Binary data compression using combinations of entropy and dictionary encoding algorithms*

Oleksii Turuta†, Nataliia Golian†, Vira Golian†, Iryna Afanasieva†, Kostiantyn Onyshchenko† and Sviatoslav Mychka*†

Kharkiv National University of Radio Electronics, 14 Nauky Ave., Kharkiv, 61166, Ukraine

Abstract

Text data compression methods are explored. The advantages and disadvantages of different combinations of arithmetic and Huffman encoding as entropy methods and Snappy and LZ4 as dictionary methods are described. Methods of increasing efficiency of entropy compression methods are suggested.

Keywords

Text compression, encoding, algorithm, arithmetic coding, Huffman optimal coding, Snappy, LZ4 1

1. Introduction

Today, big data is stored in various distributed databases and file systems. They allow storing different files on different servers with the ability of restoration using duplicates [1]. According to [2], big data is the term that does not mean a specific amount of data, but rather its relationality to other data. This is the reason for storing the data in such database formats as Parquet, which is supported by Apache Hadoop and supports different compression algorithms, such as LZ4 or Snappy.

Binary data compression plays a crucial role in distributed systems by reducing storage costs and improving processing efficiency. Unlike specific compression algorithms, the algorithms designed to work with binary data treat any information as a sequence of bits or bytes. This means any data may be encoded and then decoded without errors or losses, which is useful for the file systems and databases that can store different types of files and data.

Two most popular entropy encoders, which are Huffman optimal encoding and arithmetic encoding are widely used in methods like Deflate, Bzip2 and other. Usually they are used in combination with dictionary encoders, which include LZ4 and Snappy, mentioned above, and also other algorithms with similar idea. These algorithms are designed to avoid entropy coding, but this paper investigates how would these algorithms work if combined with different types of data.

2. Related Works

Data compression algorithms are divided into two groups: lossy and lossless. According to [3], feature-based data compression concept is becoming more popular nowadays. This concept unifies lossless, near-lossless and lossy compression with the idea of features – pieces of information that

ISW-2025: Intelligent Systems Workshop at 9th International Conference on Computational Linguistics and Intelligent Systems (CoLInS-2025), May 15–16, 2025, Kharkiv, Ukraine

* Corresponding author.

† These authors contributed equally.

oleksii.turuta@nure.ua (O. Turuta); nataliia.golian@nure.ua (N. Golian); vira.golan@nure.ua (V. Golian); iryna.afanasieva@nure.ua (I. Afanasieva); kostiantyn.onyshchenko@nure.ua (K. Onyshchenko); sviatoslav.mychka@nure.ua (S. Mychka)

0000-0002-0970-8617 (O. Turuta); 0000-0002-1390-3116 (N. Golian); 0000-0002-7196-5286 (V. Golian); 0000-0003-4061-0332 (I. Afanasieva); 0000-0002-7746-4570 (K. Onyshchenko); 0009-0005-0292-6522 (S. Mychka)

possess high discriminative or predictive value for the human interpretation or machine processing.

In this paper specific lossless compression algorithms are investigated. They can be divided into entropy and dictionary types, depending on which approach to data compression they use. Entropy encoders usually work with data on the bit level, attempting to reduce the number of bits needed to encode a message, while the dictionary ones usually work with bytes, writing them into a dictionary-like structure to reference parts of the text later. The dictionary encoders are often called the "LZ family" due to similarity to LZ77, the first algorithm that employed this approach [4].

Data compression algorithms are qualified by compression ratio, compression speed and decompression speed. In their turn, these metrics depend on the data's size, the message's mean entropy and, in the case of entropy algorithms, the size of the alphabet used in the message. The mean entropy of the message is calculated using Shannon entropy [5, 6] (1).

$$H(X) = -\Sigma p(x)\log_2 p(x), \quad (1)$$

where p(x) – frequency of occurrence of symbol x.

An alphabet is a set of symbols used in a message [7]. In the case of byte-to-byte encoding of binary data, the alphabet always contains from 1 to 256 symbols since bytes take values from 0 to 255, and messages cannot contain zero symbols. Static alphabets can be used to create models for entropy compression algorithms, e.g., adaptive arithmetic coding.

Shannon's theorem implies that N i.i.d. random variables each with entropy H(X) can be compressed into more than N H(X) bits with negligible risk of information loss, as $N \longrightarrow \infty$; conversely if they are compressed into fewer than N H(X) bits it is virtually certain that information will be lost. Assuming that symbols in the alphabet of a binary message are not distributed independently and identically, there will be symbols that carry more information, if their occurrence frequency is lower and vice versa. This enables lossless compression algorithms, which reduce this redundancy and assign new codes to the symbols, according to the amount of information they carry. If the symbols are distributed identically and do not carry natural redundancy, i.e. the alphabet includes symbols from 0 to maximum possible value, which means the message is completely random, then such message cannot be compressed.

Data compression methods may utilize both dictionary and entropy coding [8], or only one of them [9, 10]. It is important that, if both methods are applied, entropy coding follows dictionary coding, since entropy encoding algorithms tend to make the entropy of the data rise, so the data become random and lose all the patterns that may be detected by dictionary coding [11].

The purpose of this paper is to investigate how the application of entropy encoding after already applied dictionary encoding affects the compression ratio and the time of compression and decompression. The tasks to be solved are implementation of the compressors and testing the obtained algorithms with different data types that can be stored in regular or distributed file systems. One of the materials for testing compression algorithms is Silesia compression corpus [12], which is a group of various binary, text, image, executable or other files. Some files from this group will be used for testing different approaches in this paper, too.

3. Methods

In this work four compression methods will be used for the research and experiments. Two of them belong to entropy methods, and two others – to dictionary methods. To demonstrate the algorithms we will use a simple text: "*The cat sat on the mat, and the cat lied on that hat*". It contains repetitions of symbols and consists of only 15 different symbols, which makes a fairly small alphabet relative to that containing all the symbols of binary data.

3.1 Huffman coding

Huffman coding is a data compression method designed to ensure optimal encoding for each separate symbol [13]. Developed based on Shannon-Fano method[14], it consists of three main steps:

1. During the first step the alphabet of the whole message is formed, and the frequencies of all the symbols are calculated. These frequencies are then used to construct a Huffman tree, where symbols with lower frequencies appear deeper in the tree, and those with higher frequencies are closer to the root. Figure 1 depicts the Huffman tree built from the sample text mentioned before.

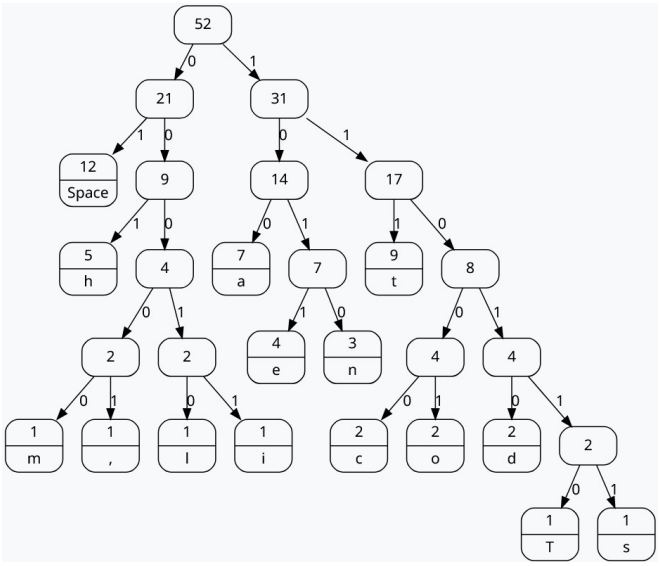

Figure 1: Huffman tree for the sample text

2. After building the tree the codes are assigned to their corresponding symbols using an associative array or other table structure. This allows to access each symbol's code in O(1) time, rather than looking it up in $O(\log_2 n)$ time for each symbol in the message.

By traversing the Huffman tree, we assign binary codes to each symbol. The tree is traversed breadth-first from the root recursively, each time adding a zero to the code when going to the left child, and a one going to the right child. It means that the codes are shorter for the symbols that occur more and longer for those that occur less. This also ensures that no code is a prefix of another, allowing for unambiguous determination of each symbol during decoding, making it more efficient. The codes for each symbol of the sample text are given in Table 1.

Table 1
Binary codes of the symbols in example text

Symbol	Code
Space	01
h	001
a	100
t	111
e	1011
n	1010
m	00000
,	00001

l	00010
i	00011
c	11000
o	11001
d	11010
T	111110
s	111111

3. Finally, the text is encoded using the stored codes. It is worth noting that Huffman and other entropy encoding algorithms operate with bits, and usually the output bit sequences are aligned so their count is a multiple of eight. The encoded example text will result into 20 bytes instead of initial 52 bytes, which means compression ratio is 2.6.

To decode the message encoded with the Huffman algorithm, we need to build the tree used in encoding and read the message bit by bit, traversing the obtained tree. When a symbol node is found, the decoder writes the symbol into the output and moves the pointer to the root again. This process is repeated until the last bit of the message is read.

The main bottleneck of Huffman compression is that it is computationally hard to make it adaptive. This means the output must contain the data needed to build the same Huffman tree used during compression. Some methods, such as the Vitter algorithm, allow adaptive Huffman encoding [15]. However, it is rarely used in practice since it requires much more computations for node swapping [16]. Herewith, each encoded block's maximum storage overhead can be calculated with (2).

$$L + (max(byte) + 1) \cdot (c + 1) = 1 + 256 \cdot 5 = 1281 bytes, \quad (2)$$

where L – number of bytes to store the length of the alphabet, *max(byte)* – maximum value of byte (255), c – number of bytes to store the number of appearances of a symbol.

These calculations apply to the corner case when all 256 values of a byte are used in the document. Still, using blocks of significant size, e.g., 100 KB or more, reduces the impact of the header's size.

3.2 Arithmetic coding

Arithmetic encoding is another data compression method, which encodes messages using an interval [0; 1) split into sub-intervals representing each symbol's frequency. The result of arithmetic coding is a number from 0 to 1, which represents the whole message, unlike Huffman coding that represents separate symbols. With infinite precision messages of any lengths can potentially be encoded into single numbers [17].

The process of a non-adaptive arithmetic encoding, like the Huffman encoding, starts with collecting the information of the alphabet used in the message and the frequencies of symbols. After this step the interval [0; 1) is split into sub-intervals with widths according to the relative frequency of each symbol. Figure 2 shows the graphical representation of the interval division for the sample text.

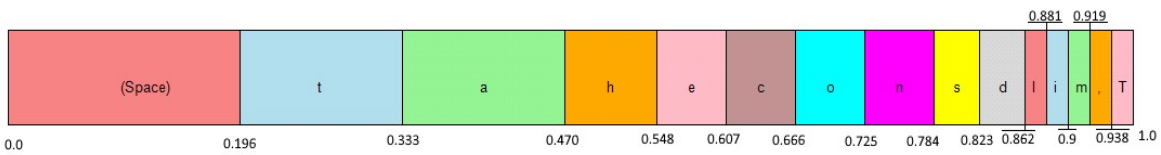

Figure 2: Representation of the sub-intervals for sample string

After this we process each symbol, refining the interval. At first we select the whole interval [0; 1) and select a number from the interval that represents the symbol. For example, after processing the letter "T" we move to [0.938; 1) and select it as the new interval. This new interval is split again, and after processing the letter "h" we get [0.945; 0.948) and so on. After encoding each symbol we reach the final interval [0.946063123456; 0.946063123478) and pick up a number inside these boundaries. For example, we'll pick a 0.946063123460. This number encodes the whole sample text and can be stored in only 4 bytes, which means the compression ratio is 13.

To decode the message we need to know the intervals for the symbols. Using them we can find the letters one by one, since the result number will always fall into the right intervals. For example, the number mentioned above fits into [0.938; 1.0) interval, which stands for "T", then into [0.945; 0.948), which stands for "h", and so on.

Unlike Huffman encoding, arithmetic encoding is more convenient to enabling adaptive variant of encoding. This allows to avoid passing the data about symbols' frequencies and does not require significant complications. For example, the adaptive algorithm has two major differences from the algorithm described above:

1. At first we assume that each symbol is present in the message's alphabet with equal frequency. It means that we initialize the initial interval with equal sub-intervals of length (3)

$$\frac{1}{max(byte)+1} = \frac{1}{256} \approx 0.0039, \quad (3)$$

where *max(byte)* – the maximum value of a byte.

2. With each encoded symbol, we make its sub-interval wider, assuming that with each occurrence, the probability of this symbol appearing in the same message gets bigger. Herewith, the frequencies of other symbols decrease proportionally to give a total of 1.

These amendments allow a simple decoding process: build the same interval with equal probabilities sub-intervals and update them with each decoded symbol. Obviously, this doesn't require any data about frequencies of symbols transferred with the encoded data.

3.3 LZ4

The LZ4 algorithm is an open-source data compression algorithm. It is designed to be fast, trading CPU time for compression ratio. The LZ4 algorithm does not include entropy encoding such as Huffman or arithmetic coding, and utilizes the dictionary approach similar to the LZ77 algorithm.

An LZ4 block contains of sequences, which are suites of literals (not-compressed bytes), followed by match copy operations. To encode a message, LZ4 scans input data for repeated sequences and replaces them with references to previous occurrences. Literals are directly copied without transformation, avoiding additional encoding overhead, their lengths are encoded with 4 bits before them. Match copies follow the literals, with 2 bytes meaning offset value, and the length of the match encoded with 4 bits after literal length. A schematic image of an LZ4 block is in Figure 3.

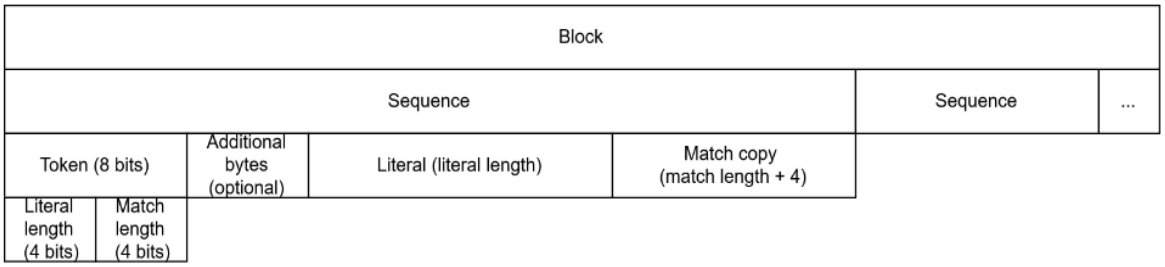

Figure 3: LZ4 block format scheme

To achieve fast compression and decompression – 577 MB/s and 3716 MB/s respectively for lz4 1.10.0, according to lzbench [18] – LZ4 minimizes cache misses by avoiding branching, where the simple block format is helpful. Also, this algorithm utilizes hash tables for storing previously seen sequences of symbols, which enables looking up for matches in O(1) time. The decompression algorithm avoids complex calculations, copying information from literals. Unlike LZ77, the match copies do not allow cyclic copying from the literals.

The encoding of the sample text "The cat sat on the mat, and the cat lied on that hat" would be performed in these three steps:

1. Finding the first literal: "The cat sat on the mat, and", and encoding its length 27. This length is encoded with 1 additional byte after token.
2. Encoding the match copy after the literal and its length in 4 lower bits of the token: "[space]the[space]" with length 5 and offset 13. It is encoded as 1 for the match length.
3. Encoding another literal: "cat lied on that hat" with length 20 with no match copies afterwards.

After the process of encoding we receive 53 bytes, which means that encoding did not result in a compression. This is the result of the small size of the message, and also in some missed matches: the words "cat", "hat" and "on" with surrounding space characters have not been encoded. The reason is that LZ4 block format has limitations: the minimum match length is 4, the last 5 bytes are always literals, and the last match must start at least 12 bytes before the end of block [9].

3.4 Snappy

Snappy is another open-source compression algorithm utilizing dictionary coding. Like LZ4, it was designed for faster encoding and decoding, so it originally does not include entropy coding and is oriented to bytes.

The stream of data encoded with Snappy starts with a preamble, which contains the uncompressed length, stored in a special *varint* type. Varint is a type where for each byte there are seven valuable bits and the highest bit is set if there are more bytes to be read. After this preamble there come elements of four different types. Every element starts with a tag byte, where two lower bits indicate the type and six higher bits – the length of the element[10].

1. Literals (00). Uncompressed data, stored directly after the tag byte.
2. Copies (01-11). References to the previous decompressed data, including literals. They consist of *offset*, which sets the relative position back in the decoded stream, and *length*, which means how many symbols should be copied. Unlike LZ4, Snappy allows lengths bigger than offsets, which means copying the symbols in cycles. The offsets for different copy types are stored in one, two or four bytes.

The block format for Snappy is similar to LZ4 (Figure 4), though it has less limitations for match length and the encoding of literals than LZ4. It also works with 32 KB blocks, but the format does not specify it directly.

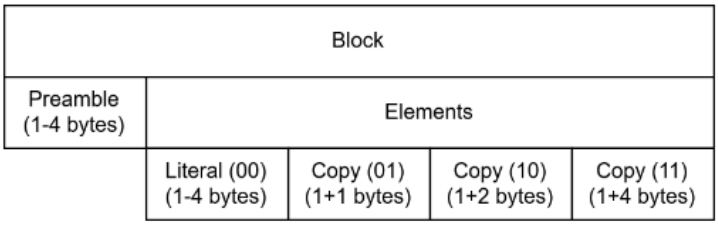

Figure 4: Snappy block format

As it comes from the diagram above, the block format for Snappy is simpler than LZ4, which potentially gives higher compression ratios due to optimized metadata, but it requires extra calculations and additional branching, which slows compression and decompression down. To encode the sample message with Snappy algorithm, the encoder would follow these steps:

1. Encode the uncompressed length in the preamble with one byte *0x34*.
2. Encode the first literal: “The cat sat on the mat, and” with length 27, encoded with one byte.
3. Encode the first match: “[space]the[space]” with one byte for offset and one byte for length.
4. Encode another match: “cat[space]”, also with one byte for offset and for length.
5. Encode literal “lied” with one byte encoding length.
6. Encode match “[space]on th” with one byte for offset and for length.
7. Encode the final literal “at hat” with one byte for length.

After encoding we will get 43 bytes against original 52, which gives the encoding ratio of 1.21 even with relatively short text.

4. Experiment

To investigate the perspectives of applying entropy encoding algorithms after the dictionary algorithms described above, we have implemented optimal Huffman coding and arithmetic coding using C# 12 and .NET 8. We also developed a structure for handling bit storing and operations:

```
  public struct CountedBitsArray : ICloneable
  {
      /// <summary>
     /// Count of bits in the last byte in the array. Bits are counted in BE
(less significant bit after more significant)
      /// </summary>
      private byte bitCount = 0;
      private List<byte> bytes = [];
      public CountedBitsArray()
      {
          bytes.Add(0);
      }
      public CountedBitsArray(Span<byte> storedBytes)
      {
          bitCount = storedBytes[0];
          bytes.AddRange(storedBytes[1..]);
      }
   }
```

This structure is kept simple to avoid redundant operations and branching. It has several convenience methods for adding a bit, checking a bit on a specific position or copying another BitArray.

```
  /// <summary>
  /// Add bit to the last byte in <paramref name="bytes"/>
  /// </summary>
  public void AddBit(bool value)
  {
      if (bitCount == 8)
      {
          bytes.Add(0);
          bitCount = 0;
      }
      if (value) // no need to assign 0 since it is already there
      {
          var mask = (byte)(1 << bitCount);
```

```
            bytes[^1] |= mask;
        }
        bitCount++;
    }
    /// <summary>
    /// Get value of a bit in the specified position, starting from the
beginning of <paramref name="bytes"/>
    /// </summary>
    public readonly bool IsBitSet(int pos)
    {
        // get the required byte
        var byteNum = pos / 8;
        if (byteNum >= bytes.Count) return false;
        // get the required bit in byte
        var bitNum = pos % 8;
        // use bit mask
        return (bytes[byteNum] & 1 << (bitNum)) != 0;
    }
    public void AddBits(int count, bool value)
    {
        for (int i = 0; i < count; i++)
        {
            AddBit(value);
        }
    }
    public void AppendBitArray(CountedBitsArray bitsArrayToAppend)
    {
        for (int i = 0; i < bitsArrayToAppend.BitsCount; i++)
        {
            AddBit(bitsArrayToAppend.IsBitSet(i));
        }
    }
```

The arithmetic encoder was implemented as an integer range coder, which works similarly but allows the use of finite-precision data types . To implement such behavior, specific numbers that play roles of a “whole”, “half” and “quarter” are introduced, such as:

$$half = \frac{whole}{2}, quarter = \frac{whole}{4}, R \cdot whole \leq max(\text{int64}) \tag{4}$$

where R – needed level of precision, max(*int64)* – maximum value of a 64-bit integer.

This gives an oppurtunity to simplify the algorithm and use only integer values without the need to perform costly division operations. Adaptive arithmetic coding becomes much faster and easier to implement. Below a part of encoding function is presented:

```
    var symbolCount = Models[modelNum].Length;
    var symbolFreq = new int[symbolCount];
    symbolFreq.Populate(1); // every symbol's frequency is initialized as 1
    var symbolFreqAdded = new int[symbolCount];
    symbolFreqAdded.SumUp(symbolFreq, 0);
    var totalFreq = symbolFreq.Sum();
    var lowerBound = 0L;
    var upperBound = Whole;
    var splits = 0;
    var result = new CountedBitsArray();
    foreach (var s in data)
    {
        EncodeSymbol(symbolFreq, symbolFreqAdded, ref totalFreq, ref lowerBound,
ref upperBound, ref splits, ref result, Array.BinarySearch(Models[modelNum],
s));
    }
    EncodeSymbol(symbolFreq, symbolFreqAdded, ref totalFreq, ref lowerBound, ref
upperBound, ref splits, ref result, symbolCount - 1);
    ++splits;
```

```
if (lowerBound <= Quarter)
{
    result.AddBit(false);
    result.AddBits(splits, true);
}
else
{
    result.AddBit(true);
    result.AddBits(splits, false);
}
```

The Snappy and LZ4 algorithms are used as 'black boxes" – their default implementations are adopted "as is" and remain unchanged. To launch and direct the algorithms, the default command line interfaces (CLI) are used, if present. If a CLI is absent, third-party CLI can be used, which does not affect the compression or decompression and serves for the convenience of utilizing the algorithms.

To test the algorithms, files from Silesia compression corpus were used [12]. Their descriptions are presented in Table 2.

Table 2
Descriptions of the files from Silesia compression corpus used for tests

Name	Type	Description	Size, bytes
dickens	English text	Collected works of Charles Dickens	10,192,446
mozilla	exe	Tarred executables of Mozilla 1.0 (Tru64 UNIX edition)	51,220,480
mr	picture	Medical magnetic resonanse image	9,970,564
nci	database	Chemical database of structures	33,553,445
sao	binary data	The SAO star catalog	7,251,944
xml	html	Collected XML files	5,345,280

The experiments are performed on a 12-core Intel Core i7-13620H, on 64-bit Windows 10.

5. Results

First, the files were compressed with LZ4 and Snappy without further application of Huffman and arithmetic coding to measure their performance without any external influence. LZ4 was used with different compression levels that tend to increase CPU time but also increase compression ratios. The results of compression are presented in Table 3.

Table 3
Ratio and approximate time of compression of the test files with LZ4 and Snappy

Program	dickens	mozilla	mr	nci	sao	xml
lz4 -1	1.58, 200 ms	1.93, 400 ms	1.83, 180 ms	6.06, 270 ms	1.06, 210 ms	4.35, 240 ms
lz4 -2	1.9, 330 ms	2.11, 570 ms	2.09, 270 ms	6.71, 290 ms	1.17, 390 ms	5.07, 180 ms
lz4 -4	2.2, 500 ms	2.28, 590 ms	2.22, 430 ms	8.29, 450 ms	1.25, 430 ms	6.54, 210 ms
lz4 -6	2.27, 740 ms	2.3, 770 ms	2.31, 820 ms	8.87, 640 ms	1.26, 520 ms	6.83, 150 ms
lz4 -8	2.29, 1 s	2.32, 860 ms	2.34, 1.53 s	9.07, 1.15 s	1.26, 570 ms	6.92, 390 ms
lz4 -9	2.29, 1.06 s	2.32, 1.14 s	2.35, 1.82 s	9.11, 1.21 s	1.26, 570 ms	6.93, 370 ms
snappy	1.61, 130 ms	1.91, 120 ms	1.83, 40 ms	5.44, 70 ms	1.008, 40 ms	4.10, 20 ms

It's worth noting that LZ4 and Snappy CLI don't have built-in measurement of time spent for compression or decompression, and third-party time measurement was used. It means that the time

periods mentoned in Table 2 are approximate and serve only for observing the relative time period changes.

The files were also compressed using the implemented Huffman and arithmetic coding algorithms with different block sizes. The results are given in Table 4.

Table 4
Ratio and approximate time of compression of the test files with Huffman and arithmetic coding

Program, block size	dickens	mozilla	mr	nci	sao	xml
Huffman, 32 KB	1.73, 419 ms	1.38, 814 ms	2.12, 470 ms	3.2, 344 ms	1.03, 245 ms	1.59, 138 ms
Arithmetic32 KB	1.76, 401 ms	1.45, 2.5 s	2.28, 451 ms	3.27, 637 ms	1.08, 357 ms	1.62, 325 ms
Huffman, 64 KB	1.74, 303 ms	1.39, 592 ms	2.17, 268 ms	3.24, 122 ms	1.05, 607 ms	1.6, 157 ms
Arithmetic, 64 KB	1.77, 455 ms	1.43, 2 s	2.27, 276 ms	3.28, 454 ms	1.08, 249 ms	1.62, 115 ms
Huffman, 200 KB	1.75, 786 ms	1.37, 539 ms	2.18, 56 ms	3.27, 101 ms	1.06, 70 ms	1.59, 29 ms
Arithmetic,200KB	1.77, 358 ms	1.38, 1.9 s	2.23, 213 ms	3.29, 400 ms	1.07, 233 ms	1.6, 138 ms
Huffman, 500 KB	1.75, 452 ms	1.34, 515 ms	2.16, 45 ms	3.28, 95 ms	1.07, 51 ms	1.57, 40 ms
Arithmetic, 500 KB	1.77, 500 ms	1.35, 1.9 s	2.18, 288 ms	3.29, 392 ms	1.07, 190 ms	1.58, 154 ms

After these tests the files were compressed in a chain: first with LZ4 or Snappy and then with Huffman or arithmetic coding. The mean compression ratios and time of encoding and decoding each file are given in Tables 5-7.

Table 5
Mean ratio of compression of the files with two applied algorithms

Methods	dickens	mozilla	mr	nci	sao	xml
LZ4, Huffman	2.29	2.47	2.57	9.19	1.25	6.7
LZ4, Arithmetic	2.34	2.52	2.63	9.39	1.27	6.85
Snappy, Huffman	1.82	2.16	2.22	6.2	1.06	4.42
Snappy, Arithmetic	1.84	2.2	2.27	6.34	1.08	4.52

Table 6
Mean time of compression of the files with two applied algorithms, ms

Methods	dickens	mozilla	mr	nci	sao	xml
LZ4, Huffman	563	818	821	623	398	156
LZ4, Arithmetic	819	1295	904	702	492	304
Snappy, Huffman	581	960	794	584	517	240
Snappy, Arithmetic	814	1388	955	744	589	223

Table 7
Mean time of decompression of the files with two applied algorithms, ms

Methods	dickens	mozilla	mr	nci	sao	xml
LZ4, Huffman	52	201	44	38	52	15
LZ4, Arithmetic	184	770	172	160	202	79
Snappy, Huffman	56	199	47	46	62	26
Snappy, Arithmetic	226	848	202	215	242	98

To compare the performance, the compressed files were also decompressed without the use of entropy encoders (Table 8).

Table 8
Approximate time of decompression of the test files with LZ4 and Snappy, ms

Program	dickens	mozilla	mr	nci	sao	xml
lz4 -1	130	470	120	320	100	150
lz4 -2	160	450	130	330	100	130
lz4 -4	140	400	120	240	120	120
lz4 -6	130	460	150	290	100	150
lz4 -8	100	460	120	300	100	170
lz4 -9	130	520	130	300	130	160
snappy	130	80	90	50	50	60

6. Discussions

From Tables 3-5 we can see that the usage of entropy coding after the dictionary coding allowed to increase the mean compression rates in range from 7 to 17% for different files. The compression efficiency depend on the types of files, and binary files without a structure, like “sao” in the test corpus, are compressed with lower ratios than binary files that contain a structure, like “nci”, which gave the highest compression ratios with both entropy and dictionary coders. XML file, named “xml”, was a good target for LZ4 and Snappy, but Huffman and arithmetic encoding gave compression ratios almost as low as for the executable and plain text files (“mozilla” and “dickens” respectively). The entropy encoding algorithms were slightly better with an image “mr”, but the best file to be encoded with entropy encoding was “nci”, too.

The tests of LZ4 implementations showed that in most cases using compression levels more than 4 was not justified, as it made compression slightly slower but did not give much higher ratios. Usually LZ4 worked slower than Snappy, but the compression ratios Snappy gave exceeded only the first level of LZ4 in simple text files. But using entropy coding after Snappy in most cases raised the ratios more than if used after LZ4 of any level.

In most cases arithmetic coding resulted in higher compression ratios than Huffman coding, but the difference was rarely significant. The biggest difference between the ratios of these two compression methods was observed during the compression of “mr” file with block size of 32 KB, with value of 0.16. Also, arithmetic coding appears to be much slower, and it can be observed the best with bigger binary files of higher entropy. The encoding speed of arithmetic compressor is affected more from the size of block, too, and using bigger blocks is more expedient with bigger files.

The decompression speed of LZ4 and Snappy depends more on the size of a file, and does not depend on the compression level or type of the file. The decompression speed in case of using dictionary and entropy coding was affected by entropy coding, which tends to be slower than the dictionary one. The arithmetic decoding is also slower than Huffman decoding, like in the case of encoding.

7. Conclusions

The dictionary encoders, widely used in distributed platforms, can give satisfactory compression ratios with structured data, performing operations at high speed. The results obtained show that using relatively slow Huffman or arithmetic encoding after fast dictionary encoders can increase

compression ratios, but require more time for compression and decompression. Combining dictionary and entropy encoders allows to mitigate the impact of high entropy of binary files.

The topic is valuable for further development of various distributed platforms, databases and other information storage structures. There are significant results shown that the algorithms' combinations are good for compressing structured databases, but also they can be useful in advancing compression ratios in the conditions of limited storage size or Internet traffic. In the future, we plan to develop and compare other compression algorithms, including the ones that use machine learning: PPM, context mixing [19, 20] etc. The models from the implemented arithmetic coding could be expanded and improved with use of these techniques.

The work has practical value for intelligent systems, as it provides a foundation for integrating data compression strategies into the pipeline of intelligent data processing. Modern intelligent systems, including recommender systems, distributed knowledge bases, and AI services, heavily rely on the storage and transmission of large volumes of structured data. The paper proposes a compression strategy that can optimize resource usage under bandwidth or memory constraints.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

1. M. S. Farooq, Z. Kalim, J. N. Qureshi, S. Rasheed, A. Abid, A Blockchain-Based Framework for Distributed Agile Software Development, IEEE Access 10 (2022) 17977-17995, doi:10.1109/ACCESS.2022.3146953
2. D. Boyd, K. Crawford, Six Provocations for Big Data, in: A Decade in Internet Time: Symposium on the Dynamics of the Internet and Society, 2011, doi:10.2139/ssrn.1926431
3. D. Podgorelec, D. Strnad, I. Kolingerová, B. Žalik, State-of-the-Art Trends in Data Compression: COMPROMISE Case Study, Entropy 26 (2024). doi:10.3390/e26121032
4. D. Kempa, T. Kociumaka, Lempel-Ziv (LZ77) Factorization in Sublinear Time, in: 2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS), Chicago, IL, USA, 2024, pp. 2045-2055, doi:10.1109/FOCS61266.2024.00122.
5. J.A. Bergstra, J.V. Tucker, On Defining Expressions for Entropy and Cross-Entropy: The Entropic Transreals and Their Fracterm Calculus, Entropy 27 (2025). doi:10.3390/e27010031
6. H. Alizadeh Noughabi and M. Shafaei Noughabi, A New Estimator for Shannon Entropy, Statistics, Optimization & Information Computing 13 (2025) 891-899, doi:10.19139/soic-2310-5070-1844.
7. A. Somazzi, P. Ferragina and D. Garlaschelli, On Nonlinear Compression Costs: When Shannon Meets Rényi, IEEE Access 12 (2024) 77750-77763, doi:10.1109/ACCESS.2024.3406912
8. H. K. Tayyeh, A. S. Ahmed AL-Jumaili, A combination of least significant bit and deflate compression for image steganography. International Journal of Electrical and Computer Engineering (IJECE) 12 (2022), 358, doi:10.11591/ijece.v12i1.pp358-364
9. Y. Collet, LZ4 Block Format Description, 2022. URL: https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md.
10. V. Costan, Snappy Format Description, 2011. URL: https://github.com/google/snappy/blob/main/format_description.txt
11. S.T. Klein, D. Shapira, On the Randomness of Compressed Data, Information 11 (2020) 196. doi:10.3390/info11040196
12. S. Deorowicz, Silesia compression corpus, 2003. URL: https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia
13. D. Huffman, A Method for the Construction of Minimum Redundancy Codes, Proceedings of the IRE 40(9) (1952) 1098-1101. doi:10.1109/JRPROC.1952.273898.

14. S. Congero, K. Zeger, Competitive Advantage of Huffman and Shannon-Fano Codes, IEEE Transactions on Information Theory (2024). doi:10.1109/tit.2024.3417010.
15. J. S. Vitter, Design and analysis of dynamic Huffman codes, Journal of the ACM 34 (1987) 825–845. doi:10.1145/31846.42227
16. S. Song, T. Lian, W. Liu, Z. Zhang, M. Luo, A. Wu, A lossless compression method for logging data while drilling, Systems Science & Control Engineering 9(1) (2021) 689–703, doi:10.1080/21642583.2021.1981478
17. F. Auli-Llinas, Fast and Efficient Entropy Coding Architectures for Massive Data Compression, Technologies 11 (2023) 132, doi:10.3390/technologies11050132
18. Przemyslaw Skibinski, lzbench, 2025. URL: https://github.com/inikep/lzbench?tab=readme-ov-file#benchmarks
19. R. M. K. Mohideen, P. Peter, J. Weickert, A systematic evaluation of coding strategies for sparse binary images, Signal Processing: Image Communication 99 (2021) 116424, doi:10.1016/j.image.2021.116424
20. L. S. Lopes, P. A. Chou, R. L. de Queiro