КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

«До захисту допущено»
Завідувач кафедри
Василь ТЕРЕЩЕНКО _____
(підпис)

«___» _____20___ р.

Кваліфікаційна робота
на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інформатика»

спеціальності 122 «Комп'ютерні науки»

на тему:

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ КОДІВ ІЗ РОЗДІЛЬНИКАМИ У СТИСКАННІ ПРИРОДНОМОВНИХ ТЕКСТІВ

Виконав студент 4 курсу
Максим КОВАЛЬЧУК
(підпис)

Керівник кваліфікаційної роботи:
доцент, доктор фіз.-мат. наук
Ігор ЗАВАДСЬКИЙ
(підпис)

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент
(підпис)

Київ – 2022

РЕФЕРАТ

Обсяг роботи 45 сторінок, 5 ілюстрацій, 4 таблиці, 26 джерел посилань.

БІНАРНО ЗАКОДОВАНІ ЗМІШАНІ КОДИ, ЕНТРОПІЙНЕ КОДУВАННЯ, КОДИ З РОЗДІЛЬНИКАМИ, СТАТИЧНІ КОДИ, СТИСНЕННЯ ДАНИХ, СТИСНЕННЯ ПРИРОДНОМОВНОГО ТЕКСТУ, ШВИДКИЙ АЛГОРИТМ ДЕКОДУВАННЯ.

Об’єктом роботи є розподіл ймовірностей слів в природномовних текстах і статичні коди, які задовольняють цю ймовірність. Було розглянуто різні, раніше запропоновані, коди. Один з них, з найкращою швидкістю декодування, був розглянутий детальніше і модифікований для покращення коефіцієнта стискання. Робота може бути розділена на 3 частини: аналіз універсальних методів стиснення даних та специфічних ентропійних кодів; дослідження нових запропонованих кодів, особливо їх алгоритмів декодування; підсумок експериментальних результатів та їх аналіз.

Метою роботи є проведення детального аналізу існуючих статичних кодів, що використовуються в стисканні природномовних текстів, їх модифікація для покращення коефіцієнту стискання та адаптація швидких алгоритмів декодування до нових кодів. Модифікації до коду BC3 було розглянуто. Сім’ю нових кодів було запропоновано разом з алгоритмами швидкого декодування для них.

Методом дослідження є розробка коду на C++ та точні виміри часу виконання програми. Інструментами розроблення є: Microsoft Visual Studio 2022, мова програмування C++. При розробці алгоритмів стискання було використано структури даних з C++ STL. Декодинг написаний більш низькорівнево і може бути легко адаптований до C. Для точного виміру часу було використано деякі функції специфічні до Windows. Для кращої швидкості декодингу було

використано 4 різні компілятори: компілятор Visual Studio 2022, два компілятори компанії Intel, а саме icl та icx, компілятор g++.

Головними результатами роботи є:

- Було запропоновано новий код BC7, який перевершує всі інші коди в швидкості декодування малих та середніх текстів;
- Було запропоновано сім'ю нових кодів BCMix. Нові коди BCMix перевершують всі інші коди в коефіцієнті стискання в переважній більшості випадків;
- Швидкі алгоритми декодування були запропоновані для сім'ї кодів BCMix. Результати показують, що швидкість декодування краща ніж в більшості інших кодів. Єдині два коди, BC7 та BC3, в яких краща швидкість декодування, мають на 3.5-10% гірший коефіцієнт стискання.

CONTENTS

ABBREVIATIONS AND CONVENTIONAL NOTATIONS

IDE – integrated development environment;

CPU – central processing unit;

SCDC – (s, c)-Dense Code;

RPBC – Restricted Prefix Byte Codes;

RMD – Reverse Multi-Delimiter Codes;

RLE – run length encoding;

AC – arithmetic coder;

RC – range coder;

ANS – asymmetric numerical systems;

tANS – table based implementation of asymmetric numerical systems;

LZ – Lempel-Ziv compressor;

DNA – deoxyribonucleic acid;

LZMA – Lempel-Ziv-Markov chain algorithm;

WRT – word replace technique;

MTF – move to front technique;

BWT – Burrow Weller Transform;

SA – suffix array;

PPM – Predict by Partial Matching technique;

CM – context-mixing technique;

HTML – HyperText Markup Language;

CSS – Cascading Style Sheets;

XML – Extensible Markup Language;

SIMD – single instruction multiple data;

VCL – Vector Class Library;

RAM – Random Access Memory;

ALU – arithmetic logic unit;

MMU – memory management unit;

OoOE – out-of-order execution;

BC3 – binary coded ternary code;

BC7 – binary coded sevenary code;

BCMix – binary coded mixed codes;

INTRODUCTION

Modern state of the research domain. A huge amount of information is transferred all over the world. Thousands of messages, photos, and videos are sent or published every second. The development of the internet gave a push to the research on data management technologies. Decreasing data size is key to saving millions of dollars on storage systems and transition costs. Therefore, a lot of compression algorithms have been developed.

The most recognizable terms in data compression are entropy coding and LZ-like algorithms. Huffman coding, range coder, and FSE have good speed and compression ratio. They are used in almost any production level compressors as the last step in the encoding model. LZ-like compressors use the repetitiveness of data.

Creating a new general-purpose compressor, that outperforms previously developed compressors, is a challenging task after 50 years of domain development. Nonetheless, the creation of separate compressors for each type of data is mainstream today. Texts, executables, DNA, photos, quantitative data – they all have unique properties that can be used to reduce the size.

Work motivation. Various static codes were proposed for natural text compression. Some of them are fast, others achieve a high compression ratio. Decoding speed is one of the most important factors when choosing the best code.

A lot of additional criteria may be considered taking into account typical operations with texts. Text is often watched starting not from the beginning. Automatic search of a word or a pattern in the text helps save tons of users' time. The support of random access or fast pattern matching algorithm to run over the compressed stream may make one code superior to other codes.

The research process is continuous and new approaches are constantly emerging. When the advantages of the new approach are fully evaluated, it can be included in existing software and systems.

The aim of the work. The aim of this work is to improve existing static codes used for natural text compression. Various modifications and generalizations of previous ideas often lead to the creation of new better approaches in different domains.

Object, methods and tools for research or development. The object of this work is static entropy codes. Microsoft Visual Studio 2022 and various C++ compilers were used during software development. For each developed executable, decompression time was measured and a compression ratio was recorded.

Potential practical applications. The results of this work can be used in systems that work with medium and large texts. Developed software can be used by booksellers like Amazon. We also propose integrating new codes in eReaders.

SECTION 1. DATA COMPRESSION OVERVIEW

As of today, data compression has been developing for more than 50 years. In this section we overview proposed algorithms and mainstream directions of data compression development, describe specifics of implementation for modern CPU, and consider the construction and decoding of byte-aligned SCDC and RPBC codes, RMD code and Huffman code.

1.1 Development of data compression algorithms

Data compression originated when both processors were much slower and computers had much less memory. Thus fancy compression schemes were not possible to use or even test. Compression techniques like RLE or delta coding are fast and require not much extra memory, but are hardly applicable for compression of texts and executables.

Important role in data compression is played by statistical compression. Consider an input data as a sequence $t_1, t_2, \dots, t_n$ of independent characters with values from the alphabet $\Sigma = \{c_1, c_2, \dots, c_m\}$. Each element of the sequence equal to c_i with the probability p_i. Then entropy defined as:

$$\mathrm{E} = -\sum_{i=1}^{m} p_i \log_2(p_i)$$

Entropy indicates an average number of bits required to store t_i. Let $cnt_i = p_i n$ show how many times c_i occurs in input sequence. Then, considering that the alphabet and probabilities are known, the minimum possible compressed size of the sequence is:

$$\mathrm{L} = -n\sum_{i=1}^{m} p_i \log_2(p_i) = -\sum_{i=1}^{m} cnt_i \log_2(p_i)$$

In practice, the probability distribution is sometimes known, at least approximately. In such case, static codes with fixed probabilities may be used. As of today, there are more than 100 different static codes. The most popular of them are unary, Golomb, Fibonacci, Elias, variable byte codes. In this work, we propose a new set of static codes together with fast decoding algorithms for them.

Such codes have different structures and properties, but two things usually hold:

- Code is complete, which means the sum of codewords probabilities equal to one. In other words, it is impossible to add a new codeword without losing the uniqueness of decoding;
- Code is prefix code. That is there is no codeword, which is prefix of other codeword;

In case probability distribution is not known, such as in general-purpose compressors, code is constructed in runtime during compression. The most popular such codes are Huffman codes. They work just like static codes but are constructed for each input sequence separately. Huffman codes are bit-aligned. Thus, they can achieve entropy level compression only if character probabilities equal to integer powers of two.

Different variations of AC and RC allow achieving compression close to entropy while working slower than Huffman codes. In addition, entropy coders may not have static probabilities but easily change them adaptively during compression. Such adaptive approaches often give a compression ratio better than entropy for the full input sequence. Huffman codes also have an adaptive approach, but updating the Huffman tree is harder and slower than updating probabilities for entropy coders.

ANS revolutionized entropy coding as it gives nearly entropy compression ratio while maintaining the speed of Huffman codes. Variation of tANS implemented in [11] is considered one of the best and most reliable libraries for entropy compression in production.

The family of LZ compressors plays an important role in data compression. It is easy to see that files like text have many exact repeats of words. Executables, DNA, and other files also have many repeats, which can be compressed. The main idea of LZ compressors is to use some kind of dictionary constructed based on already compressed text. LZ78-like compressors build a set of codewords for repeats from the text. LZ77-like compressors refer to recently encoded text as is. If a match with the recent text was found, it is encoded as two numbers: offset and length. Offset is a distance between the current position and the position of referred text. Depending on window size for recent text, type and parameters of match finder algorithm and type of entropy encoding algorithm, compression ratio, compression time and memory usage may vary.

Brotli, Zstd, gzip, bzip2, LZMA and others are popular LZ-like compressors. LZMA compressor is based on Markov chains and has several different types of matches that allow implicit encoding of fuzzy matches. An example of LZMA markup [8] is shown in figure 1.

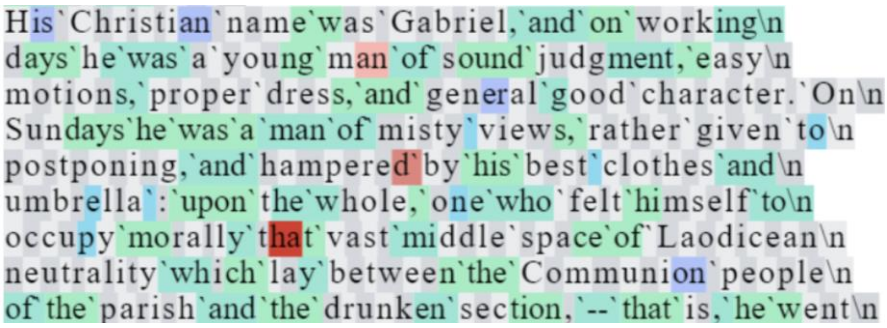

Figure 1: LZMA markup. Grey – literal packet, green – match packet, light blue – short repeat, blue – long repeat (0), red – long repeat (1-3)

Different data transformation techniques like WRT, RLE, MTF, BWT were proposed. The most interesting is BWT. It sorts all suffixes of text and then, for each suffix in a sorted array, it takes the letter just before that suffix. BWT can be uniquely transformed into original text without extra information. Fast construction of BWT is not an easy task. Nonetheless, BWT can be easily constructed based on SA, for which $O(n)$ algorithm was proposed in [12]. In [6] LZ factorization problem was solved in time $O(n)$ using SA. Unfortunately, greedy LZ factorization can be outperformed in

both compression ratio and speed by LZ-like algorithms with optimal parsing and faster match finder [7].

PPM together with an accurate entropy coder gives a better compression ratio than algorithms from the LZ family. PPM for each character from the alphabet estimates the probability that it will be next character based on contexts of last 0-15 bytes.

According to [13], the best compression ratio is achieved by CM algorithms and compressors based on neural networks. CM algorithms use several algorithms with different prediction models. Then probabilities are mixed and the entropy coder encodes the next symbol or bit. While such algorithms give the best compression ratio, they are the slowest. For comparison, Huffman codes [11] compression speed is more than 500mb/s, LZMA – 2mb/s and paq8px hardly compresses several megabytes in an hour.

1.2 Mainstream directions of data compression

Although lossy compression is not considered in this work, video compression is one of the most important topics in data compression. Companies like YouTube, Twitch, and Netflix store petabytes of video content. Such content should be highly compressed as it occupies a large part of the network and some countries do not have fast and unlimited internet. Decoding should be fast and consume not much energy, as mobile phones are an important platform with millions of users. Encoding also cannot take a lot of time, otherwise, companies would overpay for servers and online streaming would be impossible. Video quality is important for users, so compression algorithm should rely on human eye specifics.

Image compression is also important. Images are often stored in several resolutions, which produces redundant data. Social networks like Instagram and Facebook, and e-commerce companies like Rozetka and Amazon show dozens of

images on each web page. Reducing web page load time may increase income. According to [3], reducing Amazon's web page load by 100 milliseconds will increase their income by 1%.

HTML, CSS, javascript, json and XML files are sent across internet all the time.

Most popular applications and programs are frequently updated. Thus, software that can compress new patch using an old version of the program as a dictionary is highly important. One of such algorithms was proposed in [14] and it showed several times or tenfold better compression ratio than universal coders.

In [15] old web pages are stored. Not only the last version, but also all changes, that were made over the time, are available. Thus, an algorithm that can effectively compress new patches of web pages is necessary.

There are many other uses of data compression in production. It is hard to imagine a new general-purpose compression algorithm that is much better than previously developed. However, developing different compression algorithms for different data types remains an important topic. In this work, we consider the compression of natural text.

1.3 Specifics of implementing algorithms for modern CPU

Modern CPUs are developing all the time. Companies like Intel and AMD keep adding new instructions to their processors. Some instructions work with 128, 256, or even 512-bit registers, although the last are quite rare as of today. That is, using a 512-bit register you can multiply 16 32-bit numbers at once. There are some limitations to using such instructions. First, fast loading of data from memory is important and latter possible only if data placed contiguously. Second, effective usage of SIMD often require low level (almost assembler) programing. Libraries like VCL [18] may help with such complications, but it is available only for C++. Some compilers support auto vectorization during compilation, but they are imperfect ad cannot solve harder cases.

Fast memory is also an important element of modern-day optimizations. The fastest are CPU registers – they are already available to run instructions on. L1 cache memory goes next. It size mostly varies from 16 to 64 kilobytes on different processors. A lot of decoding algorithms rely on fitting the decoding table into L1 memory. L2 cache and RAM are slower. Random memory access to L2 and RAM may dramatically slow down the program. Thus accessing only contiguous data from L2 and RAM, if possible, is preferable.

Unlike in-order older processors, modern CPUs often have OoOE architecture. CPU consists of several units: control unit, arithmetic unit, address generation unit, and memory management unit. Registers are common memory for them, but mentioned units can work in parallel if they use independent registers. For example, while MMU loads some number from memory to register r1, ALU may add registers r2 and r3. An example of how OoOE improves performance relative to in-order execution showed in figure 2.

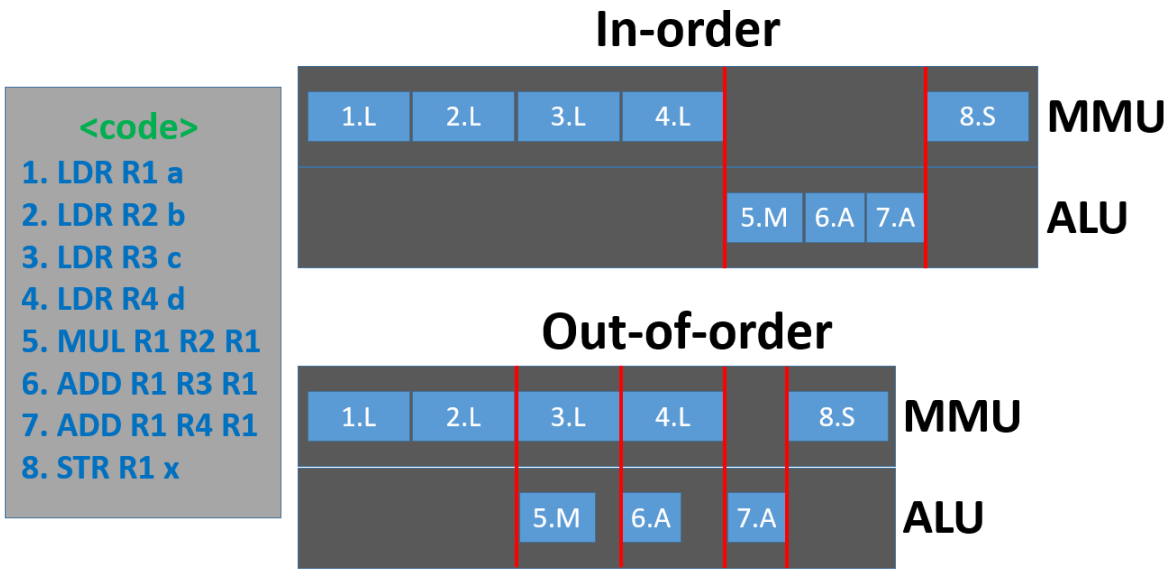

Figure 2: Computing x=a·b+c+d. In-order and out-of-order.

The architecture of the modern CPU is quite different from what was 20 years ago. The amount of memory and its speed have grown up. CPU architecture allows a lot of tricks to improve execution time. Reimplementation of old algorithm that exploits new features is often profitable. For example, the new library for SA

construction libsais [17] strongly relies on fast memory. It is more than two times faster than previous libraries like divsufsort [16], nonetheless, some tests showed that libsais is slower on older computers.

1.4 SCDC codes

This codes, together with Tagged Huffman, End-Tagged Dense, and some other codes belong to the family of byte-aligned codes. This means that any codeword comprises the whole number of bytes, which makes the very fast decoding possible, however, by the cost of compression ratio. The SCDC-codeword contains the sequence of ‘continuer’ bytes appended with the ‘stopper’ byte. All byte values less than some constant value C, $0<C<255$ are considered as continuers; the rest are stoppers. The value C can be determined either experimentally or by applying the algorithm given in [22]. All stopper bytes are in fact the delimiters and thus SC-dense codes possess all properties of codes with delimiters.

1.5 RPBC codes

SCDC construction would be more agile if the ‘border’ value C depends on the codeword length. I.e., if the first codeword byte is greater than R1, the codeword contains only this one byte. Otherwise, we check the second byte: if it is greater than R2, stop; otherwise, check the third byte etc. This approach implemented in the so called RPBC codes [23] provides a bit better compression ratio and a bit faster decoding. However, it does not allow us to access elements of the encoded file without its decompression, does not make possible a compressed search etc.

1.6 RMD codes

In [3] family of RMD codes were proposed. Each particular code is fully determined by the set of numbers $M = \{m_1, m_2, \dots, m_n\}$ that correspond to the lengths of delimiters. The set of all MD codewords consists of all codewords of form 01^{m_i} and all other codewords that meet the following requirements:

- a codeword starts with the prefix $01^{m_i}0$ for some $m_i \in M$;
- for any $m_i \in M$ a codeword does not end with a sequence 01^{m_i};
- for any $m_i \in M$ a codeword can contain the sequence $01^{m_i}0$ only as a prefix;

Sophisticated decoding algorithm was proposed in the original paper. The main idea of decoding is to use several decoding tables. The transitions between tables are determined by finite-state automaton.

1.7 BC3 codes

In [19] BC3 (binary coded trits) were introduced. Each of the codewords from the BC3 set consists of several digits and delimiter. Each digit and delimiter is encoded by 2 bits. Pairs “00”, “01”, and “10” represent digits 0, 1, and 2 respectively and “11” represent code delimiter. Let codeword consist of n digits. Then it can be written in the next form:

$$x_{n-1}x_{n-2} \dots x_0(11)$$

First 8 BC3 codewords are listed in table 1.

number	0	1	2	3	4	5	6	7
codeword	11	00 11	01 11	10 11	00 00 11	00 01 11	00 10 11	01 00 11

Table 1: First 8 codewords of BC3.

A simple encoding algorithm, which transforms nonnegative numbers to BC3 codeword, is listed in [20]. If BC3 codeword is separated from bit-stream, it can be easily decoded by the next formula:

$$x = \sum_{i=n-1}^{0} 3^{i-1}(x_i + 1)$$

As will be shown later, splitting each codeword and decoding it separately may slow down the program. Thus, decoding algorithm that reads bit-stream and decodes numbers at the same time may improve decoding performance. Fortunately, BC3 has fascinating properties, which allow codeword decoding without prior knowledge of its length. Algorithm BC3.1 uses such properties.

Algorithm: BC3.1 Decoding BC3 codeword

input : Bitstream of encoded BC3 codewords
output: Decoded BC3 codeword

```
1 x ← 0;
2 while true do
3 |   digit ← 2 bits from bitstream;
4 |   if digit == 11 then
5 |   |   return x;
6 |   else
7 |   |   x ← x * 3 + digit + 1;
```

Algorithm BC3.1 is quite slow in practice, as it has unpredictable if. However, it uses the important property, which allows to not store any extra information about decoded prefix else than the decoded number itself. That is, fast decoding algorithm exists. We can decode 5-6 digits (10-12 bits) at a time using precomputed tables and avoiding unpredictable branches. The possible structure for the block of 6 digits showed in figure 3.

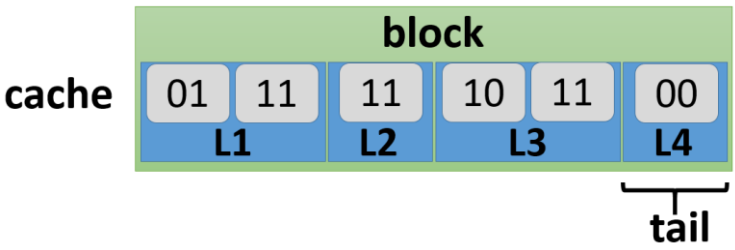

Figure 3: Possible structure of decoding block.

Algorithm BC3.2 is fast decoding algorithm for 32-bit numbers. It uses the following variables and tables:

- block is variable that contains 10 or 12 bits that are to be decoded;
- blockLen equal to 10 or 12 and blockMask equal to $2^{10} - 1$ or $2^{12} - 1$;
- L21, L43 are 64-bit tables that contain decoded numbers. Block may have less than four numbers, in which case L4, L3, L2 may be equal to 0;
- blockN is table that contains the number of delimiters "11" in block;
- pw3 is table that contains 3 to the power of the number of digits in the first number of the block;
- cache contains previously decoded part of the first number;
- pos is the position in the array of decoded numbers;

Algorithm: BC3.2 Iteration of fast decoding for 32-bit numbers

input : Encoded BC3 **bitstream**, output array **out**, auxiliary numbers **cache** and **pos**
output: Decoded 5 or 6 digits from **bitstream**

```
1 block ← bitstream & blockMask;
2 bitstream ← bitstream >> blockLen;
3 prefix ← cache · pw3[block];
4 *(uint64_t*)(out + pos) ← prefix + L21[block];
5 *(uint64_t*)(out + pos + 2) ← L43[block];
6 pos ← pos + blockN[block];
7 cache ← tail[block] + blockN[block] ? 0 : prefix;
```

Line 7 of algorithm BC3.2 contains a conditional choice. Which may lead to unpredictable if. However, the compiler will probably replace conditional choice with MOVcc instruction. To be sure unpredictable branches were not generated, *blockN[block] ? 0 : prefix* may be replaced with *(~(blockN[block] == 0) & prefix)*.

In case text is small or middle-sized, the number of unique codewords is smaller than 2^{16}. In such case, numbers may be stored in 16-bit variables, which allows storing L1, L2, L3, L4 numbers in one 64-bit variable. Thus, lookup tables L21 and L43 can be replaced with lookup table L4321. This is reflected in algorithm BC3.3.

Algorithm: BC3.3 Iteration of fast decoding for 16-bit numbers

input : Encoded BC3 **bitstream**, output array **out**, auxiliary numbers **cache** and **pos**
output: Decoded 5 or 6 digits from **bitstream**

```
1 block ← bitstream & blockMask;
2 bitstream ← bitstream >> blockLen;
3 prefix ← cache · pw3[block];
4 *(uint64_t*)(out + pos) ← prefix + L4321[block];
5 pos ← pos + blockN[block];
6 cache ← tail[block] + blockN[block] ? 0 : prefix;
```

One more important topic is reloading bitstream with bits from the compressed file. In [20] next scheme was proposed:

1. Load 64 bits (8 bytes) from file or RAM;
2. Perform 6 or 5 iterations of algorithms BC3.2 or BC3.3, each decodes 10 or 12 per iteration. Thus 60 bits will be decoded;
3. Decode 4 last bits with smaller tables;
4. If stream of compressed codewords is not ended, go to step 1.

1.8 Huffman codes

Huffman codes are one of the most recognizable theme in data compression. They have 2 important properties, which are critical for production. First, Huffman codes are built for each input sequence separately. Therefore, they are good for almost any kind of real-world input data. Second, in case the input sequence contains no more than 256 unique values (i.e. input sequence consists of bytes), they are fast to encode and even faster to decode. In this section, we describe how to construct Huffman codes, describe a modification to Huffman codes, and list fast decoding algorithm for modification.

Let input data on alphabet $\Sigma = \{c_1, c_2, \dots, c_m\}$ is given. For each c_i, cnt_i denotes how many times c_i occur in the text. Construction of Huffman code starts with creating m nodes, each corresponding to a unique value in the alphabet and containing the number of occurrences. Those nodes are added to some set sorted by the number of occurrences. While the number of nodes in the set is bigger than one, the next step is

performed. Two nodes from the set with the smallest numbers are taken and deleted from the set. A new node is created, and it becomes the parent for two taken nodes. 2 edges prom parent node to children nodes created, one has value 0 and the other 1. The new node number is equal to the sum of numbers in children nodes. This new node is added to the set.

After construction, the set will contain a single node. This node is the root of the Huffman tree. The leaves of the tree are all values from the alphabet. The sequence of values on edges from the root to the leaves is called Huffman codes. An example of a Huffman tree for string “cabadbaaca” is shown in figure 4.

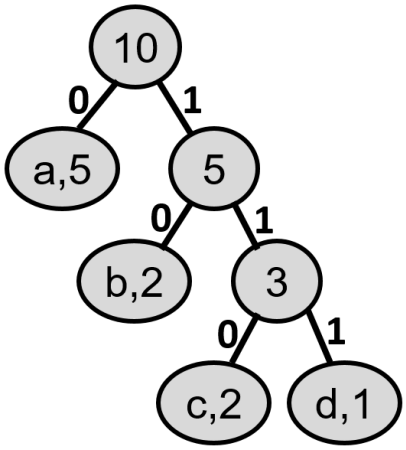

Figure 4: Huffman tree for string “cabadbaaca”. Codes are a – 0, b – 10, c – 110, d – 111.

There is a faster method for construction of Huffman codes, but we are interested in the fast decoding method, therefore speed of construction is not very important for us. The idea of the decoding algorithm is to be able to decode a single character at a time as fast as possible. This is important, as Huffman codes do not have arithmetical properties like BC3. Thus, decoding long Huffman codes may require hundreds of decoding tables for different prefixes. Let Huffman code is already constructed. We want to build a modified set of codewords, such that the longest codeword consists of maximum of 11 bits.

This problem could be solved in $O(11m)$ via package-merge algorithm [21]. Now we want to develop fast decoding algorithm. It can be done using 2 tables [2]:

- len[bits] – length of codeword stored as the prefix of 11 bits;
- symbol[bits] – byte character that corresponds to Huffman codeword;

Then decoding of character can be done in just 3 lines like in algorithm HUF.1.

Algorithm: HUF.1 Fast decoding of limited-length Huffman codewords

input : Encoded Huffman **bitstream**, position in bitstream **pos**
output: Single decoded character from **bitstream**

```
1 block ← (bitstream >> pos) & 2047;
2 pos ← pos + len[block];
3 return symbol[block];
```

The first line of algorithm HUF.1 corresponds to single assembler instruction *bextr*. The second line consists of table lookup and addition. The third line consists of table lookup and store instruction. Both tables have 2048 elements, each element fits in the byte. Thus, the total size of tables is 4 kilobytes, which easily fits in the L1 cache, even for old processors. The bitstream has to be updated every 4 or 5 characters. In [2] decoding 3 streams at the same time was proposed.

SECTION 2. RESEARCH AND SOFTWARE DEVELOPMENT

In this section, we propose two modifications to the BC3 code. We also propose several decoding methods that theoretically should be fast. Experimental results are listed in section 3.

2.1 Compression model

In this work, we consider the compression of English texts. Compression starts with the separation of all unique words in the text and mapping them to non-negative integers. Mapping is based on the frequency of words. The more frequent words are the smaller rank it gets. The word that appears in the text the most has rank 0.

Then, based on sequence of words in text, sequence of word ranks is created. In this work we focus on compressing sequence of ranks. The 2 main goals are high compression ratio and fast decoding. Encoding speed is also important, but compression is done once. Considering encoding speed of previous works, the slowest part of encoding is mapping words to ranks.

2.2 BC7

Similarly to BC3 code that uses 2-bit digits, we consider BC7 code that uses 3-bit digits. We suppose BC7 may be more beneficial for large texts than BC3.

BC7 codeword consists of several (possibly zero) digits with values “000”, “001”, “010”, “011”, “100”, “101”, “110” and single delimiter “111”:

$$x_{n-1}x_{n-2}\dots x_0(111)$$

The first 14 BC7 codewords are listed in table 2.

number	codeword	number	codeword
0	111	7	110 111
1	000 111	8	000 000 111
2	001 111	9	000 001 111
3	010 111	10	000 010 111
4	011 111	11	000 011 111
5	100 111	12	000 100 111
6	101 111	13	000 101 111

Table 2: The first 14 codewords of BC7.

BC7 has the same arithmetical properties as BC3. BC7 codeword can be decoded using one of the two next formulas:

$$x = \sum_{i=n-1}^{0} 7^{i-1}(x_i + 1)$$

$$x = 7(\dots 7(7(x_{n-1} + 1) + x_{n-2} + 1) \dots x_1 + 1) + x_0 + 1$$

Each digit of the codeword is independent of all other digits and delimiter. It allows fast construction of all codewords up to some maximal rank. In practice, maximal rank is equal to the number of unique words in the text. Algorithm BC7.1 constructs all such codes together with all their lengths. After constructions, the sequence of ranks of words from the text can be easily encoded one by one.

Algorithm: BC7.1 Fast generation of all BC7 codewords up to some maximal rank

input : Maximal rank **n**, array of powers of 7 **pw7**
output: All codewords **codes** and their lengths **codeLens**

```
1 codes[0] ← 7;
2 codeLens[0] ← 3;
3 for i ← 0 to n do
4 |   for j ← 0 to 7 do
5 |   |   num ← i + (j + 1) * pw7[codeLens[i] / 3 + 1];
6 |   |   if num < n then
7 |   |   |   codes[num] ← (code[i] << 3) | j;
8 |   |   |   codeLens[num] ← codeLens[i] + 3;
```

Algorithm BC7.2 decodes one digit at a time. Such an approach is simple but slow in practice. We implement and test it as a reference, to see how much decoding algorithm described below is better than the simple one.

Algorithm: BC7.2 Decoding BC7 codeword

input : Bitstream of encoded BC7 codewords
output: Decoded BC7 codeword

```
1 x ← 0;
2 while true do
3 |   digit ← 3 bits from bitstream;
4 |   if digit == 111 then
5 |   |   return x;
6 |   else
7 |   |   x ← x * 7 + digit + 1;
```

The iteration of the BC7 fast decoding algorithm is similar to the fast decoding of BC3. We try to help the compiler with optimizations of C++ code via some tricks that do not change the algorithm overall. First, we try to prefetch the necessary values from all tables at the beginning of the iteration. We also try to rearrange operations in such a way, that variable computed in the current line is not used in the next line. It may allow better performance on OoOE processors. In algorithm BC7.3, we decode 4 digits at a time, which is 12 bits. It is almost guaranteed that the most frequent word in natural language text never happens three times in a row. Thus a block of 12 bits has at most 3 delimiters. We replace 64-bit lookup table L43 with 32-bit table L3. Such optimization somewhat reduces table size, so they use less L1 cache memory.

Algorithm: BC7.3 Iteration of fast decoding for 32-bit numbers

input : Encoded BC7 **bitstream**, output array **out**, auxiliary numbers **cache** and **pos**
output: Decoded 4 digits from **bitstream**

```
1  block ← bitstream & blockMask;
2  bitstream ← bitstream >> blockLen;
3  _pw7 ← pw7[block];
4  _L21 ← L21[block];
5  _L3 ← L3[block];
6  _tail ← tail[block];
7  _n ← n[block];
8  prefix ← cache · _pw7;
9  out[pos + 2] ← _L3;
10 *(uint64_t*)(out + pos) ← prefix + _L21;
11 pos ← pos + _n;
12 cache ← _tail + (∼(_n == 0) & prefix);
```

The Algorithm for decoding 16-bit numbers is almost the same as algorithm BC7.3, but lookup to two tables L21 and L3 is replaced by lookup to single table L4321.

As 3-bit digits do not fit in the 64-bit bitstream, the same macroiterations as in BC3 codes are impossible. We propose to decode 96 bits in 1 macroiteration. First, we load 32-bit in the bitstream variable and execute 2 fast decoding iterations. Then we load the next 32 bits and execute 3 more iterations. Then, again, we load the last 32 bits and execute the last 3 iterations in macroiteration. Such a scheme of macroiteration allows avoiding conditional branches for bitstream reloading. The full process of decoding is implemented in algorithm BC7.4.

Algorithm: BC7.4 Complete decoding of BC7 data

```
   input  : Array in of 32-bit elements with BC7 codes, array length len
   output: Decoded to out word ranks
 1 cache ← 0;
 2 bitstream ← 0;
 3 pos ← 0;
 4 i ← 2;
 5 while i < len do
 6 |   bitstream ← in[i - 2];
 7 |   fastDecodingIteration(out, cache, bitstream, pos);
 8 |   fastDecodingIteration(out, cache, bitstream, pos);
 9 |   bitstream ← bitstream | (in[i - 1] << 8);
10 |   fastDecodingIteration(out, cache, bitstream, pos);
11 |   fastDecodingIteration(out, cache, bitstream, pos);
12 |   fastDecodingIteration(out, cache, bitstream, pos);
13 |   bitstream ← bitstream | (in[i] << 4);
14 |   fastDecodingIteration(out, cache, bitstream, pos);
15 |   fastDecodingIteration(out, cache, bitstream, pos);
16 |   fastDecodingIteration(out, cache, bitstream, pos);
17 |   i ← i + 3;
18 decodeLastFewBytes(out, cache, in, pos - 2, len);
```

2.3 BC7 vectorized

According to [2], decoding several compressed streams in parallel may speed up the program. In parallel does not mean using several threads but exploiting such a trick as OoOE or data vectorization. We attempted to decode 2, 4, 8 streams at the same time. We also attempted to use [18] to force the compiler using 128-bit and 256-bit registers for decoding 8 streams at the same time.

For some reasons, to be investigated later, neither compiler auto-vectorization nor VCL gives a faster algorithm. Even more, all attempts ended up with a much slower algorithm. We concluded that decoding algorithms for BC3 and BC7 are hard or impossible for effective vectorization usage. This is probably due to the large number of table lookups that, unlike arithmetical and logical operations, are not gaining much from intrinsic instructions. Also, some parts of code, like writing to decoded streams, require storing variables in discontinued memory locations, which require slow extraction of numbers from 256-bit register to 4 64-bit registers.

For the reasons described above, we will list the result only for decoding with 2 streams. To help the compiler with auto-vectorization, we tested the next modifications of code:

- *nopw7* – reduces 2 lookups to tables *mypw7* and *pw7* to single lookup, but new table takes almost 4 times more memory than *mypw7* and *pw7* together;
- *restr* – adding C++ keyword *__restrict* to decoded streams pointers;
- *arrs* – passing function parameters as arrays instead of passing each variable separately;

2.4 BCMix

So far, in this work we focused on BC3 and BC7 codes. Their digits have constant length of 2 or 3 bits. The main idea of BCMix codes is constructing such code that each digit have its own length. We considered 3 possible lengths of digits: 2 bits, 3 bits and 4 bits. We denote particular mixed code as letter M together with lengths of first several digits. For example M3242 denotes code, which first digit consist of 3 bits, second digit consist of 2 bits, third digits consist of 4 bits, fourth digit consist of 2 bits, and all other digits consist of 2 bits. All digits which size is not specified in code name have length 2 bits. Thus, M3222 and M3 denote the same code.

Consider we concluded on some particular binary coded mixed code. Let denote the length of a particular digit as $dLen_i$. There are some additional arrays that need to be defined to simplify further algorithms. Let $dMask_i$ be the number of different values that the corresponding digit can represent excluding the delimiter. It can be computed using the next formula:

$$dMask_i = 2^{dLen_i} - 1$$

Also, $dMask_i$ represents bit mask for the corresponding digit and delimiter value. We also define two variables $dPow_i$ and $dPref_i$:

$$dPow_i = \begin{cases} 1, i = 0 \\ dPow_{i-1} \cdot dMask_{i-1}, else \end{cases}$$

$$dPref_i = \begin{cases} 0, i = 0 \\ dPref_{i-1} + dPow_{i-1}, else \end{cases}$$

Value of $dPow_i$ can be interpreted as increment of digit. Value of $dPref_i$ denotes how many BCMix codewords that have no more than i digits exist.

Finally, BCMix codeword is defined as $x_0x_1 \dots x_{n-2}x_{n-1}(dMask_n)$. There are two important differences between BCMix and BC7 or BC3 definitions. First, different digits may have different lengths. It is also relevant for the delimiter. Second, the ordering of digits is reversed. BCMix codewords start with the least significant digit. BCMix codes do not have some mathematical properties of BC7 code. Thus, the same fast decoding is not possible. In addition, if the ordering of digits is not reversed, i.e. the first digit is x_{n-1}, it is impossible to tell the index of this digit and its length. Given separate digits of codeword $x_0x_1 \dots x_{n-2}x_{n-1}(dMask_n)$, its rank can be computed using the next formula:

$$x = \sum_{i=0}^{n-1} (x_i \cdot dPow_i) + dPref_n$$

Generating all codewords and their lengths is also different from BC7. Algorithm BCMix.1 generates all codewords. Generation is done separately for each

digit, without using the information of other codewords. Algorithm is similar to converting decimal number to binary, but the powers of two are replaced by array $dPow$.

Algorithm: BCMix.1 Generation of all codewords up to some rank

```
   input  : Max rank n
   output: Codewords codes and their lengths codeLens
 1 digitN, curNum ← 0;
 2 for i ← 0 to n do
 3 |  if curNum == dPow[digitN] then
 4 |  |  curNum ← 0;
 5 |  |  digitN ← digitN + 1;
 6 |  code, shift ← 0;
 7 |  t ← curNum;
 8 |  for j ← 0 to digitN - 1 do
 9 |  |  code ← code | ((t mod dMask[j]) << shift);
10 |  |  t ← t div dMask[j];
11 |  |  shift ← shift + dLen[j];
12 |  code ← code | (dMask[digitN] << shift);
13 |  codes[i] ← code;
14 |  codeLens[i] ← shift + dLen[digitN];
15 |  curNum ← curNum + 1;
```

A simple digit-by-digit decoding algorithm is somewhat similar to converting a string variable to an integer one. It is implemented in algorithm BCMix.2.

Algorithm: BCMix.2 Decoding BCMix codeword

```
   input  : Encoded bitstream and its length len
   output: Decoded codeword
 1 x, digitN ← 0;
 2 while true do
 3 |  digit ← bitstream & dMask[digitN];
 4 |  bitstream ← bitstream >> dLen[digitN];
 5 |  len ← len - dLen[digit];
 6 |  if digit == dMask[digitN] then
 7 |  |  x ← x + dPref[digitN];
 8 |  |  break;
 9 |  else
10 |  |  x ← x + digit · dPow[digitN];
11 |  |  digitN ← digitN + 1;
12 return x;
```

2.4.1 Construction of the best BCMix code

As was mentioned before, BCMix codes cannot be decoded with the same fast algorithm as BC3 or BC7. Therefore, the main advantage of BCMix codes is a better compression ratio. An effective algorithm that can find the best code is necessary.

Particular code is fully determined by lengths of digits. Assuming the biggest codeword consist of 10 digits, there are 59049 different codes. Taking each of them separately and scanning full text to compute compressed size is impractical, as it will take too much time.

A simple improvement is to take frequencies of words that were computed during text to ranks transformation. It allows iterating through all unique words instead of iterating through the full text. Nonetheless, the algorithm continues to be quite slow and may take several minutes for large texts.

We propose an algorithm that is linear-time to the number of codes and number of unique words. For the beginning, let's take word frequencies from text to the ranks encoding stage. Compute prefix sums p_i on array of frequencies. It other words, p_i denotes how many times words with frequencies with ranks less or equal than i occur in text. Prefix sums allow answering in $O(1)$ how many times words with ranks from some segment $[l:r]$ occur in text.

Algorithm: BCMix.3 Effective search of the best BCMix code

input : Temporary array with digit lengths **dSz**, digit position **dN**, prefix sums array **pref**, max rank **n**, number of fixed codewords **fixedN**, number of codewords without delimiter **auxN**, length of codewords without delimiter **auxLen**, length of compressed text with only fixed codewords **fullBitsSz**

output: BCMix code that gives the best compression ratio

```
if fixedN >= n then
    updateTheBestCode(dSz, dN, fullBitsSz);
    return;
dSz[dN] ← 2;
newBitsSz ← fullBitsSz + (auxLen + 2) · getSum(fixedN, fixedN + auxN - 1, pref);
BCMix.3(dSz, dN + 1, pref, n, fixedN + auxN, auxN · 3, auxLen + 2, newBitsSz);
dSz[dN] ← 3;
newBitsSz ← fullBitsSz + (auxLen + 3) · getSum(fixedN, fixedN + auxN - 1, pref);
BCMix.3(dSz, dN + 1, pref, n, fixedN + auxN, auxN · 7, auxLen + 3, newBitsSz);
dSz[dN] ← 4;
newBitsSz ← fullBitsSz + (auxLen + 4) · getSum(fixedN, fixedN + auxN - 1, pref);
BCMix.3(dSz, dN + 1, pref, n, fixedN + auxN, auxN · 15, auxLen + 4, newBitsSz);
```

Algorithm BCMix.3 brute forces all mixed codes recursively, starting with codes consisting of 1 digit. On each step of recursion, it adds a new digit to the code. Several variables are to be held:

- A temporary array with digit lengths is used during updating of the best code;
- Digit position denotes how many digits were already processed and the index of the current digit to be processed;

- Array of prefix sums on words frequencies;
- Max rank. Codewords that denote larger ranks are not necessary during decoding;
- Fixed codewords are ones that already have delimiter, thus they haven't to be expanded;
- Auxiliary codewords consist of digits only and don`t have delimiter. They will be transformed to fixed via adding several more digits and delimiter.
- Length of auxiliary codewords in bits;
- Length of compressed text if only ranks with fixed codewords are considered;

Algorithm BCMix.3 calls function $getSum(l, r, pref)$ that simply calculates sum on segment $[l: r]$ using prefix sums.

According to experimental results, for all tested texts it is enough to brutforce lengths of the first 4 digits. All other digits always consist of 2 bits. Due to such optimization, it is enough to test only 81 different codes, which highly reduces execution time.

2.4.2 Two phase BCMix decoding

As digits of BCMix codes have varying digit lengths and their digits are stored in reverse order relative to BC7 or BC3, the same fast decoding is impossible. We tested various algorithms to find new fast decoding for mixed codes, as they give a better compression ratio.

In this section, we propose a two phase decoding algorithm. This algorithm is in the middle by the number of required decoding tables. Two other algorithms have fewer and more decoding tables described later in the text.

Two phase BCMix decoding algorithm uses up to 8 decoding tables, up to 5 in the first phase and 3 in the second phase. In the first phase, we separate codewords from the encoded stream. In the second phase, we convert BCMix codewords to word ranks.

Let first consider splitting encoded bitstream to separated codewords. On each iteration of decoding, we take block of 10 bits from bitstream. Then we use tables similarly to algorithm BC7.3. At first, we extract bits of the very first number in the block and store them in the *cache* variable. As bits of first number may started in previous blocks, we store additional variable, which equal to the number of already decoded bits of first number. Then we write to array of decoded codewords first, second, and third codewords, if such fully present in the block. Then we update *cache* variable and its length based on number of delimiters in the block and tail of the block. Finally, we update pointer to the table.

As was mentioned before, the splitting phase requires up to 5 tables. The decoding of the block depends on the length of the digits. Thus, for different sequences of digit lengths, we need different tables. Let's consider code M4233, in which digit lengths are 4, 2, 3, 3, 2, 2, 2, 2, 2… Depending on the first digit of the block, there are 5 different combinations of digits: 42332, 23322, 33222, 32222, and 22222. In other words, for each of the first 4 digits, a separate decoding table is required. All next digits have a common length of 2, thus can be decoded with the same table. For example, code M3 requires only 2 decode tables for digit lengths sequences 32222 and 22222. As was mentioned before, only the first 4 digit lengths are brutforced during encoding, therefore no more than 5 tables are required.

The iteration of the splitting phase is shown in algorithm BCMix.4. The last line of the algorithm looks like a full copy of the decoding table. Such an approach is very slow. Storing the index of the current table is the better approach, but in such case table lookup will consist of 2 indexes. Therefore, in practice, we use a C++ pointer that is pointing to the current decoding table. Each macroiteration consists of 3 iterations, after which bitstream is reloaded.

One more difference between the decoding of BC7 code and BCMix code is the varying length of the block. On each iteration of two phase decoding, we take 10 bits out of bitstream. However, consider code M3. The first 3 digit lengths are 3, 2, 2, 2, and 2. Thus, 10 bits fully cover only the first 4 digits and half of the fifth digit. In such a case, it is possible to decode only 9 out of 10 bits and the last bit should remain in the bitstream for the next iteration. Therefore, relative to BC7 decoding, one additional lookup array is required for keeping the bitstream, and reloading of the bitstream is almost impossible without unpredictable branching.

Algorithm: BCMix.4 Iteration of first phase of two phase decoding

input : Encoded **bitstream** and other auxiliary variables like pointer to the current table **tb**, decoding tables
output: Decoded up to 10 bits of **bitstream**

```
1  block ← (bitstream >> bitpos) & blockMask;
2  bitpos ← bitpos + tb.shift[block];
3  cache ← cache | (tb.firstNum[block] << cacheLen);
4  cacheLen ← cacheLen + tb.firstLen[block];
5  out[pos] ← cache;
6  *(uint64_t*)(out + pos + 1) ← tb.L32[block];
7  pos ← pos + tb.n[block];
8  cache ← tb.n[block] ? tb.tail[block] : cache;
9  cacheLen ← tb.n[block] ? tb.tailLen[block] : cacheLen;
10 tb ← tables[tb.nextTable[block]];
```

The second phase can be done as in algorithm BCMix.2. However, decoding BCMix codewords can be done faster, using 3 lookup tables. Let split any codeword into 3 blocks. Each block consists of an integer number of digits. The first block consists of no more than 11 bits, the second – 10 bits, third – 10 bits. As all digits are independent, each block is independent of the others. The scheme of decoding is shown in figure 5. Algorithm BCMix.5 implements fast codeword decoding.

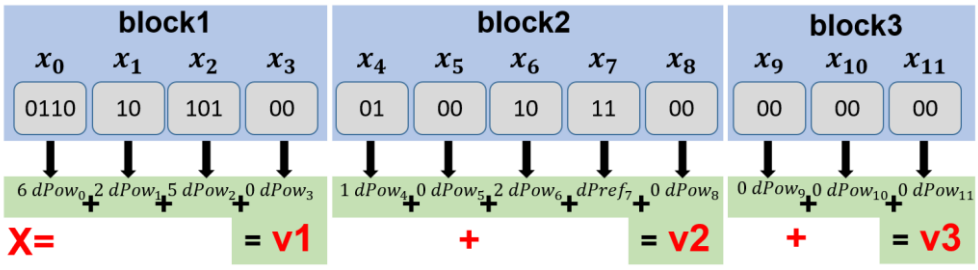

Figure 5: Fast BCMix codeword decoding scheme.

Algorithm: BCMix.5 Fast decoding of BCMix codeword

input : Codeword **code**, length of first 2 blocks **sh1** and **sh2**, bitmasks of first 2 blocks **m1** and **m2**, decoding tables **t1**, **t2**, **t3**

output: Decoded codeword

```
1 block1 ← code & m1;
2 block2 ← (code >> sh1) & m2;
3 block3 ← code >> (sh1 + sh2);
4 v1 ← t1[block1]; v2 ← t2[block2]; v3 ← t3[block3];
5 return v1 + v2 + v3;
```

2.4.3 Single codeword BCMix decoding

In this section, we propose an algorithm that is similar to the Huffman decoding described in [2]. The main idea of the original article is to reduce the length of Huffman codewords to 11 bits, so each codeword can be decoded via one lookup to two arrays of the small table in the L1 cache. BCMix codewords may reach up to 29 bits for large texts, therefore cannot be decoded in one lookup to the small table.

We propose an algorithm that decodes no more than 1 codeword at a time. The algorithm require 3 decoding table. Those tables are similar to ones used in algorithm BCMix.5, but they are extended by some extra information like bitstream shift.

In total each table consists of 3 lookup arrays and one non-array variable:

- $shift[block]$ – how many bits of block belongs to the first number. This number should cover integer number of bits;
- $L[block]$ – value that bits of codeword contribute to decoded rank;
- $n[block]$ equal to 1 if codeword ends in current block or 0 otherwise;
- $nxtTable$ – pointer to the next table if codeword is not ended in the current block;

Algorithm: BCMix.6 Iteration of single codeword BCMix decoding

input : Encoded **bitstream** and other auxiliary variables like pointer to the current table **tb**, decoding tables
output: Decoded part of codeword or fully decoded codeword

```
1 block ← bitstream & blockMask;
2 bitstream ← bitstream >> tb.shift[block];
3 bitlen ← bitlen - tb.shift[block];
4 cache ← cache + tb.L[block];
5 tb ← tb.n[block] ? firstTable : tb.nextTable;
6 out[pos] ← cache;
7 pos ← pos + tb.n[block];
8 cache ← tb.n[block] ? 0 : cache;
```

Algorithm BCMix.6 implements described idea. The main advantage of algorithm is the much smaller number of lookups per iteration – only 3. In addition, the biggest among the 3 arrays 32-bit array *L*, which holds decoded values, is much smaller than 96-bit or 128-bit arrays in other decoding algorithms. The disadvantage of the algorithm is that it often returns a lot of bits to the bitstream. This happens when the first or second digit of the block is the delimiter. Overall performance results are listed in section 3.

2.4.4 Digit aligned BCMix decoding

We also propose a fast decoding algorithm that is similar to the decoding of RMD codes. The algorithm requires more tables than the two previously proposed algorithms, but both process most of the bits of block and decodes everything in 1 phase.

The idea of the algorithm is to decode no more than 2 codewords in one iteration, up to the last digit possible. In other words, the algorithm decodes all digits taken from bitstream to block, but no more than two delimiters. In practice, it will usually decode all digits of the block. As the algorithm can finish decoding on any digit of the codeword, there should be a separate decoding table for each digit of the biggest codeword. Considering there are no more than 32 bits in the longest codeword and all digits have a length 2, no more than 16 decoding tables are required. In practice, only 8-14 tables will be used, depending on the number of unique words in the text.

Algorithm: BCMix.7 Iteration of digit aligned BCMix decoding

input : Encoded **bitstream** and other auxiliary variables like pointer to the current table **tb**, decoding tables
output: Decoded bitstream

```
1 block ← bitstream & blockMask;
2 oldPos ← pos;
3 bitstream ← bitstream >> tb.shift[block];
4 bitlen ← bitlen - tb.shift[block];
5 pos ← pos + tb.n[block];
6 *(uint64_t*)(out + oldPos) ← cache + tb.L21[block];
7 tb ← tables[dt.nextTableIndex[block]];
8 cache ← out[pos];
```

Although the idea of proposed in this section algorithm is quite different from the one described in the previous section, there is not very much difference in decoding tables. Arrays $n[block]$ and $shift[block]$ play the same role and have the same size. Array of 32-bit elements $L[block]$ of algorithm BCMix.6 is replaced with array of 64-bit elements $L32[block]$. The number of processed digits and position of the delimiter inside them is not easy to compute on a fly, array $nextTableIndex[block]$ stores index of next decoding table. One more trick used in algorithm BCMix.7 is creating a copy of the position in decoded array at the beginning of iteration and not changing both variables later during iteration. It is believed that such a trick may slightly optimize the algorithm on some processors.

SECTION 3. EXPERIMENTAL RESULTS

In this section, we show the results of the experimental comparison of different algorithms on different input data. At first, we list English texts used for testing and some of their properties. Next, we compare different variations of the vectorized BC7 decoding algorithm. Afterward we compare different variations of BC7 and BC3 decoding algorithms. Finally, we compare different decoding algorithms for BCMix codes with BC7 and previously proposed decoding algorithms.

3.1 Input data

As input English texts we choose 5 files of different sizes. Three smaller files alice29.txt, book1, bible.txt are taken from [1]. Large files with English Wikipedia articles enwik8 and enwik9 are taken from [4] and [5]. Properties of files are listed in table 3.

name	Size (bytes)	Unique words	Total words	Word entropy (bits)	Huffman encoded size (bytes)
alice29.txt	152,089	5949	27334	9.8495	33746
book1	768,771	21075	141274	10.4973	185838
bible.txt	4,047,392	28659	766111	9.4802	911093
enwik8	100,000,000	1439355	13303056	13.4364	22390527
enwik9	1000,000.000	8859143	129347859	13.7344	222505605

Table 3: English texts and their properties.

Huffman encoded size is ignoring the size of the Huffman table. The first three files are mostly raw English texts. enwik8 and enwik9 contain markup, tags, links, and other extra information required for rendering in web browsers.

3.2 Results for vectorized BC7 decoding

As mentioned in section 2.3, vectorized version of BC7 decoding has three possible modifications aimed to help the compiler with optimization. We tested different combinations of modifications and compared them to the original 1-stream fast BC7 decoding. Tests were done on an Intel Core i7-10510U processor. We tested 4 different compilers: Visual Studio 2022 compiler, two Intel compilers icl and icx, and g++ compiler. Results are listed in table 4. We tested decompression speed on enwik8 text only. For each variation and compiler, we run decompression 1000 times and take minimal time.

name	vs (ms)	icl (ms)	icx (ms)	g++ (ms)
BC7, 2 streams	44.23	42.98	43.28	64.59
BC7, 2 streams, nopow7	51.11	50.30	50.65	72.89
BC7, 2 streams, nopow7, arrs	67.94	50.03	51.33	69.70
BC7, 2 streams , restr	53.33	52.54	51.10	74.73
BC7, 2 streams, restr, arrs	66.32	50.91	51.77	69.41
BC7, 2 streams, restr, nopow7	50.95	50.53	51.24	72.01
BC7	**34.64**	**35.24**	**35.64**	38.94

Table 4: Decoding time of variations of 2-stream BC7 algorithm.

As we can see, in all cases original BC7 decoding is much faster, than 2-stream modification. Such a result is unexpected and further investigations are needed. At the current moment, we concluded that vectorization optimizations are not possible for the BC7 algorithm. Nonetheless, decoding text in parallel in several threads is still a viable option that can improve decompression time 2-8 times. However, algorithms that use several threads, rather than specific single-threaded ALU instructions, are out of the scope of this work.

3.3 Results for various decoding algorithms for BC3 and BC7 codes

In this section, we compare previously proposed BC3 codes and newly developed BC7 codes. Each algorithm was compiled using Visual Studio 2022 compiler and Intel icx compiler. Between two executables produced by different compilers, one with a smaller decompression time was chosen. All tests were done on 3 computers with processors i7-7700HQ, i5-9300H, and i5-6500. The same algorithm produces the same output file regardless of the compiler and computer, therefore compressed text size is always equal. On the other hand, decompression time varies among different machines. We take the average decompression time among three computers.

Sizes of compressed texts are listed in table 5. Previously proposed code BC3 always gives a better compression ratio. BC3 compresses smaller files quite better than BC7, leading by 6-8%. For larger files, compressed sizes are closer, but BC3 still leads by 0.3-0.5%.

File name	alice29.txt	book1	bible.txt	enwik8	enwik9
BC3	**36684, 8.7%**	**199084, 7.1%**	**956580, 5%**	**23891804, 6.7%**	**235624268, 5.8%**
BC7	38932, 15,3%	207940, 11.9%	1028212, 12.8%	24011236, 7.2%	236267668, 6.1%

Table 5: Compressed sizes and overhead relative to Huffman codes for BC3 and BC7 codes.

Time measurements showed more varying results. We tested the next algorithms: simple BC7 decoding, fast BC7 decoding for 16-bit and 32-bit numbers, and fast BC3 decoding for 16-bit and 32-bit numbers. BC3 algorithms were tested with 10-bit tables and 12-bit tables, but in all cases, 12-bit tables turned out to be faster. Each algorithm was executed on each text multiple times: 10000 times in alice29.txt, 5000 times on book1 and bible.txt, 1000 times on enwik8, and 200 times on enwik9. Minimal and average decoding time was recorded. As both numbers show similar results, we only list minimal compression time in table 6. Among the results of two

compilers, we choose one with the smallest execution time. Detailed results together with codes of all algorithms are available in [10].

Algorithm	alice29.txt	book1	bible.txt	enwik8	enwik9
BC3, 32-bit	**0,0390**	**0,2135**	**1,381**	**31,62**	**322,5**
BC3, 16-bit	0,0423	0,2318	**1,294**	-	-
BC7, 32-bit	0,0411	0,2233	1,464	32,42	328,7
BC7, 16-bit	**0,0358**	**0,1924**	**1,133**	-	-
BC7, simple	0,1926	1,0401	5,966	120,12	1210,0

Table 6: execution time in milliseconds of different BC3 and BC7 decoding algorithms.

Fast decoding algorithms are 4 times faster for 32-bit numbers and 5 times faster for 16-bit numbers than simple decoding algorithms. For 32-bit numbers, BC3 and BC7 have about the same decoding speed. However, BC3 compressed file is smaller than BC7 one, so 32-bit BC3 decoding is slightly faster in practice. For some reason, 16-bit BC3 decoding is slower than 32-bit one. It is unexpected behavior and may require further investigations. On the other side, 16-bit BC7 decoding is the fastest algorithm among all considered in this section.

3.4 Comparison of our and previously proposed codes

In the same way, as in the previous section, we also tested BCMix decoding algorithms and previously proposed RMD, SCDC, and RPBC decoding algorithms.

We first summarize the results for different decoding algorithms of BCMix codes. We tested the next BCMix decoding algorithms: simple algorithm BCMix.2, two-phase algorithm BCMix4 together with algorithm BCMix.5, single codeword algorithm BCMix.6, and digit aligned algorithm BCMix.7. For algorithm BCMix.7 we tested 3 variations, as it was unclear which block length to choose. The measured

decoding time is shown in table 7. In table 7 we also list particular codes from the BCMix family that give the best compression ratio.

Algorithm	alice29.txt	book1	bible.txt	enwik8	enwik9
BCMix.2	0,2884	1,5971	8,893	176,24	1808,4
BCMix.4+BCMix.5	0,1223	0,6706	3,829	110,01	1024,6
BCMix.6	0,1018	0,5612	3,118	75,03	706,3
BCMix.7, 8 bits	**0,0931**	**0,5284**	2,773	**69,21**	**717,2**
BCMix.7, 9 bits	0,1051	0,5739	3,063	76,28	823,0
BCMix.7, 10 bits	0,1105	0,6102	**2,725**	71,30	726,8
Best code	**M4**	**M4**	**M3**	**M4233**	**M423**

Table 7: Execution time in milliseconds of different BCMix decoding algorithms.

In 4 out of 5 cases, the best time was achieved by the BCMix digit aligned algorithm with an 8-bit block. However, in the case of bible.txt, the best time was achieved with the same algorithm but with a 10-bit block. The fast decoding algorithm is 2.5-3 times faster than the simple decoding algorithm.

Finally, we compare all static codes considered in this work. Table 8 shows the compression ratio of different codes.

Code	alice29.txt	book1	bible.txt	enwik8	enwik9
BC3	36684, 8.7%	199084, 7.1%	956580, 4.9%	23891804, 6.7%	235624268, 5.8%
BC7	38932, 15.3%	207940, 11.8%	1028212, 12.8%	24011244, 7.2%	236267668, 6.1%
BCMix	**35294, 4.5%**	**192693, 3.6%**	938709, 3.0%	**22952627, 2.5%**	**226788813, 1.9%**
R2,4-inf	35504, 5.2%	194550, 4.6%	**929969, 2.0%**	24045884, 7.3%	-
R2-inf	35441, 5%	192744, 3.7%	933621, 2.7%	23118897, 3.2%	228112827, 2.5%
SCDC	39094, 15.8%	214980, 15.6%	1049003, 15.1%	25276782, 11.3%	250301493, 12.4%
RPBC	39712, 17.6%	210415, 13.2	1055691, 15.8%	25554967, 12.8%	251296235, 12.9%

Table 8: Compressed texts sizes and redundancy relative to Huffman codes for different static codes.

For R2,4-inf encoding failed as one of the large codewords get out of bound of the 32-bit variable. In 4 out of 5 cases BCMix codes provide the best compression ratio. For bible.txt the best compression ratio achieved by R2,4-inf code, is only 2% over Huffman codes, while BCMix code has 3% of redundancy over Huffman codes. In general, BCMix codes usually give a 0.5-1% better compression ratio than better of RMD codes. Such a result is fascinating because BCMix codes do not use some bit lengths, as the minimum length of the digit is 2.

We also measure decompression time for all codes. Table 9 shows the results.

Code	**alice29.txt**	**book1**	**bible.txt**	**enwik8**	**enwik9**
BC3	**0,0390**	**0,2135**	**1,294**	**31,62**	**322,5**
BC7	**0,0358**	**0,1924**	**1,133**	32,42	328,7
BCMix	**0,0931**	**0,5284**	**2,725**	**69,21**	**717,2**
R2,4-inf	0,1102	0,6257	3,322	93,38	-
R2-inf	0,1249	0,7151	3,791	103,78	1064,7
SCDC	0,0956	0,6214	3,040	80,56	777,9
RPBC	0,1105	0,6377	3,187	85,73	805,9

Table 9: Decompression time of different static codes.

A good compromise between compression ratio and decoding speed is achieved by the previously proposed BC3 code. In case of decompression speed is very critical and words ranks fit in 16 bits, BC7 code can be used. In case compression ratio is the most important, BCMix codes are the best option, as they outperform RMD, SCDC, and RPBC codes in both compression ratio and decompression speed.

SECTION 4. CONCLUSION

In this work, research on the effectiveness of codes with delimiters in natural text compression was conducted. Multiple previously proposed static entropy codes were considered. Possible modifications of the BC3 code were researched. The first modification simply changes the bit length of codeword digits. The second one produces a family of new codes called BCMix.

Different decoding algorithms were developed for newly proposed codes. BC7 decoding implementation is similar to BC3 decoding. So, only a bitstream loading logic was changed. Several decoding algorithms for BCMix were proposed. They are all quite different as they are based on ideas of RMD, Huffman, BC3 decoding. Also, a fast algorithm to find the best candidate from BCMix family was proposed.

The experiments were conducted on 5 different English texts. Each decompression algorithm was executed on each text hundreds or thousands of times. Our codes were compared with RMD, SCDC, RPBC codes.

Experimental results showed that BCMix codes outperform all other codes by 0.5-1% in 4 out of 5 cases. In the case of bible.txt file, the R2,4-inf code gave a 1% better compression ratio than BCMix codes. BC3 showed to be superior to the BC7 in compression ratio on all texts.

Experiments also showed that the best of BCMix decoding algorithms is faster than RMD, SCDC, and RPBC decoding algorithms. That is, in most cases, BCMix codes are superior to RMD, SCDC, RPBC codes in both compression ratio and decoding speed. BC3 codes turned out to be faster than BC7 codes on large files. However, BC7 is faster for small and medium-size texts.

In case compression ratio is preferred, the family of BCMix codes is the best choice. In case decompression time is critical, BC3 code is the best choice for large texts and BC7 code is well suited for small and medium-sized texts. The difference

between the decoding time of BC3 and BC7 is not large. As a compromise between compression ratio and decoding speed, BC3 still may be a viable choice.

Further improvements to BCMix codes are possible. New faster decompression algorithms may be found. Decompression algorithms proposed in this paper may also have low-level optimizations. Modification of BCMix codes, in which each digit has its own set of delimiters may further improve the compression ratio. Currently, BCMix codes are complete in general but are not complete for particular text. Modifications to the processing of the last two or three digits may improve the compression ratio. We note, that these modifications are compatible with proposed decoding algorithms. Single codeword BCMix decoding allows free codeword distribution inside bocks of 9-12 bits. How to find the best distribution inside such blocks is also an interesting topic for further research.

REFERENCES

[1]. The Canterbury Corpus [Electronic resource]: https://corpus.canterbury.ac.nz/

[2]. Fabian Giesen. “Entropy decoding in Oodle Data: Huffman decoding on the Jaguar” [Electronix resourse] 2022.04.04, https://fgiesen.wordpress.com/2022/04/04/entropy-decoding-in-oodle-data-huffman-decoding-on-the-jaguar/

[3]. Colt McAnlis, Aleks Haecky. “Understanding compression: data compression for modern developers”, 2016

[4]. enwik8 Dataset [Electronic resource]: https://deepai.org/dataset/enwik8

[5]. enwik9 Dataset [Electronic resource] : https://archive.org/details/enwik9

[6]. Enno Ohlebuschmy. “Lempel-Ziv Factorization: LZ77 without Window”, (May 13, 2016)

[7]. Maksym Kovalchuk, “Match finder comparison in LZ-like compressor” [Electronoc resource] : https://encode.su/threads/3579-New-saca-and-bwt-library-(libsais)?p=74399&viewfull=1#post74399

[8]. Maksym Kovalchuk, “LZMA Markup” [Electronoc resource] : https://github.com/reeWorlds/LZMAMarkup

[9]. Zavadskyi, Igor & Anisimov, A.. (2020). Reverse Multi-Delimiter Compression Codes. 173-182. 10.1109/DCC47342.2020.00025.

[10]. Maksym Kovalchuk, “BCMix codes” [Electronic resource] : https://github.com/reeWorlds/BCMix_codes

[11]. Yann Collet, “New Generation Entropy coders” [Electronic resource] : https://github.com/Cyan4973/FiniteStateEntropy

[12]. Nong, Ge & Zhang, Sen & Chan, Daricks Wai Hong. (2009). Linear Suffix Array Construction by Almost Pure Induced-Sorting. Proceedings of the Data Compression Conference. 193-202. 10.1109/DCC.2009.42.

[13]. Matt Mahoney. “Large Text Compression Benchmark” [Electronic resource] : http://mattmahoney.net/dc/text.html

[14]. Colin Percival. “Naive Differences of Executable Code”. Computing Lab, Oxford University

[15]. Internet Archive “WaybackMachine” [Electronic resource] : https://web.archive.org/

[16]. Yuta Mori. “libdivsufsort – implementation of lightweight suffix array construction algorithm” [Electronic resource] : https://github.com/y-256/libdivsufsort

[17]. Ilya Grebnov. “libsais” [Electronic resource] : https://github.com/IlyaGrebnov/libsais

[18]. Agner Fog. “Vector Class Library” [Electronic resource] : https://github.com/vectorclass/version2

[19]. Frieder, G., Luk, C.: Algorithms for binary coded balanced and ordinary ternary operations. IEEE Trans. on Computers 23(2), 212–215 (1975)

[20]. Zavadskyi, Igor. (2022). Binary-Coded Ternary Number Representation in Natural Language Text Compression.

[21]. Package-merge algorithm [Electronic resource] : Wikipedia, https://en.wikipedia.org/wiki/Package-merge_algorithm

[22]. Brisaboa, Nieves & Fariña, Antonio & Navarro, Gonzalo & Paramá, José. (2007). Lightweight natural language text compression. Inf. Retr.. 10. 1-33. 10.1007/s10791-006-9001-9.

[23]. Culpepper, J.S., Moffat, A. (2005). Enhanced Byte Codes with Restricted Prefix Properties. In: Consens, M., Navarro, G. (eds) String Processing and Information Retrieval. SPIRE 2005. Lecture Notes in Computer Science, vol 3772. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11575832_1