tries used during the compression).

This should give us a good knowledge of which tries affect the compression ratio the most, which tries are or are not efficient etc.

2.8 Formal specification of experiments

As the whole analysis is finished, we can finally present the full and formal specification of each of the compression experiments proposed in Section 2.4 (for more info about the individual models and algorithm steps, please refer to that section). We only show the encoding algorithms here, since the decoding algorithms are just inverse to the encoding algorithms. We remind that each of the models mentioned below is represented by an individual PPM trie. When we use a variable k_x in connection with any model M_x in the following text, then k_x is order of the trie which represents model M_x (the orders are specified later in this thesis).

2.8.1 Basic word-based compression

Used models and their properties:

1. model of black tokens M_{BT}; this model is used for prediction of the next black token based on previous k_{BT} black tokens encoded/decoded by this model;
    - the model uses dynamically growing alphabet A_{BT} (the initial size of the alphabet is 1, containing only a special symbol signalizing a not-yet-known token);
2. models of white tokens M_{WT_c} for every Unicode character c that has appeared as last character of any black token; these models are used for prediction of the next white token based on the last character c of the preceding black token and k_{WT_c} white tokens that have most recently appeared after a black token containing c as the last character;
    - all models M_{WT_c} use a shared, dynamically growing alphabet A_{WT} (there is again a special symbol signalizing an unknown token);
3. model of black token characters M_{BC}; this model is used for prediction of the next character based on previous k_{BC} characters encoded/decoded by this model; we remind that this model is used to encode/decode not-yet-known black tokens;
    - the model uses alphabet A_{BC} of size 257 (one symbol for every possible value of a byte and one special symbol to signalize end of the string);
4. model of white token characters M_{WC}; this model is used for prediction of the next character based on previous k_{WC} characters encoded/decoded by this model; we remind that this model is used to encode/decode not-yet-known white tokens;

- the model uses alphabet A_{WC} of size 257 (one symbol for every possible value of a byte and one special symbol to signalize end of the string);

5. model of letter cases M_{LC} (used by the letter case heuristics); this model is used for prediction of the case of the first letter of the following black token based on k_{LC} previous cases encoded by this model;

   - the model uses alphabet A_{LC} of size 2 (one symbol represents the upper case, one symbol the lower case).

Algorithm description (encoding loop):

1. If no black token can be read from the input, encode an EOF using M_{BT} and terminate the loop.
2. Read black token BT from the input.
3. If the letter case heuristics is in activated state and BT allows transition to deactivated state, modify BT as needed (BT_1 is the result of modification).
4. Encode BT_1 using M_{BT} (known token) or $M_{BT} + M_{BC}$ (not-yet-known token).
5. Encode case of the first character of BT using M_{LC} if the letter case heuristics switched from activated state to deactivated state in step 3.
6. Set c to last Unicode character of BT.
7. If no white token can be read from the input, encode an EOF using M_{WT_c} and terminate the loop.
8. Read the white token WT from the input and encode it using M_{WT_c} or $M_{WT_c} + M_{WC}$.

An important thing to mention is that an EOF can be decoded only using M_{BT} and M_{WT_c}, thus decoding EOF with any other model indicates an error (we don't point this out in description of the other experiments; if EOF is decoded by a model which never encodes EOF during encoding, it always means an error).

2.8.2 Basic experiment using morphological generator

Used models and their properties:

1. all models and their alphabets specified in Section 2.8.1 (models M_{BT} and M_{BC} are in fact used to encode/decode lemmas and pseudolemmas instead of real black tokens during this experiment);

2. model of indexes M_{I_L} for each lemma or pseudolemma L; each of the models M_{I_L} is used for prediction of the next form generated from lemma or pseudolemma L based on previous k_{I_L} indexes encoded/decoded by this model (we remind that each index specifies a unique word form);

    - the size of the alphabet A_{I_L} of each model is equal to number N of forms generated from L and an empty tag if L is among the generated forms, or to $N+1$ if L is not among the generated forms;

Algorithm description (encoding loop):

1. The same as step 1 in Section 2.8.1.
2. The same as step 2 in Section 2.8.1.
3. The same as step 3 in Section 2.8.1.
4. Get the lemma L of BT.
5. Strip L to either raw lemma or lemma id depending on the algorithm settings (L_1 is the stripped lemma).
6. Decide whether to store L_1 (lemma) or BT_1 (pseudolemma) (L_2 is the result of this choice).
7. Encode L_2 using M_{BT} or $M_{BT} + M_{BC}$.
8. Encode index using $M_{I_{L_2}}$ if needed to identify BT_1 among the forms generated from L_2.
9. The same as step 5 in Section 2.8.1.
10. The same as step 6 in Section 2.8.1.
11. The same as step 7 in Section 2.8.1.
12. The same as step 8 in Section 2.8.1.

2.8.3 Experiment with part-of-speech tags (1)

Used models and their properties:

1. all models and their alphabets specified in Section 2.8.1 except the model of black tokens M_{BT};
2. model of POS tags M_{POS}; this model is used for prediction of the POS tag of the next black token based on POS tags of previous k_{POS} black tokens;

- the model uses alphabet A_{POS} containing 12 symbols (this is the number of distinct values which may appear on the first position in MorphoDiTa tags);

3. model of black tokens M_{BT_X} for every value X of POS tag (i.e. 12 distinct models); each of the models M_{BT_X} is used to estimate the next black token with POS tag equal to X based on previous k_{BT_X} black tokens with POS tag equal to X;

   - each of the models M_{BT_X} uses dynamically growing alphabet A_{BT_X} (the initial size of the alphabet is 1, containing only a special symbol signalizing a not-yet-known token);

Algorithm description (encoding loop):

1. If no black token can be read from the input, encode an EOF using M_{POS} and terminate the loop.
2. The same as step 2 in Section 2.8.1.
3. Get the POS tag of BT and encode the value X of the POS tag using M_{POS}.
4. The same as step 3 in Section 2.8.1.
5. Encode BT_1 using M_{BT_X} or $M_{BT_X} + M_{BC}$.
6. The same as step 5 in Section 2.8.1.
7. The same as step 6 in Section 2.8.1.
8. The same as step 7 in Section 2.8.1.
9. The same as step 8 in Section 2.8.1.

2.8.4 Experiment with part-of-speech tags (2)

Used models and their properties:

1. all models and their alphabets specified in Section 2.8.1 except the model of black tokens M_{BT};
2. model of black tokens M_{BT_X} for every value X of POS tag (i.e. 12 distinct models); each of the models M_{BT_X} is used for prediction of the next black token appearing after a black token with POS tag equal to X based on previous k_{BT_X} black tokens encoded by the model;

- all these models share a single alphabet of black tokens A_{BT} (the initial size of the alphabet is 1, containing only a special symbol signalizing a not-yet-known token).

Algorithm description:

Before entering the encoding loop, set variable X (previous part-of-speech tag) to value 'Z' (at the start of the text, we simulate this way that there was some preceding sentence, since part-of-speech of "." is 'Z').

Encoding loop:

1. If no black token can be read from the input, encode an EOF using M_{BT_X} and terminate the loop.
2. The same as step 2 in Section 2.8.1.
3. The same as step 3 in Section 2.8.1.
4. Encode BT_1 using M_{BT_X} or $M_{BT_X} + M_{BC}$.
5. The same as step 5 in Section 2.8.1.
6. Update the value of X: Use isolated tagging to get the part-of-speech tag of BT_1 and set X to this value (we don't use BT here since the first letter of BT_1 is in expected case, which may not be true for BT).
7. The same as step 6 in Section 2.8.1.
8. The same as step 7 in Section 2.8.1.
9. The same as step 8 in Section 2.8.1.

2.8.5 Experiment with non-part-of-speech tags

We remind that this experiment has two input parameters: list *selectedPOS* specifying for which parts of speech we want to store the tag, and number *tagPosition* specifying which position in the tag we want to store. We define the following variables and symbolic notation:

- *TAG_VALS* is the set of values which may occur in the tag at position *tagPosition*;
- *VAL_COUNT* is the size of set *TAG_VALS*;
- T_{TAG} is an empty tag where the position *tagPosition* has been set to value *TAG*.

Used models and their properties:

1. all models and their alphabets specified in Section 2.8.2 except M_{I_L};
2. model M_{TAG} for storing the tag values appearing on position *tagPosition*; this model is used for prediction of the next tag value from set TAG_VALS based on previous k_{TAG} values stored by the model;
   - this model uses alphabet A_{TAG} containing VAL_COUNT symbols;
3. model of indexes $M_{I_{L,T_{TAG}}}$ for each lemma or pseudolemma L and tag T_{TAG} (TAG is any value from set $TAG\_VALS \cup$ '?'); each of the models is used for prediction of the next form generated from lemma or pseudolemma L and tag T_{TAG} based on previous $k_{I_{L,T_{TAG}}}$ indexes encoded/decoded by this model (we remind that each index specifies a unique word form);
   - each of the models uses alphabet $A_{I_{L,T_{TAG}}}$; the size of the alphabet is equal to number N of forms generated from L and T_{TAG} if L is among the generated forms, or to $N + 1$ if L is not among the generated forms).

Algorithm description:

1. The same as step 1 in Section 2.8.1.
2. The same as step 2 in Section 2.8.1.
3. The same as step 3 in Section 2.8.1.
4. The same as step 4 in Section 2.8.2.
5. Strip L to raw lemma L_1 (we don't strip to lemma id since a lemma id may contain some technical info which would ruin the accuracy of isolated tagging).
6. If the POS tag of L_1 (acquired by isolated tagging) is among values in *selectedPOS*, then decide (using L_1) which tag value TAG from set $TAG\_VALS \cup$ '?' should be stored as described in Section 2.4.5; also determine the set of pre-excluded values $PREEXCLUDED$.
7. Using T_{TAG} for generation, decide whether to store L_1 (lemma) or BT_1 (pseudolemma) (L_2 is the result of this choice); this step is similar to step 6 in Section 2.8.2.
8. If L_2 is not equal to L_1, repeat step 6 using L_2 instead of L_1 to get new values of TAG and $PREEXCLUDED$.
9. The same as step 7 in Section 2.8.2.

10. If TAG is not equal to '?', then encode the value of TAG using model M_{TAG} (the encoder pre-excludes symbols contained in $PREEXCLUDED$).

11. Encode index using $M_{I_{L_2,T_{TAG}}}$ if needed to identify BT_1 among the forms generated from L_2 and T_{TAG} (this step is similar to step 8 in Section 2.8.2).

12. The same as step 5 in Section 2.8.1.

13. The same as step 6 in Section 2.8.1.

14. The same as step 7 in Section 2.8.1.

15. The same as step 8 in Section 2.8.1.

Chapter 3

Design

This chapter describes the design of our compression program, which is based on the previous analysis.

The architecture of the program is mostly object-oriented. Moreover, our program is logically divided into the following parts:

- *MorphoDiTa static library* and *MorphoDiTa header file* (an interface between MorphoDiTa and our program),
- *language tools* — various tools for processing the text files (this includes conversion from or to UTF-8 encoding and wrappers of MorphoDiTa tools),
- *input/output* — classes responsible for managing input and output of the program (including classes for preprocessing of text files as described in Section 2.2.2),
- *range coder* — arithmetic coder used by PPM,
- *PPM structure* — data structures used within the PPM algorithm,
- *PPM coder* — the PPM algorithm itself,
- *language compression* — classes implementing the compression algorithms specified in Section 2.8,
- *main file* — an entry point of the program, parsing the command line arguments and launching the implemented algorithms accordingly.

In the following sections, we give a more detailed description of each of the parts (except MorphoDiTa library and MorphoDiTa header file, which have already been described). We focus here on general description of functionality of individual classes and on description of public methods of the classes;

for an even more detailed description, please refer to the attached Doxygen-generated[44] documentation of the whole program. Additionally, we show much of the program architecture on three class diagrams: Figure A.1 shows the most important classes related to range coder, PPM coder and PPM structure, Figure A.2 shows some of the most important classes related to input and language tools and Figure A.3 shows classes representing the so-called *compression units* which are described further in this chapter.

The design of PPM coder, PPM structure and range coder is heavily influenced by the original implementation by Jiří Krotil, although a big number of important changes has been introduced, as proposed.

When an error occurs during the execution of the program, immediate termination of the program and displaying a brief error message is an acceptable behaviour for us.

To make the debugging easier, all compression experiments first compress the specified file and then immediately decompress it to verify there is no error in the compression program (the content of the decompressed file must be equal to the content of the original file). Moreover, during encoding, the encoded symbols are being stored in a special array. When decoding, the decoded symbols are being stored in another array and compared with the symbols stored in the first array to discover any mismatch as early as possible. We decided to keep this mechanism in the final implementation; if any mismatch between the original and the decompressed file occurs (which is not expected to happen with any of the implemented algorithms), it is always detected and the program is terminated with an error.

3.1 Language tools

3.1.1 Class UTF8Helper

This class is used for processing UTF-8-encoded text as described in Section 2.5; all methods of this class are static (no instances of this class should be created). Instead of enumerating all the methods of this class, we show here just a general description of which methods are contained in this class:

- methods for recognizing Unicode characters in an UTF-8-encoded string;
- methods for generating UTF-8-encoded byte sequences for given Unicode characters;
- methods for converting Unicode characters to upper/lower case (these methods only know how to convert letters with diacritics which are contained in Czech alphabet).

3.1.2 Class TokenizerWrapper

This is an interface (abstract class) for tokenizer classes. It includes the following methods:

- `virtual void setText (string& text) = 0`
  – sets text to be tokenized;

- `virtual bool tokenizeNext (`
  `vector<token_byte_range>& tokenByteRanges`
  `) = 0`
  – fills the parameter array with a new bunch of tokens; returns false if the text has already been fully tokenized (no more tokens to process); `token_byte_range` is a simple structure holding byte range of a token.

3.1.3 Class BasicTokenizer

Child class of `TokenizerWrapper`, implementing its abstract methods. This class uses MorphoDiTa class `tokenizer` to tokenize the text (with each call to `BasicTokenizer::tokenizeNext`, `tokenizer::next_sentence` is called). Apart from the inherited methods, the class contains the following public method:

- `BasicTokenizer (ufal::morphodita::tokenizer* tknzr)`
  – constructor accepting a pointer to MorphoDiTa tokenizer.

3.1.4 Class TaggerWrapper

This is an interface (abstract class) for tagger classes. It includes the following method:

- `virtual void tag (`
  `const vector<string>& tokens,`
  `vector<tag_and_lemma>& taggedLemmas`
  `) const = 0`
  – method which assigns a tag and a lemma to every token in parameter `tokens`, parameter `taggedLemmas` then holds the result of tagging (`tag_and_lemma` is a simple structure holding tag and lemma of a single token).

3.1.5 Class BasicTagger

Child class of `TaggerWrapper`, implementing its abstract method. This class uses MorphoDiTa class `tagger` to tag the tokens (with each call to `BasicTagger::tag`, `tagger::tag` is called). Apart from the inherited method, the class contains the following public methods:

- `BasicTagger (ufal::morphodita::tagger* tgr)`
  – constructor accepting a pointer to MorphoDiTa tagger;

- `ufal::morphodita::tokenizer* loadTokenizer () const`
  – loads instance of MorphoDiTa tokenizer associated with the MorphoDiTa tagger (using MorphoDiTa method `tagger::new_tokenizer`);

- `const ufal::morphodita::morpho* loadMorpho () const`
  – loads instance of MorphoDiTa generator associated with the MorphoDiTa tagger (using MorphoDiTa method `tagger::get_morpho`).

3.1.6 Class MorphoWrapper

This is an interface (abstract class) for generator classes. It includes the following methods:

- `virtual string stripLemmaToUniqueId (const string& lemma) const = 0;`

- `virtual string stripLemmaToRawLemma (const string& lemma) const = 0;`

- ```
  virtual void generateForms (
    const string& lemma,
    const string& tag,
    vector<string>& forms
  ) const = 0
  ```
  – method generating forms from the given lemma and tag; if `lemma` is not a unique identifier (e.g. a raw lemma) and the generator finds multiple corresponding lemmas, the generation results are merged to a single array (notice that multiple corresponding lemmas may occasionally be found even if using a lemma id); the result of generation `forms` doesn't contain duplicate items.

3.1.7 Class BasicMorpho

Child class of `MorphoWrapper`, implementing its abstract methods. This class uses MorphoDiTa class `morpho` to strip the lemmas and to generate the forms (with each call to `BasicMorpho::generateForms`, `morpho::generate` is called with guesser mode enabled — MorphoDiTa ensures that the guesser mode is not used if at least one lemma is found in the dictionary). Apart from the inherited methods, the class contains the following public method:

- `BasicMorpho (const ufal::morphodita::morpho* m)`
  – constructor accepting a pointer to MorphoDiTa generator.
  ```

3.1.8 Class LanguageToolsWrapper

This is an interface (abstract class) for classes holding all the linguistic tools. It contains the following methods:

- `virtual const TaggerWrapper* getTagger () const = 0;`
- `virtual TokenizerWrapper* getTokenizer () const = 0;`
- `virtual const MorphoWrapper* getMorpho () const = 0;`
- `virtual bool toolsSuccesfullyLoaded () const = 0`
  – returns true if all three tools (tagger, tokenizer and generator) are ready for use, false otherwise.

3.1.9 Class MorphoditaToolsWrapper

Child class of `LanguageToolsWrapper`, implementing its abstract methods. This class wraps the MorphoDiTa tools used in this thesis (tagger, tokenizer and generator). Apart from the inherited methods, the class contains the following public method:

- `MorphoditaToolsWrapper (const char* pathToTagger)`
  – constructor accepting a path to MorphoDiTa tagger model (the tokenizer and generator are loaded from the tagger instance).

3.1.10 Class TagHelper

This class supports working with MorphoDiTa tags. The most important responsibility of this class is to ensure that a tag is valid (containing only known values specified in [26]) and giving info on values available on a specific position in the tag. The class contains following public methods (all methods of this class are static, no instances of this class should be created):

- ```
  static uint8_t getNthPositionVal (
    unsigned int n,
    const string& tag,
    bool failIfUnknown
  )
  ```
  – returns the `n`-th value in the tag; the main purpose of this method is to make sure that the value is known; parameter `failIfUnknown` specifies whether the method should fail if an unknown value is encountered (if set to true, the program is terminated, if false, the function returns '?', which is a wildcard);
  ```

- `static void setNthPositionVal (`
  `unsigned int n,`
  `string& tag,`
  `uint8_t val`
  `)`
  – sets the n-th position in the tag to the given value (the program is terminated if the value is unknown);

- `static string getEmptyTag ()`
  – returns an empty tag;

- `static unsigned int`
  `getNumberOfValuesAtPos (unsigned int pos)`
  – returns how many distinct values may appear on the given position in the tag;

- `static vector<uint8_t> getNthPositionValues (unsigned int n)`
  – returns the list of values which may appear on the `n`-th position in the tag;

- `static unsigned int getTagLength ()`
  – returns the length of MorphoDiTa tags.

3.2 Input/output

3.2.1 Class ByteInputModule

This class is used for reading the input file byte by byte. It contains the following methods:

- `ByteInputModule (FILE* i)`
  – constructor accepting a pointer to the input file;

- `uint8_t getByte ()`
  – returns the next byte from the input file; after this method has been called, method `isEndOfInput()` must always be called to check for EOF/error;

- `bool isEndOfInput ()`
  – returns true if the end of input has been reached, false otherwise; the program is terminated if the end of input is caused by some error.

3.2.2 Class ByteOutputModule

This class is used for writing the output byte by byte to a file. List of methods:

- `ByteOutputModule (FILE* o)`
  – constructor accepting a pointer to the output file;

- `void putByte (uint8_t byte)`
  – outputs the given byte; the program is terminated if some error occurs.

3.2.3 Class InputReaderWrapper

This class is an interface (abstract class) wrapping the input readers used to read the text file before delivering the text to a tokenizer (see Section 2.2.2.2). It contains just one, abstract method, which is to be overridden by the child classes:

- `virtual bool getNextTextBlock (string& buffer) = 0`
  – reads the next piece of text; returns false if EOF has been reached.

3.2.4 Class DummyInputReader

This is a child of class `InputReaderWrapper`. It reads the whole input text at once. Public methods of this class include:

- `DummyInputReader (const char* inputPath)`
  – constructor accepting path to the input file which should be opened an read; if the argument is null, standard input is used;

- `bool getNextTextBlock (string& buffer)`
  – implementation of the abstract method inherited from parent (the first call of the method delivers the whole input text, the following calls return false).

3.2.5 Class FeedingModule

This is an interface (abstract class) for classes which deliver the input text as a sequence of tokens (with tag and lemma for every black token) without adding any "artificial" empty tokens (see Section 2.2.2.3 where the idea of adding empty tokens is described). Such classes are intended to work in steps — in each step, a new token from the input is read and processed. Class methods:

- `virtual bool stepForward () = 0`
  – prepares the next token from the input (also lemma and tag, if the token is black); returns false if end of input has been reached;

- `virtual string getLastToken () const = 0`
  – returns token which was acquired in the last step;

- `virtual bool isLastTokenWhite () const = 0`
  – returns true if the current token is white, false otherwise;

- `virtual string getLastLemma () const = 0`
  – returns lemma of the current token (returns a valid value only if the current token is black);

- `virtual string getLastTag () const = 0`
  – returns tag of the current token (returns a valid value only if the current token is black).

3.2.6 Class BasicFeedingModule

This is a child class of `FeedingModule` implementing its abstract methods. It uses `DummyInputReader` to read the input text, `TokenizerWrapper` to tokenize the text and `TaggerWrapper` to do the tagging and lemmatisation. Apart from the inherited methods, the class contains the following public method:

- ```
  BasicFeedingModule (
      const TaggerWrapper* tgr,
      TokenizerWrapper* tknzr,
      const char* inputPath
  )
  ```
  – constructor accepting pointers to tagger and tokenizer which should be used to process the text and path to input file (standard input is used if the path equals null).

3.2.7 Class TokenStreamNormalizer

This class uses `FeedingModule` to get the stream of tokens, lemmas and tags and modifies the stream as needed to ensure that the stream always starts with a black token and that the white and black tokens alternate regularly (see Section 2.2.2.3). The class contains the following public methods:

- `TokenStreamNormalizer (FeedingModule* fm)`
  – constructor accepting a pointer to `FeedingModule` which should be used to process the text;

- `string getToken () const`
  – returns token which was acquired in the last step;

- `string getLemma () const`
  – returns lemma of the current token (returns a valid value only if the current token is black and non-empty);
  ```

- `string getTag () const`
  – returns tag of the current token (returns a valid value only if the current token is black and non-empty);

- `bool stepForward ()`
  – prepares the next token + lemma + tag from the input; returns false if there is no more input.

3.3 Range coder

3.3.1 Class RangeCoder

This class holds the functionality and member variables common for both encoder and decoder. It holds the variables `Low` and `High` which are initialized to values of 0 and `UINT64_MAX >> 16`, respectively (both values are stored as unsigned 64-bit integers, while the effective size of the numbers is 48 bits). The class also implements methods for recomputing and rescaling `Low` and `High`, this includes outputting the common leftmost digits and underflow prevention. No instances of this class should be created.

3.3.1.1 Coder limits

The arithmetic coder, of course, has some numerical limits — the values of `Low` and `High` must not cross during the coding so that they always represent a valid interval (the value of `High` must always be greater than the value of `Low`).

According to [45], to prevent `Low` and `High` from crossing, the value of `TotalFreq` must be less than 2^{N-2} when we are using N bits to hold the values of `Low` and `High`; this condition is fulfilled in our coder, since `TotalFreq` is stored in an unsigned 32-bit integer and we are using 48 bits for `Low` and `High`.

However, when recomputing the values of `Low` and `High` as described in Section 1.5.4, the 64-bit integer holding the value of `High` may overflow if the value of `HighCumFreq` is greater than 65535. Since `HighCumFreq` is lower than or equal to `TotalFreq`, this limit is also valid for the value of `TotalFreq`. Using this restriction, we also prevent another possible overflow when calculating the index during decoding (described in Section 1.5.4.2).

Since the frequencies of symbols are stored as integers, this gives us a limit on the size of the alphabet used during coding — it cannot be greater than 65535 (in PPM, this includes also the escape symbol). This is acceptable for us, as proposed in Section 2.6.

3.3.2 Class RangeEncoder

This class is a child of `RangeCoder`, representing the arithmetic encoder. The class contains the following public methods:

- `RangeEncoder (ByteOutputModule *o)`
  – constructor accepting an instance of byte output module, which is used to output the compressed data;

- `void encodeRangeOfSymbol (`
  `const INTERVAL* interval,`
  `uint32_t totalCount`
  `)`
  – method which encodes a symbol specified by range `interval` (`INTERVAL` is a structure holding `LowCumFreq` and `HighCumFreq` of a symbol); the second parameter specifies `TotalFreq`;

- `void flushOutputOfRangeCoder ()`
  – flushes the rest of code and pending underflow bits to the output (used after EOF has been encoded).

3.3.3 Class RangeDecoder

This class is a child of `RangeCoder`, representing the arithmetic decoder. The class contains the following public methods:

- `RangeDecoder (ByteInputModule *i)`
  – constructor accepting an instance of byte input module, which is used to read the compressed data;

- `uint32_t computeIndex (uint32_t totalCount) const`
  – returns a number from the range corresponding to the decoded symbol (see computing of index in Section 1.5.4.2); `totalCount` specifies `TotalFreq`;

- `void decodeRangeOfSymbol (`
  `const INTERVAL* interval,`
  `uint32_t totalCount`
  `)`
  – updates `Low` and `High` after decoding the symbol specified by its range (`interval`); the second parameter specifies `TotalFreq`.

3.4 PPM structure

3.4.1 Class PPMNode

This abstract class represents a node in the trie. It holds the variables and functionality common for both inner (internal) and leaf nodes of the trie (as already stated earlier in this thesis, inner nodes represent contexts and thus

they don't appear in the deepest layer of the trie structure, while leaf nodes only appear in the deepest layer — if the order of PPM is N, then leaf nodes only appear in depth $N + 1$).

An instance of class `PPMNode` holds info about symbol, count and suffix link (see Section 1.6.4). Moreover, as the child nodes of a node are held in a linked list, an instance of `PPMNode` also holds a pointer to its right sibling node in the list.

The most important methods among the public ones are getters of symbol (`getSymbol`), count (`getCount`) and the target node of a suffix link (`getSuffixContextNode`); otherwise, the real structure of the trie (e.g. storing the child nodes in a linked list) is hidden from the PPM coder and the coder doesn't directly modify the nodes (this is to separate the coder from the trie structure as proposed in Section 2.6.2).

3.4.2 Class PPMLeafNode

This class represents a leaf node of the trie, it is a child class of `PPMNode`. The class adds no variables or functionality when compared with the parent class, it just identifies itself as a leaf node.

3.4.3 Class PPMInnerNode

This class represents an inner node of the trie, it is a child class of `PPMNode`. Apart from variables inherited from the parent, it holds info on number of child nodes, sum of counts in the child nodes, order (depth) of the node within the trie and pointers to the start and end of the linked list of child nodes.

Important public methods include getters of order (`getOrder`), number of childs (`getNumberOfChilds`) and sum of counts in the child nodes (`getSumOfChildCounts`); again, the PPM coder doesn't directly modify the nodes nor it knows the real structure used for storing the trie.

3.4.4 Class ExclusionList

This class represents a list of symbols excluded during the PPM encoding or decoding (when using the exclusion principle). It contains methods for reinitialization of the list, setting an symbol excluded, getting number of excluded symbols and getting info on whether a specific symbol is excluded or not.

3.4.5 Class InnerNodeBackupForExclusion

We were thinking about two different ways how to implement the symbol exclusion in a given context — either to temporarily modify the context node (so that its list of child nodes doesn't include nodes representing the excluded symbols) or to keep the trie structure unmodified and force the coder to skip

the excluded symbols. We chose the first way since it seemed easier to implement.

Class `InnerNodeBackupForExclusion` is used to backup the member variables of an instance of `PPMInnerNode` when the node gets temporarily modified because of exclusion; once the exclusion in the node is no more needed, the node is restored from the backup. Apart from a trivial constructor, the class contains the following public methods:

- ```
  void backupAndExclude (
    PPMInnerNode* nodeToBackup,
    const ExclusionList* list
  )
  ```
  – a method which backups the member variables of the specified node and then modifies the node to exclude the symbols specified by the exclusion list;

- `void restoreFromBackup ()`
  – sets the previously backed-up node to its original state.

3.4.6 Memory manager

This class is used for allocating nodes of the trie structure; the nodes are allocated in groups, which makes the memory allocation efficient. Original functionality of reallocating the model if it gets too big has been disabled by setting the memory limit to 4 GB; if the memory limit is exceeded anyway, the program terminates with an error (this is not expected to happen in our experiments unless the order of some PPM trie is set to some extremely big value). The class contains the following public methods:

- `MemoryManager ()`
  – constructor; allocates the first portion of leaf nodes and inner nodes (the next portion is always allocated when the previous portion has been exhausted);

- `void freeAllocatedMemory ()`
  – deallocates all nodes allocated so far;

- `PPMInnerNode* getNewNode ()`
  – returns a new instance of `PPMInnerNode`;

- `PPMLeafNode* getNewLeafNode ()`
  – returns a new instance of `PPMLeafNode`;

- `void checkModelSize (PPMCoder* ppmCoder)`
  – a method intended to check the size of the model (trie structure) and
  ```

to do the reallocation of the model if it gets too big; the functionality of this method is currently disabled as mentioned above.

3.4.7 Class PPMTrieConfig

Class holding info about some of the parameters of a trie, namely order of the trie, current size of the alphabet used by the trie and current limit on sum of symbol frequencies in a context (see Section 3.4.9.1 for description of this limit).

3.4.8 Class PPMTrieStatistics

Class holding info proposed in Section 2.7 for a specific trie, namely number of encodings carried out using this trie and entropy of the encoded message in bits (the name of the trie is held by class `PPMTrie`). The class variables are updated by classes `PPMEncoder` and `PPMCompressionSupport` (see further) during encoding.

3.4.9 Class PPMTrie

Class representing a trie used by the PPM algorithm. An instance of this class holds an instance of `PPMTrieConfig`, an instance of `PPMTrieStatistics`, a pointer to the root node of the trie, a pointer to memory manager (used to allocate new nodes), a string specifying the name of the trie (see Section 2.7) and a pointer serving as *base pointer* (described in Section 1.6.4).

While classes representing the nodes just hold the necessary values, class `PPMTrie` implements also some intelligence which is utilized by the PPM coder so that the PPM coder doesn't need to know all details about the actual implementation of the trie structure. Apart from trivial getters of member variables, the class implements the following public methods:

- ```
  PPMTrie (
    MemoryManager* mm,
    PPMTrieConfig& config,
    string name
  )
  ```
  – constructor accepting a pointer to memory manager, a trie config and desired name of the trie;

- `void updateAfterSymbolRead (SYMBOL_ID symbol)`
  – method updating the trie with an incoming symbol;

- ```
  PPMNode* findSymbolInContextSimple (
    const PPMInnerNode* context,
    SYMBOL_ID symbol
  ```

```
)
```
– method which, for the given context node, returns its child node representing the given symbol or null if no such child node exists;

- `PPMNode* findSymbolInContext (`
  `const PPMInnerNode* context,`
  `SYMBOL_ID symbol,`
  `uint32_t& lowCum,`
  `uint8_t& position`
  `)`
  – an extended version of `findSymbolInContextSimple` which also calculates `LowCumFreq` of the given symbol and its position in the linked list (the position is used by PPM variants B and D);

- `PPMNode* getNodeUsingIndex (`
  `const PPMInnerNode* context,`
  `uint32_t index,`
  `uint8_t variant,`
  `uint32_t& lowCum`
  `) const`
  – method which, using `index` calculated by the range decoder, finds the corresponding node in the given context and calculates its `LowCumFreq` for the given PPM variant;

- `void incrementAlphabetSize ()`
  – method which extends the current size of the trie alphabet by one (used to increase the size of the alphabet when encountering new tokens in the word-based compression);

- `static void excludeSymbols (`
  `const PPMInnerNode* context,`
  `ExclusionList* excluded`
  `)`
  – method which fills `excluded` with all symbols that have appeared in the given context.

3.4.9.1 Rescaling symbol frequencies

With every update of the trie, the corresponding context node is inspected to verify that the current limit on sum of symbol frequencies in a context has not been exceeded for this context node. If the sum has reached the limit (specified by the instance of `PPMTrieConfig` held by `PPMTrie`), the frequencies are rescaled by dividing all symbol counts by 2 (if a symbol count reaches the

value of 0 this way, it is set to 1). This ensures that the sum doesn't grow over technical limits of the coder.

The initial value of the limit, which is specified when the trie is created, may be increased over time so that the model doesn't get too flat (this is done by a simple heuristics: the limit is doubled when the number of different symbols in some context reaches one quarter of the limit). However, the limit is being increased only until it reaches the value of 65535, because `TotalFreq` in the range coder cannot exceed this value (see Section 3.3.1.1); if the limit is already equal to 65535 and the number of different symbols in the context reaches 90% of the limit, the program is terminated to warn the user that the model is getting too flat (i.e. the observed frequencies of distinct symbols are mostly held on similar low values due to the rescaling).

3.5 PPM coder

3.5.1 Class PPMCoder

This is a class holding functionality and member variables common for both PPM encoder and PPM decoder, such as accepting a trie instance which should be used for the next encoding or decoding, updating the trie with the incoming symbol after the symbol is encoded/decoded, holding the list of excluded symbols etc. No instances of this class should be created. The coder supports all four PPM variants implemented by Jiří Krotil (A, B, C, D); however, the variant is currently not selectable and variant C is selected by default. Similarly, exclusion principle is used by default and we currently don't allow the user to turn it off.

We don't show any list of methods here since all methods of this class are rather technical, supporting the PPM algorithm which has already been described.

3.5.2 Class PPMEncoder

This class is a child class of `PPMCoder`, it is responsible for the encoding part of PPM algorithm. The class contains the following public methods:

- `PPMEncoder (RangeEncoder* re, MemoryManager* mm)`
  – constructor accepting an instance of range encoder (which is used as subprocedure by the PPM algorithm) and an instance of memory manager which is responsible for reallocation of the PPM model (trie) if the model gets too big (this functionality is currently disabled);

- ```
  void encodePPMSymbol (
    SYMBOL_ID symbol,
    PPMTrie* trie,
  ```
  ```

```
  const vector<SYMBOL_ID>* preexcludedSymbols
)
```
– method encoding the given symbol using the given trie (which is updated with the symbol after encoding) and a list of pre-excluded symbols;

- `void encodePPMEOF (`
  `PPMTrie* trie,`
  `const vector<SYMBOL_ID>* preexcludedSymbols`
  `)`
  – method encoding an end-of-file symbol using the given trie (should be called after the last symbol from input has been encoded).

3.5.3 Class PPMDecoder

This class is another child class of `PPMCoder`, it is responsible for the decoding part of PPM algorithm. The class contains the following public methods:

- `PPMDecoder (RangeDecoder* rd, MemoryManager* mm)`
  – constructor accepting an instance of range decoder (which is used as subprocedure by the PPM algorithm) and an instance of memory manager (the functionality of reallocating the model is disabled);

- `SYMBOL_ID decodeNextPPMSymbol (`
  `PPMTrie* trie,`
  `const vector<SYMBOL_ID>* preexcludedSymbols`
  `)`
  – returns a symbol from the trie alphabet if not EOF (after calling this method, method `hasEOFBeenDecoded` must be used to check whether the returned symbol is a valid symbol from the trie alphabet);

- `bool hasEOFBeenDecoded ()`
  – returns true if the EOF symbol has been decoded (this means that the whole decoding process is finished).

3.5.4 Class PPMEncoderWrapper

This class wraps the whole PPM encoding process. It holds a pointer to an instance of `ByteOutputModule`, an instance of `RangeEncoder`, an instance of `PPMEncoder` and an instance of `MemoryManager`, which are all needed during encoding. The class contains the following public methods:

- `PPMEncoderWrapper (ByteOutputModule* outputModule)`
  – constructor accepting a pointer to an instance of byte output module which should be used to output the compressed data (the instances of

range encoder, PPM encoder and memory manager are created within this constructor);

- `void encodeNextSymbol (`
  `PPMTrie* trie,`
  `SYMBOL_ID symbol,`
  `const vector<SYMBOL_ID>* preexcludedSymbols`
  `)`
  – calls `PPMEncoder::encodePPMSymbol` with the given parameters;

- `void encodeEOF (`
  `PPMTrie* trie,`
  `const vector<SYMBOL_ID>* preexcludedSymbols`
  `)`
  – calls `PPMEncoder::encodePPMEOF` with the given parameters;

- `MemoryManager* getMemoryManager ()`
  – returns the instance of memory manager (we need this to enable allocating new tries from the outside since all tries are currently allocated using a single instance of memory manager).

3.5.5 Class PPMDecoderWrapper

This class wraps the whole PPM decoding process. It holds a pointer to an instance of `ByteInputModule`, an instance of `RangeDecoder`, an instance of `PPMDecoder` and an instance of `MemoryManager`, which are all needed during decoding. The class contains the following public methods:

- `PPMDecoderWrapper (ByteInputModule* inputModule)`
  – constructor accepting a pointer to an instance of byte input module which should be used to read the compressed data (the instances of range decoder, PPM decoder and memory manager are created within this constructor);

- `bool decodeNextSymbol (`
  `PPMTrie* trie,`
  `SYMBOL_ID& decodedSymbol,`
  `const vector<SYMBOL_ID>* preexcludedSymbols`
  `)`
  – this method calls `PPMEncoder::decodeNextPPMSymbol`, the return value of which is used to set the value of `decodedSymbol`, and returns true if end of file has not yet been reached;

- `MemoryManager* getMemoryManager ()`
  – analogous to `PPMEncoderWrapper::getMemoryManager`.

3.6 Language compression

In this section we describe classes which have been designed to support the compression experiments and then the experiments themselves. A special subsection is dedicated to so-called *compression units* which are used as building blocks of the proposed compression experiments.

The order of many tries used in the compression experiments is preset to some reasonable value which has been acquired experimentally and the user cannot set it manually (we don't show the experiments here since this is not what we should be focusing on in this thesis). If we don't allow setting the order for a specific trie, it is set to some default value as described. Moreover, the values of initial limits on sum of symbol frequencies in a given context are always preset. It is ensured that a specific model always has the same default values across all word-based compression experiments, so that the experiments are comparable.

To be more specific, the orders are preset to the following values:

- The order of models of black tokens (e.g. M_{BT}) is preset to 1 in all experiments.
- The order of models of white tokens M_{WT_c} is preset to 0 in all experiments.
- The order of models of black token characters M_{BC} is preset to 4 in all experiments.
- The order of models of white token characters M_{BC} is preset to 1 in all experiments.
- The order of models of letter cases M_{LC} is preset to 0 in all experiments.

Otherwise, the order is selectable by the user or the order is specified in the following text.

3.6.1 Support classes

3.6.1.1 Tiny support classes

We briefly summarize here five support classes with simple functionality:

- class `AllocationManager<T>`
  – class used to store pointers to dynamically allocated data structures of any type `T` (e.g. tries); the main purpose of this class is making the deallocation of these structures easy and automated;

- class `MappingToID<T>`
  – class mapping data structures of any type `T` to symbol IDs (the main purpose of this class in our experiments is to map tokens to IDs during the compression since the PPM algorithm only works with IDs);

- class `MappingFromID<T>`
  – class mapping symbol IDs to data structures of any type `T` (the main purpose of this class in our experiments is to map IDs to tokens during the decompression);

- class `CustomContext<A,B>`
  – class used for mapping data structures of any type `A` to data structures of any type `B` (this class is used for example when we want to keep different models for different contexts and it's impossible to achieve this using a single trie, e.g. when we want to have different models of black tokens depending on stored part of speech);

- class `SharedSymbolsManager`
  – class used when multiple tries share the same dynamic alphabet; it manages the alphabet size and updates the tries so that all the tries have the same alphabet when the size of the alphabet is incremented.

3.6.1.2 Class PPMSupport

This class holds various variables and methods which are commonly used during compression/decompression. The class is responsible e.g. for opening and closing input and output files (except files used by input readers described in Section 3.2.3) and for allocating and deallocating PPM tries used during the compression. No instances of this class should be created (we create only instances of child classes). Public methods of this class include:

- `void closeFiles ()`
  – method closing all files opened by this class;

- `uint8_t readOrder ()`,
  `void writeOrder (uint8_t order)`
  – methods reading/writing order of a trie from/to a file;

- `uint8_t readByte ()`,
  `void writeByte (uint8_t byte)`
  – methods reading/writing a byte from/to a file
  using a `ByteInputModule`/`ByteOutputModule`;

- `bool isEndOfInput ()`
  – method returning true if end of the input file has been reached;

- `PPMTrie* createNewTrie (PPMTrieConfig cfg, string trieName)`
  – method creating a new trie with the given configuration and name (class `PPMSupport` is responsible for deallocation of this trie);

- `static off_t getFileSize (const char* fileName)`
  – method returning size of the specified file;

- ```
  static double getCompressionRatio (
    const char* origFile,
    const char* compressedFile
  )
  ```
  – calculates achieved compression ratio (returns size of the compressed file divided by the size of the original file);

- ```
  static bool filesEqual (
    const char* file1,
    const char* file2
  )
  ```
  – returns true if the specified files have identical content (a debug method).

3.6.1.3 Class PPMCompressionSupport

This is a child class of `PPMSupport`, specialized on the encoding process. Apart from the inherited variables, it includes an instance of `PPMEncoderWrapper`. The public methods of this class include:

- ```
  PPMCompressionSupport (
    const char* inputFileName,
    const char* outputFileName
  )
  ```
  – constructor accepting path to input/output file (either of the two parameters can be null if the file is not needed);

- ```
  void encodeSymbol (
    PPMTrie* trie,
    SYMBOL_ID symbol,
    const vector<SYMBOL_ID>* preexcludedSymbols = NULL
  )
  ```
  – this method calls `PPMEncoderWrapper::encodeNextSymbol` with the given parameters; moreover, it stores the encoded symbol into a debug array and updates the trie statistics, namely the number of encoded symbols (entropy of the encoded message is updated by `PPMEncoder`);

- ```
  void encodeEOF (
    PPMTrie* trie,
  ```
  ```

```
  const vector<SYMBOL_ID>* preexcludedSymbols = NULL
)
```
– calls `PPMEncoderWrapper::encodePPMEOF` with the given parameters;

- `void printFinalSummary ()`
  – this method prints the trie statistics at the end of the encoding process as proposed in Section 2.7.

3.6.1.4 Class PPMDecompressionSupport

This is a child class of `PPMSupport`, specialized on the decoding process. Apart from the inherited variables, it includes an instance of `PPMDecoderWrapper`. The public methods of this class include:

- ```
  PPMDecompressionSupport (
    const char* inputFileName,
    const char* outputFileName
  )
  ```
  – constructor accepting path to input/output file (either of the two parameters can be null if the file is not needed);

- `void initDecompression ()`
  – method initializing the decoding process (all additional info from the input file, e.g. stored orders of tries, have to be read before calling this method, otherwise it is wrongly read by the range decoder);

- ```
  bool decodeNextSymbol (
    PPMTrie* trie,
    SYMBOL_ID& decodedSymbol,
    const vector<SYMBOL_ID>* preexcludedSymbols = NULL
  )
  ```
  – method which calls `PPMDecoderWrapper::decodeNextSymbol` with the given parameters and returns its return value; moreover, it stores the decoded symbol into a debug array (the program is terminated if there is any mismatch between the debug array for decoding and the debug array for encoding);

- `void writeString (const string& str)`
  – writes the specified string to the output (useful when decoding tokens).

3.6.2 Compression units

3.6.2.1 BasicCompressionUnit

This is the simplest compression unit, wrapping a single trie. It includes the following public methods:

- `BasicCompressionUnit (PPMTrie* trie)`
  – constructor accepting the trie which the unit should work with;

- ```
  void encodeSymbol (
    SYMBOL_ID symbol,
    PPMCompressionSupport* cs,
    const vector<SYMBOL_ID>* preexcludedSymbols = NULL
  )
  ```
  – calls `PPMCompressionSupport::encodeSymbol` with the given parameters;

- ```
  bool decodeSymbol (
    SYMBOL_ID& decodedSymbol,
    PPMDecompressionSupport* ds,
    const vector<SYMBOL_ID>* preexcludedSymbols = NULL
  )
  ```
  – calls `PPMDecompressionSupport::decodeSymbol` with the given parameter and returns its return value;

- `void incrementAlphabetSize ()`
  – increments size of the alphabet used by the wrapped trie;

- ```
  void encodeEOF (
    PPMCompressionSupport* cs,
    const vector<SYMBOL_ID>* preexcludedSymbols = NULL
  )
  ```
  – calls `PPMCompressionSupport::encodeEOF` with the given parameters.

3.6.2.2 StringCompressionUnit

This compression unit is used to encode/decode strings. A string is encoded/decoded byte by byte using an instance of `BasicCompressionUnit`, the end of the string is marked by a special terminating symbol. The public methods of this class include:

- `StringCompressionUnit (PPMTrie* charTrie)`
  – constructor accepting the trie which should be used to encode/decode the strings;

- ```
  void encodeToken (
    const string& token,
    PPMCompressionSupport* cs
  )
  ```
  – encodes the given string;

- `bool decodeToken (`
  `string& decodedToken,`
  `PPMDecompressionSupport* ds`
  `)`
  – decodes the encoded string; if the string isn't properly decoded (including the terminating symbol), returns false.

3.6.2.3 BlackTokenCompressionUnit

This compression unit is used to encode/decode black tokens. It holds an instance of `StringCompressionUnit` (string compressor) to encode/decode not-yet-known black tokens (i.e. the black tokens with no ID assigned) and an instance of `BasicCompressionUnit` (token compressor) to encode/decode the IDs of known black tokens; ID equal to 0 is reserved to mark that the token should be encoded/decoded by the string compression unit. Classes `MappingToID<T>` and `MappingFromID<T>` are used to map the tokens to IDs and vice versa. The class includes the following public methods:

- `BlackTokenCompressionUnit (`
  `PPMTrie* tokenTrie,`
  `PPMTrie* charTrie`
  `)`
  – constructor accepting tries which should be used by the token compressor and string compressor, respectively;

- `void encodeToken (`
  `const string& token,`
  `PPMCompressionSupport* cs`
  `)`
  – encodes the given token;

- `bool decodeToken (`
  `string& decodedToken,`
  `PPMDecompressionSupport* ds`
  `)`
  – decodes the encoded token; returns false if EOF has been decoded;

- `void encodeEOF (PPMCompressionSupport* cs)`
  – encodes EOF using the token compressor.

3.6.2.4 TokenCompressionUnitWithSharedAlphabet<T>

This class is used for encoding/decoding token IDs using multiple different tries. A trie for encoding/decoding a token ID is selected using a key of any data type `T`; if there is no trie yet for the given key, a new trie is automatically

created. The mapping of keys to tries is done using an instance of class `CustomContext`.

Similarly as `BlackTokenCompressionUnit`, this class uses `MappingToID<T>` and `MappingFromID<T>` to map the tokens to IDs and vice versa. While tries for encoding/decoding token IDs are selected by a key, there is only one instance of `StringCompressionUnit` to encode tokens with no ID assigned. All tries holding token IDs share the same alphabet (`SharedSymbolsManager` is used to manage the alphabet), otherwise the mechanism of encoding/decoding tokens is basically the same as for `BlackTokenCompressionUnit`.

The class contains the following public methods:

- ```
  TokenCompressionUnitWithSharedAlphabet (
    PPMTrie* charTrie,
    PPMTrieConfig cfg,
    string (*getName) (T)
  )
  ```
  – constructor accepting a trie which should be used by the string compressor, a trie config which should be used for tries holding token IDs and a pointer to function which generates name of the token tries based on the given key;

- ```
  void encodeToken (
    const string& token,
    PPMCompressionSupport* cs,
    T key
  )
  ```
  – encodes a token (the trie for encoding is selected using the given key);

- ```
  bool decodeToken (
    PPMDecompressionSupport* ds,
    T key,
    string& decodedToken
  )
  ```
  – decodes a token (the trie for decoding is selected using the given key); returns false if EOF has been decoded;

- `void encodeEOF (PPMCompressionSupport* cs, T key)`
  – encodes an EOF (the trie for encoding is selected using the given key).

3.6.2.5 WhiteTokenCompressionUnit

This class is used to encode/decode white tokens. It is a child class of `TokenCompressionUnitWithSharedAlphabet<uint32_t>`, (Unicode symbols are used as keys for selecting token tries). The class contains the following public methods:
  ```

- `WhiteTokenCompressionUnit (`
  `PPMTrie* charTrie,`
  `PPMTrieConfig cfg`
  `)`
  – constructor accepting a pointer to trie which should be used by the string compressor and a config which should be used for tries holding the token IDs;

- `static uint32_t getLastBlackChar (const string& token)`
  – method returning the last Unicode character of the UTF-8-encoded string (used to select a proper key to encode/decode a white token).

3.6.2.6 LemmaIndexCompressionUnit

This class is used for encoding/decoding positions (indexes) of forms in the list of forms generated from a lemma (see Section 2.4.2.1). For each lemma, the class holds an individual model (trie) of indexes (mapping is done using class `CustomContext`). The class contains the following public methods:

- `LemmaIndexCompressionUnit (`
  `const MorphoWrapper* mrph,`
  `uint8_t order`
  `)`
  – constructor accepting pointer to an instance of morphological generator and desired order of the index trie;

- `static bool beforeIndexEncoding (`
  `const string& lemmaId,`
  `const string& origToken,`
  `const string& tag,`
  `const MorphoWrapper* mrph`
  `)`
  – method returning true if it's possible to generate `origToken` from the given `lemmaId` and `tag`;

- `void encodeIndex (`
  `const string& lemmaId,`
  `const string& origToken,`
  `PPMCompressionSupport* cs,`
  `const string& tag`
  `)`
  – method encoding index of `origToken` in the list of forms generated from `lemmaId` + `tag`;

- `string decodeIndex (`
  `const string& lemmaId,`
  `PPMDecompressionSupport ds,`
  `const string& tag`
  `)`
  – method decoding index of original token in the list of forms generated from `lemmaId` + `tag` (the original token is returned).

3.6.2.7 LargeFirstLetterCompressionUnit

This class is an implementation of the letter case heuristics described in Section 2.4.1.2. It holds a basic compression unit for encoding the case and an indicator whether for the next word the case of the first letter should be encoded/decoded (i.e. whether the heuristics is in activated state or not, as shown by Figure 2.1). The class includes the following public methods:

- `LargeFirstLetterCompressionUnit (PPMSupport* support)`
  – constructor using the pointer to `PPMSupport` to create the trie for basic compression unit;

- `bool willEncodeFirstLetterCase (`
  `string& blackToken,`
  `const string& lemma`
  `)`
  – returns true if the unit is prepared to encode the first letter case (then method `encode` must be called before calling this method again), false otherwise; the method inspects the black token; if it is an "activating token" (e.g. "."), the unit just sets the state to *activated* and returns false; else if the state is *activated* and the case of the first character of the black token can be changed (i.e. it is a standard letter of Czech alphabet), the token is possibly changed (the decision whether to change it or not is done using parameter `lemma`, see Section 2.4.1.2) and true is returned; else, false is returned;

- `void encode (PPMCompressionSupport* cs)`
  – sets the state to *deactivated* an encodes the symbol ("lower" or "upper") prepared by `willEncodeFirstLetterCase`;

- `bool willDecodeFirstLetterCase (string& blackToken)`
  – sets the state to *activated* if the black token specified in parameter is an activating token (e.g. "."); returns true if `decodeIfNeeded` should be called before the next calling of this method (in fact it returns whether the state is *activated* or not)

- `bool decodeIfNeeded (`
  `string& tokenToEdit,`
  `PPMDecompressionSupport* ds`
  `)`
  – this method can be called only if the state is *activated* (i.e. the previous calling of `willDecodeFirstLetterCase` returned true); if the token specified in parameter is a token starting with a letter with known lower-case and upper-case variant, then the case is decoded, the token is updated as needed to restore the original case of the first letter and state is set to *deactivated*; method returns false if the decoding fails (that means a serious error which is not expected to happen in our final implementation).

3.6.3 Compression experiments

3.6.3.1 SimplePPMCompressor

This class represents basic byte-oriented compression method where the file is compressed as a sequence of bytes (using a single model M with alphabet A containing one symbol for every possible value of a byte). It is a relic from the original PPM implementation by Jiří Krotil [42] which has been redesigned and kept in the program for comparison purposes. The class contains the following public methods:

- `static void compress (`
  `uint8_t order,`
  `const char* inputFile,`
  `const char* outputFile`
  `)`
  – method which encodes the specified input file; the order of model (trie) M is set to the specified value;

- `static void decompress (`
  `const char* inputFile,`
  `const char* outputFile`
  `)`
  – method which decodes the specified input file; the order of the model is read from the input file.

3.6.3.2 WordBasedCompressorConfig

Class `WordBasedCompressorConfig` is used to hold the orders and also initial limits on sum of symbol frequencies in a given context for some of the tries used in basic word-based compression. Namely, `WordBasedCompressorConfig`

holds configuration info for models of white and black tokens and for models used to encode white and black tokens byte by byte.

3.6.3.3 WordBasedCompressor

This class implements the experiment described in Section 2.8.1. The public methods of this class include:

- `static void compress (`
  `FeedingModule* fm,`
  `const char* outputFile,`
  `const WordBasedCompressorConfig& cfg`
  `)`
  – encodes the input delivered by the feeding module, using the specified config to set the initial configuration of some of the tries; since the described parameters are used the same way in every word-based experiment, we don't comment their purpose in the rest of the design specification;

- `static void decompress (`
  `const char* inputFile,`
  `const char* outputFile,`
  `const WordBasedCompressorConfig& cfg`
  `)`
  – method for decoding.

3.6.3.4 LemmaPlusIndexCompressor

This class implements the experiment described in Section 2.8.2. The public methods of this class include:

- `static void compress (`
  `FeedingModule* fm,`
  `const char* outputFile,`
  `const WordBasedCompressorConfig& cfg,`
  `const MorphoWrapper* mrph,`
  `bool useRawLemmas,`
  `uint8_t order`
  `)`
  – encoding method; `useRawLemmas` specifies whether the stored lemmas should be stripped to raw lemmas (true) or lemma ids (false); `order` specifies the desired order of index tries M_{I_L}; `mrph` is pointer to generator which should be used to generate forms;

- `static void decompress (`
  `const char* inputFile,`
  `const char* outputFile,`
  `const WordBasedCompressorConfig& cfg,`
  `const MorphoWrapper* mrph`
  `)`
  – decoding method; the order of index tries is read from the input file.

3.6.3.5 POSCompressor1

This class implements the experiment described in Section 2.8.3. The public methods of this class include:

- `static void compress (`
  `FeedingModule* fm,`
  `const char* outputFile,`
  `const WordBasedCompressorConfig& cfg,`
  `uint8_t order`
  `)`
  – encoding method; `order` is the desired order of POS trie M_{POS};

- `static void decompress (`
  `const char* inputFile,`
  `const char* outputFile,`
  `const WordBasedCompressorConfig& cfg`
  `)`
  – decoding method; the order of POS trie is read from the input file.

3.6.3.6 POSCompressor2

This class implements the experiment described in Section 2.8.4. The public methods of this class include:

- `static void compress (`
  `FeedingModule* fm,`
  `const char* outputFile,`
  `const WordBasedCompressorConfig& cfg,`
  `const TaggerWrapper* tgr`
  `)`
  – encoding method; `tgr` is pointer to tagger which should be used to perform isolated tagging;

- `static void decompress (`
  `const char* inputFile,`
  `const char* outputFile,`

```
    const WordBasedCompressorConfig& cfg,
    const TaggerWrapper* tgr
)
```
– decoding method.

3.6.3.7 LemmaPlusIndexPlusTagCompressor

This class implements the experiment described in Section 2.8.5. The order of index tries $M_{I_{L,T_{TAG}}}$ is set to an optimal value discovered when experimenting with `LemmaPlusIndexCompressor`. The public methods of this class include:

- ```
  static void compress (
    FeedingModule* fm,
    const char* outputFile,
    const WordBasedCompressorConfig& cfg,
    const MorphoWrapper* mrph,
    const TaggerWrapper* tgr,
    set<uint8_t> selectedPOS,
    uint8_t tagPosition,
    uint8_t order)
  ```
  – encoding method; `mrph` is pointer to generator which should be used to generate forms, `tgr` is pointer to tagger which should be used to perform isolated tagging, `order` is the desired order of tag trie M_{TAG}; meaning of `selectedPOS` and `tagPosition` has already been described;

- ```
  static void decompress (
    const char* inputFile,
    const char* outputFile,
    const WordBasedCompressorConfig& cfg,
    const MorphoWrapper* mrph,
    const TaggerWrapper* tgr,
    set<uint8_t> selectedPOS,
    uint8_t tagPosition
  )
  ```
  – decoding method; the order of tag trie is read from the input file.

Notice that the variables `selectedPOS` and `tagPosition` must be specified for both encoding and decoding, they are not being stored in the compressed file. We decided not to implement storing of these values since it is not needed for demonstration purposes in this thesis.

3.7 Main file

The main file contains procedures responsible for processing command line arguments and launching the experiments accordingly. Apart from the six

compression experiments described in Section 3.6.3, it allows launching five other experiments:

- tokenizing experiment — experiment for verifying that the tokenizing works correctly;
- tagging and lemmatization experiment — experiment showing basic functionality of tagger and tokenizer;
- morphological generation experiment — experiment showing basic functionality of morphological generator;
- POS dependency experiment — the experiment which was used to produce data shown in Table 2.1;
- isolated tagging experiment — the experiment which was used to produce data shown in Table 2.2.

There is a help info which can be displayed by the user to know how to launch each of the experiments.

Chapter 4

Implementation

The whole program has been implemented as proposed in the previous chapters. In this chapter, we present some additional info about the final implementation, not-yet mentioned technical details etc.

4.1 Code properties and compiling

The code of the program complies with standard C++11 of the C++ programming language. The program is intended to be run on GNU/Linux as mentioned before (we may improve the portability in the future, now this thesis is just a proof of concept).

CMake software [46] is used to generate a makefile on the target platform (instructions on how to generate the makefile are attached to the source files). The makefile is generated using file `CMakeLists.txt` created with help of CLion IDE [47]. Minimum version 2.8 of CMake is required. The generated makefile also automatically creates the MorphoDiTa static library from MorphoDiTa source code. The code is compiled using a high level of optimization.

4.2 Handling of errors

When an error is encountered during execution of the program, the program is typically terminated by a fail of assertion. In such case, a brief error message is printed, telling the user which line in the source code contains the failed assertion to allow further inspection of the error.

4.3 Performance

The speed of the compression and decompression proved to be acceptable; however, we used program `gprof` [48] to identify the most time-consuming parts of code anyway. The PPM algorithm (mainly the exclusion mechanism)

and MorphoDiTa tools are the most time-consuming parts of our program; using the data generated by the profiler, we managed to speed up the PPM algorithm significantly but it is still relatively slow (which is not a problem in this thesis, though).

4.4 Running the program

When the program is compiled, a binary called `NLC` is created. When running the binary without any parameter, a help info is shown telling the user how to set the necessary command line parameters. For all compression experiments, path to the original file, compressed file and decompressed file is required. For experiments using MorphoDiTa, path to a tagger model is required.

Chapter 5

Testing and evaluation

This chapter summarizes the testing and evaluation of all compression algorithms defined in Section 2.8. We compare our compression algorithms not only with each other, but also with simple byte-oriented PPM compression implemented by `SimplePPMCompressor` and with common compression programs `gzip`, `bzip2` and `lzma` pre-installed in Ubuntu 15.10, a distribution of GNU/Linux.

For some of the implemented experiments, there is a handicap of storing one extra byte into the compressed file to specify order of some of the tries used by the coder (see Section 3.6.3); we consider this handicap negligible.

In Table 1.1 we summarized which tagger models we can use during the compression. We also stated that we will use only taggers 131112-best_accuracy, 131112-fast or 160310-main. To simplify the names of the taggers, we use name "BEST" for tagger 131112-best_accuracy, name "FAST" for tagger 131112-fast and name "NEW" for tagger 160310-main; we also renamed the taggers on the attached CD so that it's easier to use a specific tagger in our program.

When presenting the trie statistics (see Section 2.7) in this chapter, the legend is as follows:

- # — number of encoded symbols;
- H — bit entropy of the encoded message;
- *bps* — bits per encoded symbol;
- % — how many percents of the estimated file size have been produced by the specific trie.

All testing is done on our set of test files, which is specified in Section 2.3. The sizes of the test files are summarized once again in Table 5.1.

Table 5.1: Summary of test files

file name	size (bytes)
genesis.txt	124 194
komunikace.txt	49 027
mloci.txt	456 808
zakonik.txt	1 290 540

Table 5.2: Compression results of algorithms gzip, bzip2 and lzma (the cells contain size of the compressed file in bytes + compression ratio).

	gzip	bzip2	lzma
genesis.txt	44 196 (35.57 %)	37 301 (30.03 %)	40 367 (32.50 %)
komunikace.txt	19 226 (39.22 %)	16 696 (34.05 %)	18 141 (37.00 %)
mloci.txt	187 163 (40.97 %)	147 355 (32.26 %)	161 253 (35.30 %)
zakonik.txt	346 143 (26.82 %)	252 793 (19.59 %)	270 754 (20.98 %)

5.1 Algorithms gzip, bzip2 and lzma

First we compressed our test files using the common compression programs `gzip`, `bzip2` and `lzma`. The compression ratios achieved by these programs are shown in Table 5.2.

5.2 Byte-oriented PPM compression

This algorithm, represented by class `SimplePPMCompressor`, can be run using the following command (the command shows how to run it on file `mloci.txt` using a trie of order 2):

```
./NLC -e 0 -o 2 -c compressed
                  -d decompressed.txt -i testfiles/mloci.txt
```

The achieved compression ratio largely depends on the selected order. We just show the compression ratios for best order setting for each of the test files:

- `genesis.txt`: best results for order 5 (36 903 bytes, 29.71 %);
- `komunikace.txt`: best results for order 4 (16 196 bytes, 33.03 %);
- `mloci.txt`: best results for order 5 (144 946 bytes, 31.73 %);
- `zakonik.txt`: best results for order 7 (256 884 bytes, 19.91 %).

We can see that the achieved results are better than for any of the three algorithms from Table 5.2 (except bzip2 on `zakonik.txt`). However, we want to achieve even better results using word-based compression.

5.3 Basic word-based compression

This algorithm, represented by class `WordBasedCompressor`, can be run using the following command:

```
./NLC -e 1 -c compressed -d decompressed.txt
      -t morphodita_models/FAST -i testfiles/mloci.txt
          | tee output.txt
```

We strongly recommend to store the output to some file as shown, so that it can be read and analyzed easily (this applies to all word-based compression algorithms in this chapter).

5.3.1 Effect of different tagger models

We ran the algorithm on test file `mloci.txt` to show how the selection of a specific tagger model influences the compression ratio. We achieved the following results for the three taggers:

- tagger FAST: 143 143 bytes (31.34 %);
- tagger BEST: 143 137 bytes (31.33 %);
- tagger NEW: 143 141 bytes (31.34 %).

It's obvious that the selection of tagger model has just a negligible influence on the compression ratio in case of our basic word-based compression algorithm, thus we decided to keep using model FAST by default.

5.3.2 Results for different test files

Using tagger model FAST, we ran the algorithm on each of our test files. The achieved compression ratios are summarized in Table 5.3. We added a column containing number of distinct black tokens contained in each of the files, so that we can see how the size of vocabulary influences the compression ratio.

Using the trie statistics, we also prepared Table 5.4 to show how each of the tries performed (results of models of white tokens M_{WT_c} have been merged for each c).

5.3.3 Summary

For all test files, the achieved compression ratio is better than compression ratios achieved with other compression algorithms tested earlier in this chapter. We observe the following:

- Most of the encoded data is output by models M_{BC} and M_{BT}.

Table 5.3: Compression results of basic word-based compression (the cells in the second column contain size of the compressed file in bytes + compression ratio).

	compression results	number of distinct black tokens
genesis.txt	36 175 (29.13 %)	4260
komunikace.txt	15 899 (32.43 %)	2652
mloci.txt	143 143 (31.34 %)	17720
zakonik.txt	246 496 (19.10 %)	15328

Table 5.4: Basic word-based compression — trie statistics.

	model (trie)	#	H	*bps*	%
genesis.txt	M_{WC}	22	93	4.235	0.03
	M_{BC}	34091	110161	3.231	38.07
	M_{BT}	24561	160991	6.555	55.63
	M_{LC}	1151	449	0.390	0.16
	M_{WT_c}	24561	17699	0.721	6.12
komunikace.txt	M_{WC}	18	81	4.506	0.06
	M_{BC}	24355	74399	3.055	58.49
	M_{BT}	7737	46554	6.017	36.60
	M_{LC}	371	200	0.539	0.16
	M_{WT_c}	7737	5957	0.770	4.48
mloci.txt	M_{WC}	29	80	2.767	0.01
	M_{BC}	161875	495281	3.060	43.25
	M_{BT}	82224	590297	7.179	51.55
	M_{LC}	5256	1116	0.212	0.10
	M_{WT_c}	82224	58367	0.710	5.10
zakonik.txt	M_{WC}	20	87	4.359	0.00
	M_{BC}	140593	353529	2.515	17.93
	M_{BT}	210270	1469693	6.990	74.53
	M_{LC}	9229	4946	0.536	0.25
	M_{WT_c}	210270	143708	0.683	7.29

- It seems that M_{BC} has highest impact on the compression ratio (see column %) in short files (`komunikace.txt`) and in files with rich vocabulary (`mloci.txt`); this is because in both files, relatively a lot of not-yet-known black tokens has to be encoded.
- The impact of M_{WC} on the compression ratio is negligible (there obviously are only a few distinct white tokens in each file, as expected).
- The direct impact of M_{LC} on the compression ratio (see column %) is nearly negligible (but it surely decreases the number of distinct black tokens; we don't show an empirical proof in this thesis, though).
- The impact of M_{WT_c} on the compression ratio (see column %) is moderate (and similar for all test files).

We expect that our other algorithms are able to push the compression ratios even lower.

5.4 Basic experiment using morphological generator

This algorithm, represented by class `LemmaPlusIndexCompressor`, can be run using the following command (`-x` specifies whether the stored lemmas should be stripped to lemma id (0) or raw lemma (1), `-o` specifies order of tries M_{I_L}):

```
./NLC -e 2 -c compressed -d decompressed.txt
      -t morphodita_models/FAST -x 0 -o 0 -i testfiles/mloci.txt
          | tee output.txt
```

We first inspect the influence of parameters `-x` and `-o` on the compression ratio, then we inspect how the compression ratio is influenced by selection of tagger model. Finally, we present detailed results for each of the test files and compare the results with the basic word-based compression.

5.4.1 Influence of parameters -x and -o

First, we inspect the influence of parameter `-x` (with `-o` being set to 0), which is summarized in Table 5.5. We can see that with parameter `-x` set to 1, we achieved slightly better compression ratios, and the number of distinct lemmas recognized by our algorithm decreased slightly (this is what we supposed in our analysis). This means that for further testing, we set `-x` to 1.

The inspection of influence of parameter `-o` follows. We already know the compression ratios for `-o` set to 0. Here we show sizes of the compressed files and compression ratios for `-o 1`:

Table 5.5: Compression results of Basic experiment using morphological generator for different setting of parameter `-x`.

	-x 0		-x 1	
	lemmas	file size & ratio	lemmas	file size & ratio
genesis	2666	36 360 (29.28 %)	2636	36 130 (29.09 %)
komunikace	1768	15 887 (32.40 %)	1753	15 718 (32.06 %)
mloci	10797	144 053 (31.53 %)	10669	143 204 (31.35 %)
zakonik	8404	250 602 (19.42 %)	8336	249 929 (19.37 %)

- `genesis.txt`: 36 151 (29.11 %);
- `komunikace.txt`: 15 747 (32.12 %);
- `mloci.txt`: 143 297 (31.37 %);
- `zakonik.txt`: 250 077 (19.38 %).

We can see that increasing the order of tries M_{I_L} doesn't help, thus parameter `-o` is set to 0 in all the following testing of `LemmaPlusIndexCompressor` (and for `LemmaPlusIndexPlusTagCompressor`, we use this order settings by default for models $M_{I_{L,T_{TAG}}}$ as proposed in Section 3.6.3.7).

5.4.2 Influence of different tagger models

We ran the algorithm on test file `mloci.txt` with the three different models. We achieved the following results:

- tagger FAST: 143 204 bytes (31.35 %);
- tagger BEST: 143 174 bytes (31.34 %);
- tagger NEW: 143 206 bytes (31.35 %).

It's obvious that the selection of tagger model has just a negligible influence on the compression ratio as it was in case of the previous algorithm, thus we keep using model FAST by default.

5.4.3 Detailed results for different test files

We use `-x` 1, `-o` 0 and tagger model FAST here, thus we already know the achieved compression ratios (see the right half of Table 5.5).

Using the trie statistics, we also prepared Table 5.6 to show how each of the tries performs on each of the files (results of models of indexes M_{I_L} have been merged for each L). We don't show the statistics for models M_{WC}, M_{WT_c} and M_{LC} — the compression algorithms have been designed in such a way that the statistics of these models are the same in each algorithm (except column %, which depends also on size of the compressed file).

Table 5.6: Basic experiment using morphological generator — trie statistics.

	model (trie)	#	H	*bps*	%
genesis.txt	M_{BC}	20957	71335	3.404	24.68
	M_{BT}	24561	155199	6.319	53.70
	M_{I_L}	16675	44252	2.654	15.31
komunikace.txt	M_{BC}	15647	51240	3.275	40.75
	M_{BT}	7737	50813	6.567	40.41
	M_{I_L}	5449	17442	3.201	13.87
mloci.txt	M_{BC}	94730	312181	3.295	27.25
	M_{BT}	82224	612299	7.447	53.45
	M_{I_L}	54503	161578	2.965	14.10
zakonik.txt	M_{BC}	66537	191410	2.877	9.57
	M_{BT}	210270	1371533	6.523	68.60
	M_{I_L}	150589	287731	1.911	14.39

5.4.4 Summary

Our task in this subsection is to summarize the experiment and compare it with our basic word-based compression tested in Section 5.3.

For two test files, the compression ratio improved slightly. For file `mloci.txt`, the compression ratio worsened negligibly. For file `zakonik.txt`, which contains the most unnatural text of our test set, the disimprovement is more significant. Looking at the trie statistics, we can state the following:

- Index tries M_{I_L} produce approximately 14–15% of the total output. They are most efficient (see column *bps*) on the longest text with relatively small vocabulary (`zakonik.txt`) and least efficient on short texts or texts with an extensive vocabulary (this is probably because there are frequent encodings in PPM order -1).

- The main improvement can be observed for models M_{BC}, where the number of encodings decreased significantly for all files (this is exactly what we supposed in the analysis).

- For models M_{BT}, the results are ambiguous — in two cases (files `komunikace.txt` and `mloci.txt`), the efficiency (see column *bps*) decreased slightly, for the two other files we observe a slight improvement.

- In case of file `zakonik.txt`, the amount of data encoded by index tries M_{I_L} is so big that it overbalanced all improvements of M_{BC} and M_{BT} (even though M_{I_L} has a very good value of *bps* here).

As a conclusion, we can state that this algorithm is at least fully competitive with the basic word-based compression algorithm, if not better. We

Table 5.7: Compression results of Experiment with part-of-speech tags (1) for different setting of parameter `-o`.

	-o 0	-o 1	-o 2	-o 3
genesis.txt	32.30 %	31.45 %	31.26 %	31.35 %
komunikace.txt	34.66 %	34.10 %	34.18 %	34.37 %
mloci.txt	33.92 %	33.23 %	33.16 %	33.25 %
zakonik.txt	22.79 %	22.00 %	21.60 %	21.46 %

expect that algorithm implemented as `LemmaPlusIndexPlusTagCompressor`, which is a direct extension of the algorithm tested in this section, is able to achieve even better compression ratios.

5.5 Experiment with part-of-speech tags (1)

This algorithm, represented by class `POSCompressor1`, can be run using the following command (`-o` specifies order of trie M_{POS}):

```
./NLC -e 3 -c compressed -d decompressed.txt
      -t morphodita_models/FAST -o 1 -i testfiles/mloci.txt
          | tee output.txt
```

We first inspect the influence of parameter `-o` and influence of individual tagger models on the compression ratio, then we present trie statistics for all test files in similar way as for the previous algorithms.

5.5.1 Influence of parameter -o

We have run the algorithm on all test files with different setting of `-o`, the results (compression ratios) are summarized in Table 5.7. The table shows that for compressing POS tags, a context-based compression method is better than a statistical one (i.e. zero-order PPM), as expected. It seems that, on average, best compression ratios are achieved when parameter `-o` is set to 2, we thus use this setting by default in the following experiments with `POSCompressor1`.

5.5.2 Influence of different tagger models

We ran the algorithm on test file `mloci.txt` with the three different models. We achieved the following results:

- tagger FAST: 151 485 bytes (33.16 %);
- tagger BEST: 151 444 bytes (33.15 %);

Table 5.8: Experiment with part-of-speech tags (1) — trie statistics.

	model (trie)	#	H	*bps*	%
genesis.txt	M_{BC}	34532	111174	3.219	35.80
	M_{BT_X}	24561	120799	4.918	38.89
	M_{POS}	24561	60365	2.458	19.44
komunikace.txt	M_{BC}	24454	74667	3.053	55.70
	M_{BT_X}	7737	31867	4.119	23.77
	M_{POS}	7737	21275	2.750	15.87
mloci.txt	M_{BC}	163291	499249	3.057	41.20
	M_{BT_X}	82224	437004	5.315	36.06
	M_{POS}	82224	216053	2.628	17.83
zakonik.txt	M_{BC}	141337	355191	2.513	15.93
	M_{BT_X}	210270	1229224	5.846	55.13
	M_{POS}	210270	496680	2.362	22.27

- tagger NEW: 151 463 bytes (33.16 %).

Again, it's obvious that the selection of tagger model doesn't significantly affect the compression ratio.

5.5.3 Detailed results for different test files

We use `-o` 2 and tagger model FAST here, thus we already know the achieved compression ratios (see the corresponding column in Table 5.7). Detailed info about performance of each of the tries is contained in Table 5.8 (results of models of black tokens M_{BT_X} have been merged for each X). We don't show statistics for models M_{WC}, M_{WT_c} and M_{LC} since they perform here exactly the same as in the previous two algorithms.

5.5.4 Summary

We can state that the compression ratio achieved by this algorithm is definitely worse than for the basic word-based compression. Looking at the trie statistics, we can state the following:

- For models M_{BC}, the amount of encoded information increased slightly for all test files. This is probably because some of the tokens are incorporated in more than one of the models M_{BT_X} and each of the models M_{BT_X} has an individual alphabet of tokens, thus the same token has to be encoded repeatedly by M_{BC}.
- For models of black tokens M_{BT_X}, there is a significant decrease in amount of encoded information for all test files. This is what we expected in the analysis.

Table 5.9: Compression results of Experiment with part-of-speech tags (2).

	file size (compression ratio)
genesis.txt	37 594 (30.27 %)
komunikace.txt	16 098 (32.83 %)
mloci.txt	146 731 (32.12 %)
zakonik.txt	274 011 (21.23 %)

- Storing the value of POS tag takes approximately 2.5 bits per encoded black token; this is quite a big amount of information which overbalances the positive effect of this approach on models M_{BT_X}, making the compression ratio worse than if the basic word-based compression was used.

5.6 Experiment with part-of-speech tags (2)

This algorithm, represented by class `POSCompressor2`, can be run using the following command:

```
./NLC -e 4 -c compressed -d decompressed.txt
      -t morphodita_models/FAST -i testfiles/mloci.txt
          | tee output.txt
```

We first briefly show how the compression ratio is affected by the selection of tagger model, then we present detailed results for all test files.

5.6.1 Influence of different tagger models

We ran the algorithm on test file `mloci.txt` with the three different models. We achieved the following results:

- tagger FAST: 146 731 bytes (32.12 %);
- tagger BEST: 146 688 bytes (32.11 %);
- tagger NEW: 146 725 bytes (32.12 %).

We can see that there is no big difference between the three models, so we keep using model FAST.

5.6.2 Detailed results for different test files

Table 5.9 summarizes achieved compression ratios and Table 5.10 shows the trie statistics (results of models M_{BT_X} have been merged for each X); we don't include models M_{WC}, M_{WT_c}, M_{LC} and M_{BC}, since they perform here exactly the same as for the basic word-based compression.

Table 5.10: Experiment with part-of-speech tags (2) — trie statistics.

	model (trie)	#	H	*bps*	%
genesis.txt	M_{BT_X}	24561	172346	7.017	57.31
komunikace.txt	M_{BT_X}	7737	48144	6.223	37.38
mloci.txt	M_{BT_X}	82224	618996	7.528	52.73
zakonik.txt	M_{BT_X}	1689813	496680	8.036	77.09

5.6.3 Summary

The achieved compression ratio is worse than compression ratio achieved by the basic word-based compression for all test files, this is caused by more bits spent to encode a black token on average than for the basic word-based compression. Thus we can state that for Czech text, this way of using POS tags is probably not beneficial.

5.7 Experiment with non-part-of-speech tags

This algorithm, represented by class `LemmaPlusIndexPlusTagCompressor`, can be run using the following command (`-o` specifies order of the tag trie M_{TAG}, `-x` specifies the list of selected parts of speech *selectedPOS*, `-y` specifies the selected position in the tag *tagPosition*; notice that parameter `-y` uses zero-based indexing):

```
./NLC -e 5 -c compressed -d decompressed.txt
      -t morphodita_models/FAST -i testfiles/mloci.txt
          -o 0 -x AV -y 10 | tee output.txt
```

We remind that this algorithm is a direct extension of the algorithm tested in Section 5.4. For index tries $M_{I_{L,T_{TAG}}}$, the order is always set to 0, which proved to be an optimal setting.

We want to test the influence of parameters `-x`, `-y` and `-o`, as well as the influence of selection of tagger model, on the compression ratio. However, this experiment is far more extensive than the previous ones, so we must choose wisely what is worth testing (the number of possible configurations of this algorithm is really huge). First, we test the influence of different tagger models.

5.7.1 Influence of different tagger models

We ran the algorithm on test file `mloci.txt` with the three different models for two selected configurations. We achieved the following results:

- configuration `-x N -y 4 -o 0`

  - tagger FAST: 145 811 bytes (31.92 %);
  - tagger BEST: 145 801 bytes (31.92 %);
  - tagger NEW: 145 838 bytes (31.93 %).

- configuration `-x A -y 10 -o 0`
  - tagger FAST: 142 922 bytes (31.29 %);
  - tagger BEST: 142 888 bytes (31.28 %);
  - tagger NEW: 142 922 bytes (31.29 %).

The results show that the selection of a specific tagger model makes just a negligible difference, thus it's acceptable to keep using model FAST.

5.7.2 Selecting useful configurations

In this subsection, we summarize which combinations of values of parameters `-x`, `-y` and `-o` should be tested in detail.

Our main goal is to decrease the amount of data that needs to be encoded by index tries, this should be achieved by storing a specific grammatical category for some selected part(s) of speech. It's obviously not worth storing *gender* for prepositions or *number* for adverbs; here we present a summary of configurations we want to test (for each tested `-y`, there is a list of values of `-x`):

- `-y 2` (gender): N (nouns), A (adjectives), P (pronouns), C (numerals), V (verbs);
- `-y 3` (number): N, A, P, V;
- `-y 4` (case): N, A, P, C;
- `-y 5` (possessor's gender): A, P;
- `-y 6` (possessor's number): A, P;
- `-y 7` (person): P, V, J (conjunctions);
- `-y 8` (tense): V;
- `-y 9` (grade): A, D (adverbs);
- `-y 10` (negation): N, A, V, D;
- `-y 11` (voice): V.

Table 5.11: Experiment with non-part-of-speech tags — trie statistics for file `mloci.txt`.

configuration	model (trie)	#	H	*bps*	%
`-x N -y 4 -o 0`	M_{TAG}	16163	42156	2.608	3.61
	$M_{I_{L,T_{TAG}}}$	53846	140275	2.605	12.02
`-x A -y 10 -o 0`	M_{TAG}	6328	1350	0.213	0.12
	$M_{I_{L,T_{TAG}}}$	54503	157970	2.898	13.82

It's possible to set `-x` and `-y` to any applicable value, but we don't expect that configurations not contained in the presented list are useful. We test all the values of `-x` separately first, then we combine all the values of `-x` for each `-y` (i.e. configuration `-y 3 -x NAPV`, for example).

Each of the configurations has been tested with `-o 0` and then with `-o 1` to see if the context model helps when storing the tag values or not. The testing has been performed only on test file `mloci.txt`, which should be representative enough anyway since this file is long and it contains a more natural text than most of the other test files.

The results are shown in Tables B.1 and B.2. We can state the following:

- In most cases, the compression ratio wasn't improved.
- Storing the tag for multiple parts of speech is often not beneficial.
- For some positions in the tag, a context model is better than a statistical model (zero-order trie).

Before we go further, we show detailed trie statistics for two configurations, `-x N -y 4 -o 0` (one of the worst configurations) and `-x A -y 10 -o 0` (one of the best configurations), in Table 5.11. The results of models M_{WC}, M_{WT_c}, M_{LC} are the same as in algorithm tested in Section 5.4; this is true also for models M_{BC} and M_{BT}, which is not guaranteed by the algorithm though. The results of index tries $M_{I_{L,T_{TAG}}}$ have been merged for all pairs $\{L, T_{TAG}\}$.

We can state that the amount of data encoded by index tries decreased in both cases, as expected. It's the tag trie M_{TAG} what makes the biggest difference — for the first case, the trie encodes 2.6 bits per symbol, while in the second case it's only 0.2 bits per symbol. It seems that storing some positions in the tag is not useful, mostly because of too many possible values with similar probability (compare 9 possible values on position 4 with just 3 possible values on position 10).

For further testing on other files from our test set, we reduce the set of configurations. The final set is specified in the following subsection.

5.7.3 Main testing with the selected configurations

From our previous set of configurations, we keep only those configurations which have positive or neutral effect on the compression ratio in case of file `mloci.txt` (when the results are similar for both settings of `-o`, we prefer setting `-o` to 1; we also exclude the configurations where multiple parts of speech are used). We check here whether these configurations are helpful in general or not, testing the configurations on all files from our test set. The selected configurations and compression results are summarized in Table B.3.

5.7.4 Summary

Looking at the results, we can state the following:

- Storing the value of the 10th position in the tag (negation) seems generally beneficial (except for adverbs).
- Storing the value of the 9th position (grade) seems generally beneficial as well (if we consider `zakonik.txt` an untypical text, which it really is).
- Improvement can be also achieved by storing the 7th position (person) for pronouns.
- Other configurations don't improve the compression ratio or they have ambiguous results.

We have shown that the algorithm is really able to improve the compression ratio as expected (when compared with algorithm tested in Section 5.4), but the improvement happens only for specific configurations of the algorithm. It's also necessary to mention that the improvements are relatively small and insignificant. We remind that there is a big number of configurations not tested here, this summary is thus just a proof of concept.

The algorithm could be improved by storing several positions of the tag alongside, so that the number of forms generated from the lemma gets even smaller; this may be implemented and tested in some future work. We also saw that for some selections of `-y`, the compression results worsen when we use multiple parts of speech; this means that it could be useful to split tag models M_{TAG} to multiple tag models $M_{TAG_{POS}}$ for each value POS of part-of-speech tag.

5.8 Summary of experiments

In this section, we summarize the whole testing once more. We already know how each of the implemented algorithms performs and we can also compare

Table 5.12: Summary of compression ratios (in percents) for all tested algorithms and programs for each of the test files (see legend below). Best results are marked in bold.

	gz	bz2	lzma	E_0	E_1	E_2	E_3	E_4	E_5
f_1	35.57	30.03	32.50	29.71	29.13	29.09	31.26	30.27	**29.04**
f_2	39.22	34.05	37.00	33.03	32.43	32.06	34.18	32.83	**31.97**
f_3	40.97	32.26	35.30	31.73	31.34	31.35	33.16	32.12	**31.29**
f_4	26.82	19.59	20.98	19.91	**19.10**	19.37	21.60	21.23	19.35

Table legend:

f_1 = genesis.txt
f_2 = komunikace.txt
f_3 = mloci.txt
f_4 = zakonik.txt

gz = gzip
bz2 = bzip2
E_0 = byte-oriented PPM (mixed configurations)
E_1 = basic word-based compression
E_2 = basic experiment using morphological generator
E_3 = experiment with part-of-speech tags (1) with -o 2
E_4 = experiment with part-of-speech tags (2)
E_5 = exp. with non-part-of-speech tags (final testing only, mixed configurations)

them to common compression programs which have not been implemented in this thesis. Table 5.12 shows the best compression ratios achieved by each of the tested algorithms and programs for each of the test files (using tagger model FAST in the word-based algorithms). Looking at the table, we can state the following:

- For all test files, some of our word-based compression algorithms achieved better compression ratios than any non-word-based algorithm or program.

- For all test files except one (`zakonik.txt`), we were able to improve the compression ratio using lemmatisation, tagging and morphological generation. It is necessary to mention that the text in file `zakonik.txt` is largely structured and less natural than the texts contained in other test files, thus we can state that using the linguistic tools really improves the compression ratio of natural texts in general.

- The improvements have been achieved using the idea of storing lemmas (or lemmas + tags) and generating the word forms using the stored lem-

mas (or lemmas + tags); we didn't manage to improve the compression ratio of the basic word-based compression using other approaches.

- None of the improvements achieved by our algorithms can be considered significant; we just proved the concept of utilizing the linguistic tools to improve the compression ratio.
- Although we proposed that we don't care about the speed of the algorithms, we admit that our word-based algorithms (especially when MorphoDiTa is used frequently) are significantly (but acceptably) slower than the tested non-word-based algorithms and programs.

Conclusion

We have designed and implemented several algorithms for lossless compression of natural Czech text in the C++ programming language. All the algorithms are based on adaptive PPM compression method. First we created a basic word-based compression algorithm (a reference algorithm) and then we tried to improve it using lemmatiser, morphological tagger and morphological generator from open-source software MorphoDiTa (four improving algorithms have been designed and implemented). MorphoDiTa has been chosen as a suitable tool based on previous survey.

Each of the four algorithms is typically based on different idea, this way we wanted to verify which ideas are useful and which are not. First of the algorithms uses lemmatiser to get lemma of each token in the text; the lemmas are stored instead of the real tokens and morphological generator is used to generate the original token. The second algorithm stores part-of-speech tags delivered by morphological tagger to take advantage of the structure of a natural text; the probability of a certain word is estimated by the stored part-of-speech tag. The third algorithm uses part-of-speech tags in a different way, without storing them — the probability of a certain word is estimated using part-of-speech tag of the previous word. The fourth algorithm is an extension of the first algorithm, where a part of morphological tag is stored alongside the lemma (the original token is acquired using the morphological generator, again).

The testing on our set of test files has shown that the second and third algorithm achieve worse compression ratios than the basic word-based reference algorithm, while the first algorithm is at least fully competitive with the reference algorithm and the fourth algorithm slightly but generally surpasses the reference algorithm for some configurations. Although the improvements are not significant, we proved this way that the linguistic tools can be successfully used to improve the compression ratio when compressing a natural Czech text.

We also created an important by-product in this thesis — a highly univer-

sal implementation of PPM compression algorithm with detailed compression statistics, which can be incorporated into ExCom [43] compression library maintained by the Prague Stringology Club and/or used in any future work. The implementation of PPM is based on an implementation by Jiří Krotil, which has been modified and improved to match our needs in this thesis.

At the very end of this thesis, we suggest possible future work:

- All of our compression algorithms use a purely adaptive compression, building the compression model from scratch. We suppose that our algorithms would achieve better compression ratios with some reasonable initial model. The future work may thus include the task of creating such model.
- We didn't run our algorithms on any text written in some other language than Czech and we don't guarantee that the algorithms work flawlessly with other languages; however, it could be interesting to test some of the approaches and ideas on English texts (MorphoDiTa works also with English models, as mentioned).
- The fourth one of the four algorithms mentioned in this section could be improved to store multiple tag positions at the same time, as proposed earlier in this thesis (we focused only on one-dimensional experiments, storing just single tag positions).
- There are many features of Czech language which have not been utilized in the algorithms (we didn't utilize the fact that a specific preposition strongly determines the case of the following noun, adjective, pronoun or number, for example); new algorithms utilizing these features can be designed and evaluated.
- A possible goal of the future work is to search for more useful ideas and then to create a complex algorithm utilizing all those ideas.
- When this thesis was already in advance, we discovered a new open-source linguistic tool — dependency parser Parsito [49]. When some models for Czech language are issued, this tool might be used to inspect the structure of individual sentences during the compression, which could result in new compression algorithms.
- The speed of the implemented PPM algorithm could be improved. This may be a complex task though, since there is a trade-off between algorithm speed and memory efficiency, including the time needed to allocate the memory.
- Another PPM-related task is to reimplement the memory manager used in PPM, which is currently not able to reallocate a too big model if multiple models are used.

Bibliography

[1] MorphoDiTa User's Manual | ÚFAL. 2016, Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://ufal.mff.cuni.cz/morphodita/users-manual

[2] Services. 2015, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: https://lindat.mff.cuni.cz/en/services/

[3] Project pages | ÚFAL. 2015, Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: https://ufal.mff.cuni.cz/projects

[4] Sikora, R. *Vyhledávání v českých dokumentech pomocí Apache Solr [online]*. Master's thesis, Masarykova univerzita, Fakulta informatiky, Brno, 2012 [cit. 2016-05-30]. Available from: http://is.muni.cz/th/256499/fi_m/

[5] MorphoDiTa | ÚFAL. 2016, Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://ufal.mff.cuni.cz/morphodita

[6] Straková, J.; Straka, M.; Hajič, J. Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, Johns Hopkins University, Baltimore, MD, USA, Stroudsburg, PA, USA: Association for Computational Linguistics, 2014, ISBN 978-1-941643-00-6, pp. 13–18. Available from: http://www.aclweb.org/anthology/P14-5003

[7] MorphoDiTa. 2016, Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://lindat.mff.cuni.cz/services/morphodita/

[8] Hajič, J. Czech Morphological Analyzer v1. 2014, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: https://lindat.mff.cuni.cz/services/morph/index.html

[9] Flect | ÚFAL. 2015, Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: https://ufal.mff.cuni.cz/flect

[10] Morče - Czech morphological tagger: Introduction. Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://ufal.mff.cuni.cz/morce/index.php

[11] Morče - Czech morphological tagger: Download. Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://ufal.mff.cuni.cz/morce/download.php

[12] Spousta, M. Featurama download | SourceForge.net. Available from: https://sourceforge.net/projects/featurama/

[13] LemmaGen. 2010, Jozef Stefan Institute. Available from: http://lemmatise.ijs.si/

[14] LemmaGen - Online Services. 2010, Jozef Stefan Institute. Available from: http://lemmatise.ijs.si/Services

[15] Lingware. 2001-2016, Natural Language Processing Centre, Faculty of Informatics Masaryk University. Available from: https://nlp.fi.muni.cz/cs/Lingware

[16] Šmerk, P. ajka vs. majka. Available from: https://nlp.fi.muni.cz/czech-morphology-analyser/majka.html

[17] Free natural language morphology for Czech, Slovak, Polish, Swedish, German, French, Italian, English, Portuguese, Catalan, Welsh, Spanish, Galician, Asturian and Russian. Natural Language Processing Centre, Faculty of Informatics Masaryk University. Available from: https://nlp.fi.muni.cz/czech-morphology-analyser/

[18] Jakubíček, M.; Kovář, V.; Šmerk, P. Czech Morphological Tagset Revisited. Natural Language Processing Centre, Faculty of Informatics Masaryk University. Available from: http://raslan2011.nlp-consulting.net/program/paper05.pdf

[19] 2.2.1. Positional tags. Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://ufal.mff.cuni.cz/pdt2.0/doc/manuals/en/m-layer/html/ch02s02s01.html

[20] 1. Embedding Python in Another Application - Python 3.5.1 documentation. 2001-2016, Python Software Foundation. Available from: https://docs.python.org/3/extending/embedding.html

[21] Release MorphoDiTa 1.3.0 · ufal/morphodita · GitHub. 2016, GitHub, Inc. Available from: https://github.com/ufal/morphodita/releases/tag/v1.3.0

[22] MorphoDiTa API Reference | ÚFAL. 2016, Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://ufal.mff.cuni.cz/morphodita/api-reference

[23] Institute of Formal and Applied Linguistics, Charles University in Prague. *MorphoDiTa: Morphological Dictionary and Tagger*. 2014, version 1.3.0.

[24] Straka, M.; Straková, J. Czech Models (MorfFlex CZ 160310 + PDT 3.0) for MorphoDiTa 160310. 2016, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://hdl.handle.net/11234/1-1674

[25] Straka, M.; Straková, J. Czech Models (MorfFlex CZ + PDT) for MorphoDiTa. 2013, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://hdl.handle.net/11858/00-097C-0000-0023-68D8-1

[26] Hajič, J. Positional Tags: Quick Reference (Czech "HM" Morphology). 2000. Available from: http://ufal.mff.cuni.cz/pdt/Morphology_and_Tagging/Doc/hmptagqr.html

[27] Salomon, D. *Data Compression: The Complete Reference*. Springer, 2006, ISBN 1846286026.

[28] Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal*, volume 27, no. 3, 1948: pp. 379–423.

[29] Holub, J. Introduction (lecture notes for Data Compression subject). 2016. Available from: https://edux.fit.cvut.cz/courses/MI-KOD/_media/lectures/01/mi-kod-01-intro.pdf

[30] Welcome to Graphviz. Available from: http://www.graphviz.org/

[31] Lánský, J.; Žemlička, M. Text Compression: Syllables. Charles University, Faculty of Mathematics and Physics. Available from: ceur-ws.org/Vol-129/paper6.pdf

[32] Martínez-Prieto, M. A.; Adiego, J.; de la Fuente, P. Natural Language Compression on Edge-Guided text preprocessing. *Information Sciences*, volume 181, no. 24, 2011: pp. 5387 – 5411, ISSN 0020-0255.

Available from: http://www.sciencedirect.com/science/article/pii/S0020025511003781

[33] Procházka, P. *Word-based Statistical Data Compression Methods*. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2008. Available from: https://dip.felk.cvut.cz/browse/pdfcache/prochp5_2008dipl.pdf

[34] Brisaboa, N. R.; Fariña, A.; Navarro, G.; et al. Efficiently decodable and searchable natural language adaptive compression. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, 2005, pp. 234–241. Available from: http://lbd.udc.es/Repository/Publications/Drafts/Effdecandsea.pdf

[35] Manning, C. D.; Raghavan, P.; Schütze, H.; et al. *Introduction to information retrieval*. Cambridge university press Cambridge, 2009, online edition available from http://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf.

[36] Kazík, O.; Lánský, J. Linguistic Text Compression. In *ICDT 2011: The Sixth International Conference on Digital Telecommunications*, 2011, pp. 64–73. Available from: https://www.thinkmind.org/download.php?articleid=icdt_2011_3_40_20041

[37] Horspool, R. N.; Cormack, G. V. Constructing Word-Based Text Compression Algorithms. In *Data Compression Conference*, 1992, pp. 62–71. Available from: http://webhome.cs.uvic.ca/~nigelh/Publications/wordCompression.pdf

[38] Wikizdroje. Pět knih Mojžíšových/Na počátku. 2015. Available from: https://cs.wikisource.org/w/index.php?title=P%C4%9Bt_knih_Moj%C5%BE%C3%AD%C5%A1ov%C3%BDch/Na_po%C4%8D%C3%A1tku&oldid=110176

[39] Wikizdroje. O digitální komunikaci. 2014. Available from: https://cs.wikisource.org/w/index.php?title=O_digit%C3%A1ln%C3%AD_komunikaci&oldid=76435

[40] Karel Čapek | Městská knihovna v Praze. Municipal Library of Prague. Available from: https://www.mlp.cz/cz/projekty/on-line-projekty/karel-capek/

[41] Wikizdroje. Občanský zákoník (2012). 2016. Available from: https://cs.wikisource.org/w/index.php?title=Ob%C4%8Dansk%C3%BD_z%C3%A1kon%C3%ADk_(2012)&oldid=115913

[42] Krotil, J. *Kompresní metody PPM*. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2012. Available from: https://dip.felk.cvut.cz/browse/pdfcache/krotijir_2012dipl.pdf

[43] The ExCom Library. 2013, The Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague. Available from: http://www.stringology.org/projects/ExCom/

[44] Doxygen: Main Page. 2015, Doxygen. Available from: http://www.stack.nl/~dimitri/doxygen/

[45] Dipperstein, M. Arithmetic Code Discussion and Implementation. 2014. Available from: http://michael.dipperstein.com/arithmetic/

[46] CMake. 2016. Available from: https://cmake.org/

[47] A cross-platform IDE for C and C++ :: JetBrains CLion. 2016, JetBrains. Available from: https://www.jetbrains.com/clion/

[48] GNU gprof. Available from: https://sourceware.org/binutils/docs/gprof/

[49] Straka, M. Parsito. 2015, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague. Available from: http://hdl.handle.net/11234/1-1584

Appendix A

Design — class diagrams

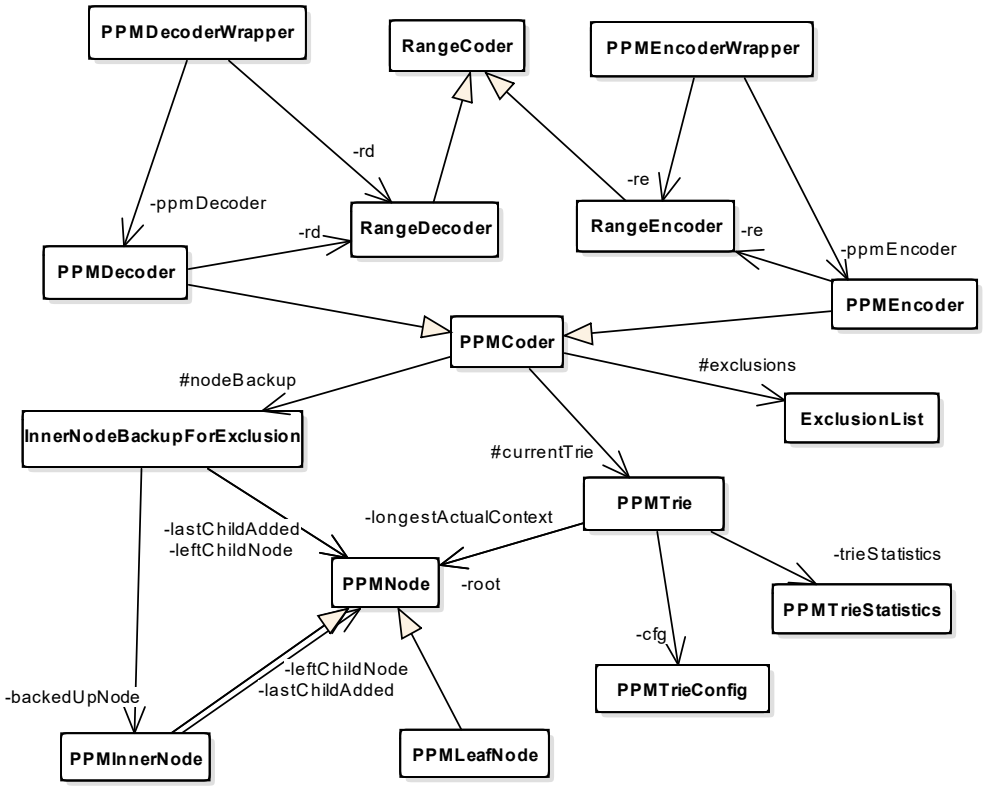


Figure A.1: Class model containing a selection of classes which are related to range coder, PPM coder and PPM structure.

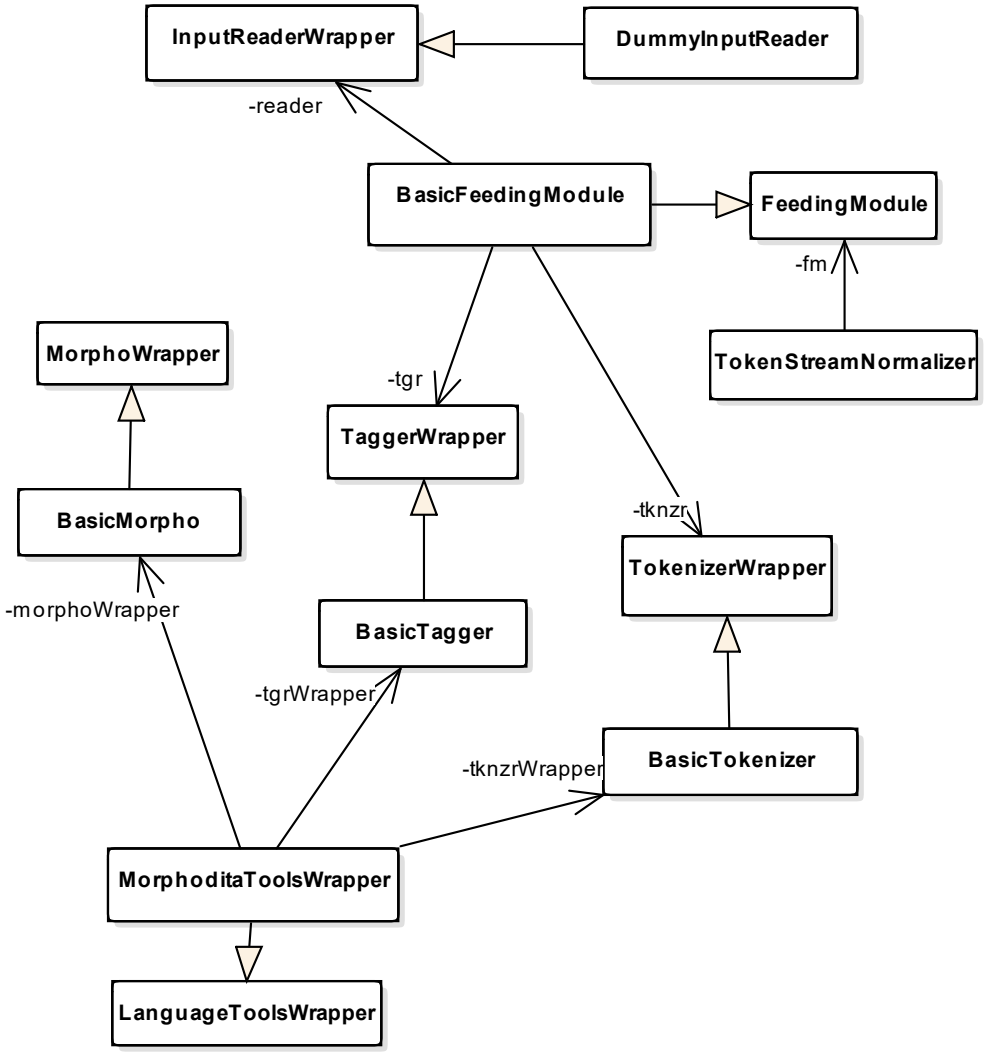


Figure A.2: Class model containing a selection of classes which are related to input processing and linguistic tools.

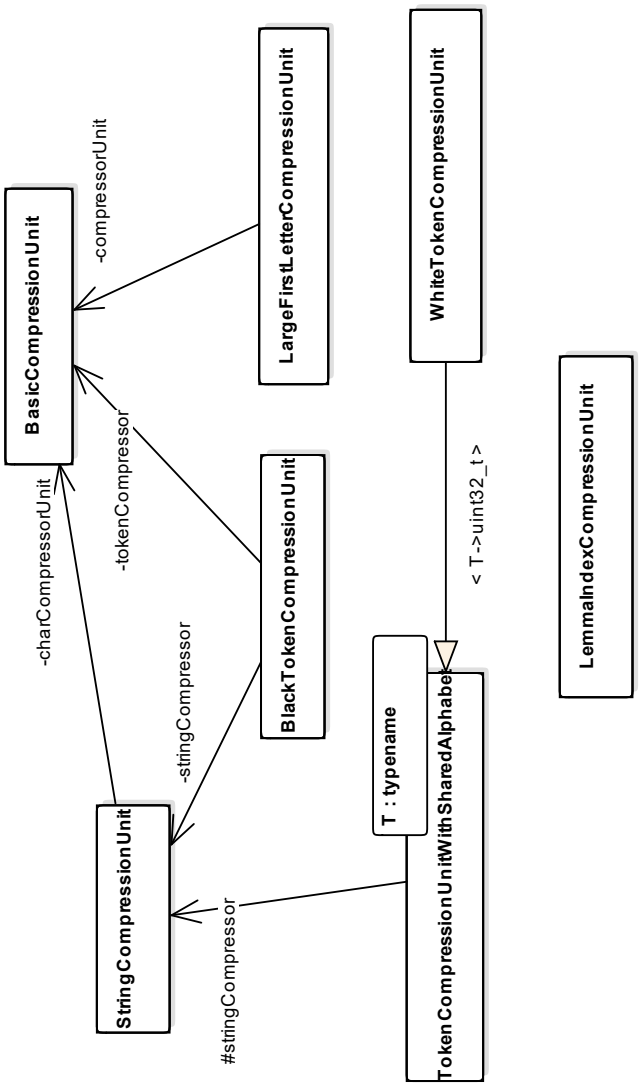


Figure A.3: Class model containing classes which represent the compression units.

Appendix B

Evaluation of algorithms — tables

Table B.1: Experiment with non-part-of-speech tags — initial testing on file `mloci.txt` with `-o 0`. Improvements of algorithm which has been tested in Section 5.4 are marked in bold.

-y	-x	file size (compression ratio)
2 (gender)	N	143 317 (31.37 %)
	A	144 486 (31.63 %)
	P	143 703 (31.46 %)
	C	143 289 (31.37 %)
	V	143 482 (31.41 %)
	NAPCV	146 778 (32.13 %)
3 (number)	N	143 897 (31.50 %)
	A	143 637 (31.44 %)
	P	143 748 (31.47 %)
	V	143 267 (31.36 %)
	NAPV	145 766 (31.91 %)
4 (case)	N	145 811 (31.92 %)
	A	144 221 (31.57 %)
	P	144 200 (31.57 %)
	C	143 349 (31.38 %)
	NAPC	148 176 (32.44 %)
5 (possessor's gender)	A	143 204 (31.35 %)
	P	143 224 (31.35 %)
	AP	143 224 (31.35 %)
6 (possessor's gender)	A	143 204 (31.35 %)
	P	143 234 (31.36 %)
	AP	143 234 (31.36 %)
7 (person)	P	**143 173 (31.34 %)**
	V	143 308 (31.37 %)
	J	143 215 (31.35 %)
	PVJ	143 539 (31.42 %)
8 (tense)	V	143 349 (31.38 %)
9 (grade)	A	**143 104 (31.33 %)**
	D	143 218 (31.35 %)
	AD	**143 129 (31.33 %)**
10 (negation)	N	**143 193 (31.35 %)**
	A	**142 922 (31.29 %)**
	V	**142 957 (31.29 %)**
	D	143 304 (31.37 %)
	NAVD	**142 862 (31.27 %)**
11 (voice)	V	143 288 (31.37 %)

Table B.2: Experiment with non-part-of-speech tags — initial testing on file `mloci.txt` with `-o 1`. Improvements of algorithm which has been tested in Section 5.4 are marked in bold. Improvements against `-o 0` marked by an asterisk.

-y	-x	file size (compression ratio)
2 (gender)	N	143 326 (31.37 %)
	A	144 480 (31.63 %) ∗
	P	143 720 (31.46 %)
	C	143 287 (31.37 %) ∗
	V	143 397 (31.39 %) ∗
	NAPCV	146 704 (32.11 %) ∗
3 (number)	N	143 883 (31.50 %) ∗
	A	143 616 (31.44 %) ∗
	P	143 739 (31.47 %) ∗
	V	143 223 (31.35 %) ∗
	NAPV	145 341 (31.82 %) ∗
4 (case)	N	145 821 (31.92 %)
	A	144 162 (31.56 %) ∗
	P	144 208 (31.57 %)
	C	143 349 (31.38 %)
	NAPC	147 527 (32.30 %) ∗
5 (possessor's gender)	A	143 204 (31.35 %)
	P	143 225 (31.35 %)
	AP	143 224 (31.35 %)
6 (possessor's gender)	A	143 204 (31.35 %)
	P	143 234 (31.36 %)
	AP	143 234 (31.36 %)
7 (person)	P	**143 174 (31.34 %)**
	V	143 322 (31.37 %)
	J	143 217 (31.35 %)
	PVJ	143 545 (31.42 %)
8 (tense)	V	143 333 (31.38 %) ∗
9 (grade)	A	**143 104 (31.33 %)**
	D	143 218 (31.35 %)
	AD	**143 125 (31.33 %)** ∗
10 (negation)	N	**143 195 (31.35 %)**
	A	**142 923 (31.29 %)**
	V	**142 958 (31.29 %)**
	D	143 304 (31.37 %)
	NAVD	**142 867 (31.28 %)**
11 (voice)	V	143 288 (31.37 %)

Table B.3: Experiment with non-part-of-speech tags — final testing with `-o 1`. Improvements of algorithm which has been tested in Section 5.4 are marked in bold. The cells show size of the compressed file and the compression ratio.

-y	-x	genesis.txt	komunikace.txt	mloci.txt	zakonik.txt
2	N	36 169 (29.12 %)	15 729 (32.08 %)	143 326 (31.38 %)	250 172 (19.39 %)
	C	36 164 (29.12 %)	15 728 (32.08 %)	143 287 (31.37 %)	250 062 (19.38 %)
	V	36 231 (29.17 %)	**15 717 (32.06 %)**	143 397 (31.39 %)	250 553 (19.41 %)
3	V	36 150 (29.11 %)	15 721 (32.07 %)	143 223 (31.35 %)	250 497 (19.41 %)
4	C	36 175 (29.12 %)	15 734 (32.09 %)	143 349 (31.38 %)	250 168 (19.38 %)
5	A	36 130 (29.09 %)	15 718 (32.06 %)	143 204 (31.35 %)	249 929 (19.37 %)
	P	36 143 (29.10 %)	15 721 (32.07 %)	143 225 (31.35 %)	250 006 (19.37 %)
6	A	36 130 (29.09 %)	15 718 (32.06 %)	143 204 (31.35 %)	249 929 (19.37 %)
	P	36 148 (29.11 %)	15 722 (32.07 %)	143 234 (31.35 %)	250 012 (19.37 %)
7	P	**36 067 (29.04 %)**	15 728 (32.08 %)	**143 174 (31.34 %)**	**249 758 (19.35 %)**
	V	36 266 (29.20 %)	15 718 (32.06 %)	143 322 (31.37 %)	250 470 (19.41 %)
	J	36 135 (20.10 %)	15 718 (32.06 %)	143 217 (31.35 %)	**249 927 (19.37 %)**
8	V	36 208 (29.15 %)	15 729 (32.08 %)	143 333 (31.38 %)	250 758 (19.43 %)
9	A	36 130 (29.09 %)	**15 699 (32.02 %)**	**143 104 (31.33 %)**	250 174 (19.39 %)
	D	36 135 (29.10 %)	**15 715 (32.05 %)**	143 218 (31.35 %)	250 105 (19.38 %)
10	N	36 135 (29.10 %)	**15 709 (32.04 %)**	**143 195 (31.35 %)**	250 008 (19.37 %)
	A	**36 104 (29.07 %)**	**15 673 (31.97 %)**	**142 923 (31.29 %)**	250 038 (19.37 %)
	V	**36 060 (29.04 %)**	**15 683 (31.99 %)**	**142 958 (31.29 %)**	249 961 (19.37 %)
	D	36 131 (29.09 %)	15 723 (32.07 %)	143 304 (31.37 %)	250 100 (19.38 %)
11	V	36 169 (29.12 %)	15 736 (32.10 %)	143 288 (31.37 %)	250 743 (19.43 %)

Appendix C

Acronyms

API Application Programming Interface

EOF End-of-file

IDE Integrated Development Environment

GPL GNU General Public License

POS Part of Speech

PPM Prediction by Partial Matching

Appendix D

Contents of enclosed CD

- `program` .....
  - `doc` ..... the directory containing Doxygen-generated documentation
  - `NLC` ..... the directory containing the whole code project
    - `readme.txt` ..... the file with building and usage instructions
- `thesis` ..... the directory containing source code of the thesis
  - `DP_Navara_Jan_2016.pdf` ..... the thesis text in PDF format
- `README.txt` ..... the file with description of the CD contents