КОМП'ЮТЕРНІ НАУКИ ТА ІНФОРМАТИКА

UDC 51.681.3
DOI: https://doi.org/10.17721/1812-5409.2025/2.25

Igor ZAVADSKYI, DSc (Phys & Math), Prof.
ORCID ID: 0000-0002-4826-5265
e-mail: ihorzavadskyi@knu.ua
Taras Shevchenko National University of Kyiv, Kyiv, Ukraine

Maksym KOVALCHUK, PhD Student
ORCID ID: 0009-0002-8450-905X
e-mail: max.koval4uk@knu.ua
Taras Shevchenko National University of Kyiv, Kyiv, Ukraine

OPTIMIZATION OF THE CANONICAL HUFFMAN CODES FOR WORD-BASED NATURAL LANGUAGE TEXT COMPRESSION

Canonical Huffman codes are widely used in data compression and allow fast decoding without compromising compression efficiency compared to classical Huffman codes. If the maximal length of a codeword does not exceed 11-12 bits or long codewords are very rare, any variable-length code, including Huffman codes, can be effectively decoded using a naive 'table' method. Canonical Huffman codes should be used primarily if the encoded text consists of a relatively large number of long codewords. For example, this is the case of word-level natural language text compression, where words are the symbols of an alphabet. A. Moffat and A. Turpin proposed the fastest state-of-the-art algorithm for decoding canonical codes. We offer several optimizations of this algorithm, which, according to our experiments, can speed it up by 20-30% without reducing the compression ratio. Some of these optimizations are based on low-level code parallelization using the SIMD architecture of modern processors, while others propose combining classical decoding of canonical codes with a universal 'table' decoding method. In addition to the above optimizations, the article discusses in detail the canonical Huffman codes, including the basic and optimized algorithms, as well as the universal 'table' method for decoding arbitrary variable-length codes. The results of an experimental comparison of the performance of the basic and optimized algorithms are presented. The results of the study can be used in information retrieval and data archiving systems.

K e y w o r d s: ***Huffman, encoding, codes, canonical, text compression.***

AMS 2020 classification: 68P30, 94A29, 94-06.

Introduction

Relevance of research. Huffman codes (Huffman, 1952) are a keystone of lossless data compression. In semi-static statistical compression, for a given data source, they provide an optimal compression ratio if symbol probabilities are negative powers of 2 and a near-optimal ratio in many practical use cases. However, the default decoding algorithm for Huffman codes requires bit-by-bit processing of the encoded text with the corresponding traversal of a Huffman tree. This approach suffers from low decoding speed, which is a crucial characteristic in many practical data compression scenarios. In the last two decades, many other codes that trade the decoding speed for compression ratio have been proposed, e.g., Byte-Aligned codes (Brisaboa et al., 2003), Reverse Multi-Delimiter Codes (Anisimov, & Zavadskyi, 2017), Binary Mixed-Digit Codes (Zavadskyi, & Kovalchuk, 2023). Also, there are known solutions that can improve the Huffman decoding throughoutput without compromising the compression efficiency. Among them, one of the simplest and most widely used is the Canonical Huffman Codes (CHC), first proposed in (Schwartz, & Kallick, 1964).

The research objects are Canonical Huffman Codes and their fast decoding algorithms.

The aim of this research is to optimize the state-of-the-art canonical Huffman code decoding algorithm proposed in (Moffat, & Turpin, 1997), specifically for cases where the encoded text contains a large number of long codewords.

The structure of the paper is as follows. In Section 1, we describe the idea of the CHC and different known up-to-date approaches to their fast decoding, as well as the more general but also efficient 'table' method of a variable-length code fast decoding. In Section 2, we discuss our main contribution - several optimizations to Algorithm TABLE-LOOKUP from (Moffat, & Turpin, 1997), which we consider as the baseline. In Section 3, we provide the results of experiments, where we measured the execution time of different CHC decoding methods.

1. Canonical Huffman Codes

Before we discuss the CHC, let us note that there exists a universal and simple 'table' method of decoding classical Huffman codes, as well as any variable length codes. Let $maxLen$ be the bitlength of the longest codeword. Then, we use lookup tables D and L that are indexed by all possible binary $maxLen$-bit vectors that start from a codeword. We call such bit vectors *buffer vectors*. Assume the bit representation of some buffer vector x starts from the codeword c. Then, the $D[x]$ is the result of the decoding of c (we also call it the result of the decoding of x), while $L[x]$ is the bitlength of c. The decoding process becomes almost trivial: at a given bitstream position, we read the maxLen-bit value x, output the result of the decoding $D[x]$ and shift the read position by $L[x]$ bits forward. This method can hardly be outperformed if $maxLen$ does not exceed 11 or 12 bits. However, each of these tables consists of about 2^{maxLen} elements, and when the maximum codeword length becomes bigger, tables do not fit into the processor cache, which slows down the decoding dramatically.

Therefore, on short alphabets, such as ASCII characters, DNA nucleotides or protein bases, the fast decoding task can be considered successfully solved by the 'table' method described above. However, compression of data on larger alphabets is no less important task, e.g. compression of word-based natural language texts. As experiments show, the

semi-static compression of a word-based English text can be almost twice as efficient as a character-based text. This is why word-based natural language text compression is the main subject of interest to us.

The Canonical Huffman Codes allow us to avoid using both the large lookup tables and, to some extent, bit-by-bit code processing. They assume using the dictionary model of the encoded text. All source alphabet symbols reside in the dictionary – the array sorted by the descending symbol frequencies. During the encoding, we find the text symbol in the dictionary and encode its index. The decoding is reverse: we decode the index and then use it to get the symbol from the dictionary. Therefore, bigger encoded numbers correspond to less frequent symbols.

For a given source symbol distribution, codewords of the CHC have the same lengths as the classical Huffman codes, and thus, the compression ratios of these code classes are the same. However, the most important property of the canonical codes is that the codewords of the same length form a contiguous set of integers. Let $baseCwd[i]$ be the smallest value of the codeword of length i and $baseSym[i]$ be the result of its decoding. Then, if we know that the buffer vector starts from the codeword c of length i, the result of its decoding can be calculated as $baseSym[i] + c - baseCwd[i]$. The codeword c can be obtained by shifting the buffer by $maxLen - i$ bits to the right. The only problem that remains is to get the codeword length given the buffer vector. In the original canonical codes, the desired length is searched linearly, i.e., the length l is incremented, starting from the minimal possible value, until the buffer value is greater or equal to the threshold $Limit[l]$, which is the smallest buffer value that starts from the smallest codeword of length l. This search can be faster than traversing the Huffman tree; nonetheless, it remains too slow. In (Moffat, & Turpin, 1997), the authors propose to get the initial length value from the lookup table $Start$ of limited size. They use b leftmost bits from the buffer as the index V_b of that table. $Start[V_b]$ is the length of the shortest codeword c, which is consistent with V_b. The word 'consistent' here means that either some prefix of V_b coincides with c or some prefix of c coincides with V_b. Then, the obtained length value l is incremented until the buffer value is no less than the threshold $Limit[l]$.

The detailed algorithm is given below.

We process the input bit array E, which is indexed by bit positions. The variable pos holds the current read position. After loading bits from E to $buffer$, we address $buffer$ both as a variable (line 5) and a bit array (lines 4 and 7). Two more 'mask and shift' operations are needed to get prefixes of the $buffer$ bit array in lines 4 and 7.

input : Encoded bit sequence $E[1..n]$;
Lookup tables $Start$, $Limit$, $baseSym$, $baseCwd$ defined above;
Dictionary D;
Maximal codeword length $maxLen$;
Length of the start lookup bit vector b.

1 $pos \leftarrow 1$;
2 **while** $pos < n$ **do**
3 $\quad buffer \leftarrow E[pos..pos + maxLen - 1]$;
4 $\quad l \leftarrow Start[buffer[1..b]]$;
5 $\quad$ **while** $buffer \geqslant Limit[l]$ **do**
6 $\quad\quad l \leftarrow l + 1$;
$\quad$ **end**
7 $\quad codeword \leftarrow buffer[1..l]$;
8 $\quad symbol \leftarrow baseSym[l] + codeword - baseCwd[l]$;
9 $\quad$ Output$(D[symbol])$;
10 $\quad pos \leftarrow pos + l$;
end

Algorithm 1: Decoding of the canonical codes - baseline algorithm

In (Liddell, & Moffat, 2006), an extension of Algorithm 1 was proposed (Algorithm 3 in the cited paper). The method combines a limited 'table' decoding procedure with the canonical Huffman decoder: p bits are first read from the bitstream, and if they already contain one or more complete codewords, decoding is performed using the 'table' method; otherwise, the canonical decoder is applied. Following authors, we denote this approach as **Ext-p**, where p is the parameter of the 'table' method, while the unmodified canonical decoder is referred to as the *Baseline*. However, the experimental results given in Tab. 1 demonstrate that this extension increases the decoding time compared to the baseline implementation when applied to texts on word-based alphabets, though it can be efficient for short character-based alphabets (the experimental environment is described in Section 3). A similar behavior was reported in (Liddell, & Moffat, 2006). For each text, we hereby take the best decoding time presented in Tab. 1 as the state-of-the-art baseline.

Table 1

Decoding time of the baseline canonical decoder and its extension (Ext-p) in seconds per 1000 iterations

File name	Baseline	Ext-6	Ext-8	Ext-10	Ext-11	Ext-12	Ext-13	Ext-14	Ext-15	Ext-20
Alice29	0.136	0.179	0.197	0.170	0.147	0.150	0.153	0.162	0.228	0.450
Harry Potter	0.385	0.482	0.550	0.565	0.503	0.506	0.506	0.527	0.764	2.665
Bible	3.738	5.934	5.515	4.760	4.219	4.194	4.139	4.226	5.769	16.918
Shakespeare	4.322	5.970	6.338	6.580	6.394	6.343	6.418	6.568	9.268	33.384
English.50MB	50.47	66.22	68.38	67.37	64.50	66.47	68.77	69.96	95.88	254.41
Shakespeare - char	2.671	2.297	1.593	1.165	1.184	1.261	1.416	1.574	2.292	5.060

Experiments on real-world word-based text compression show that, for $p = 12$, the loop in lines $5 - 6$ never has more

than 1 iteration and most often is not executed at all. The loop was never entered for files of about 0.5MB; on the $4-5$MB files, the probability of one iteration execution is less than 0.1%. At the same time, the size of the $Start$ lookup table is $2^p = 2^{12}$, which is much less than $2^{maxLen} \in [2^{17}..2^{22}]$ for the tables of the default method discussed in the Introduction, and acceptable to fit into L1 processor cache. Thus, the Algorithm 1 can be considered a fast and memory-efficient decoding method with $O(1)$ per symbol time complexity. Nonetheless, we will discuss a number of further improvements that can accelerate this method significantly.

2. Baseline algorithm optimizations

1. In Algorithm 1, arrays $baseSym$ and $baseCwd$ are used in line 8 only, both with the same index, in the expression $baseSym[l] - baseCwd[l]$. Thus, we can replace both these arrays with one array $Base$, $Base[i] = baseSym[i] - baseCwd[i]$. This simple replacement saves one subtraction and one loading from memory per decoded symbol.
2. It would be better to get the decoded value itself from the $Start$ table instead of the codeword length. However, if the alphabet size is bigger than 2^b, it is impossible to 'pack' all decoded values into the 2^b-element array. Nonetheless, some prefixes of the $buffer[1..b]$ bitstring may coincide with a whole codeword. For word-based natural language text compression, the probability of this event is about $0.7 - 0.95$, dependent on the value p and the type of text. This implies we can combine the 'table' method with the canonical codes if we store in the special array $Ncwd$ the number of whole codewords in the bitstring $buffer[1..p]$, where p is some predefined parameter; typically $p \leqslant b$. Then, if $Ncwd[buffer[1..p]] \geqslant 1$, we can extract one or two decoded numbers from $buffer[1..p]$ and proceed to the next iteration; otherwise, we decode the canonical codeword as in Algorithm 1.

   Let us note that the idea of retrieving several decoded results at each iteration of the decoded loop has repeatedly appeared, e.g., in (Choueka, Klein, & Perl, 1985) or (Milidiu et al., 2003). In application to CHC, it was investigated in detail in (Liddell, & Moffat, 2006). However, as we mentioned in 1, it appears not to be efficient in the practical decoding of texts on word-based alphabets. We claim that it is due to implementation shortcomings. The condition $Ncwd[buffer[1..p]] \geqslant 1$, which is checked in line 14 of Algorithm 3 in (Liddell, & Moffat, 2006), can be true or false with relatively high probability. Thus, the 'if' statement that checks it is unpredictable, and it slows down the program significantly. To avoid it, we exploit the advantage of the 64-bit architecture of modern processors, using the approach first implemented in the Binary Coded Ternary data compression code (Zavadskyi, 2022). Namely, we always output two 32-bit decoded symbols packed into a 64-bit word, even if the bitstring $buffer[1..p]$ contains one or zero full codewords. Then, we increase the counter of output symbols by $Ncwd[buffer[1..p]]$. Therefore, if $Ncwd[buffer[1..p]] \leqslant 1$, we output one or two fictitious numbers, and at the next iteration, the actual numbers will be output to the same memory.

   Both optimizations 1 and 2 are given in Algorithm 2. Apart from the lookup tables defined above, to shift the read position, the table $Shift$ in line 8 is used to store the total length of codewords that coincide with the prefix of $buffer[1..p]$. Also, we explicitly show using the output array as the logic of writing values to it is important.

input : Encoded bit sequence $E[1..n]$;
Lookup tables $Start$, $Limit$, $Base$, $Ncwd$, $Result$, $Shift$ defined above;
Dictionary D;
Maximal codeword length – $maxLen$;
Length of the start lookup bit vector – b;
Length of the $byffer$ prefix for table decoding – p.
output: Array of numbers Out.

```
1  pos ← 1;
2  outPos ← 1;
3  while pos < n do
4  |  buffer ← E[pos..pos + maxLen − 1];
5  |  prefix ← buffer[1..p];
6  |  Out[outPos, outPos + 1] ← Result[prefix];
7  |  outPos ← outPos + Ncwd[prefix];
8  |  pos ← pos + Shift[prefix];
9  |  l ← Start[buffer[1..b]];
10 |  while buffer ⩾ Limit[l] do
11 |  |  l ← l + 1;
   |  end
12 |  codeword ← buffer[1..l];
13 |  Out[outPos] ← D[Base[l] + codeword];
14 |  outPos ← outPos + 1;
15 |  pos ← pos + l;
   end
```

Algorithm 2: Hybrid table and canonical decoding

3. In Algorithms 1 and 2, we omitted the process of loading bits from the encoded bitstream E for simplicity. However, in the case of Canonical Huffman Codes, the decoding of individual symbols is already highly optimized, making the processing of the bitstream a significant contributor to the overall execution time. There are two baseline ways to load the chunk of the bitstream from memory to the variable $buffer$ (line 3).

(a) Byte-based buffer management. The bitstream is stored as a sequence of bytes. We need to divide pos by 8, load 8 bytes from memory to a $buffer$ processor register, and perform a shift operation to remove up to 7 previously decoded bits. However, the applicability of this method depends on the machine's endianness and whether the leftmost bit is stored as the least or most significant bit of the integer. Due to these dependencies, this approach is unsuitable for CHC decoding.

(b) Conditional buffer refilling. The buffer is always maintained with at least 32 valid bits. During decoding, the 'if' statement checks if fewer than 32 bits remain in the buffer. If so, a 32-bit integer is loaded into the buffer. This method is independent of endianness but introduces overhead due to frequent unpredictable conditional checks.

Both of these techniques refill the buffer for each decoded symbol. Now, we propose an optimization that minimizes the overhead associated with processing the bitstream E. Instead of refilling the buffer at every decoding iteration, we update it only once every $k = \lfloor \frac{64}{\text{maxLen}} \rfloor$ decoded symbols. Together with Optimization 1, this is presented in Algorithm 3. Each macro-iteration of the decoding proceeds as follows:

First, 64 bits are loaded into the buffer in line 5. In practice, the bitstream E is stored as a sequence of contiguous 64-bit integers. During the buffer update, the $buffer$ is updated with two adjacent 64-bit integers from E using bitwise OR operation. If necessary, the first 64-bit integer is shifted left to update the most significant bits of the $buffer$. Then, the 64-bit integer is shifted right to update the least significant bits of the $buffer$.

In lines 6–14, k iterations of the standard decoding are performed. Each iteration processes l bits and finishes with line 14, where l bits are removed from the top of the buffer.

input : Encoded bit sequence $E[1..n]$;
Lookup tables $Start$, $Limit$, $Base$ defined above;
Dictionary D;
Maximal codeword length $maxLen$;
Length of the start lookup bit vector b.
output: Array of numbers Out.

```
1  pos ← 1;
2  outPos ← 1;
3  k ← ⌊64/maxLen⌋;
4  while pos < n do
5  |  buffer ← E[pos..pos + 63];
   |  for i ← 1 to k do
6  |  |  prefix ← buffer[1..b];
7  |  |  l ← Start[prefix];
8  |  |  while buffer ⩾ Limit[l] do
9  |  |  |  l ← l + 1;
   |  |  end
10 |  |  codeword ← buffer[1..l];
11 |  |  Out[outPos] ← D[Base[l] + codeword];
12 |  |  outPos ← outPos + 1;
13 |  |  pos ← pos + l;
14 |  |  buffer ← shiftLeft(buffer, l);
   |  end
   end
```

Algorithm 3: Canonical decoding with fast buffer updates

4. Lines 7 and 8 of Algorithm 1 involve a shift operation to retrieve $codeword$ from $buffer$, lookup into $Base$ table (considering Optimization 1), and addition to get decoded symbol. The addition operation depends on the result of the preceding shift, potentially introducing latency. To optimize this, we propose moving all of the mentioned operations outside the main decoding loop. Both optimizations 1 and 4 are detailed in Algorithm 4. Instead of processing each symbol individually, the values of $buffer$ and lengths l are stored in temporary arrays $tmpBuffer$ and $tmpL$ during the decoding process (lines 10 and 11). Once enough symbols (e.g. 512) are processed this way, these values are decoded in batches using SIMD instructions to perform the shifts, lookups, and additions simultaneously (lines 13-17). Experiments demonstrate execution time improvements on processors supporting 256-bit SIMD operations, such as those with AVX and AVX2 instruction sets. In lines 13–18, variables ending with 8 represent SIMD registers, each holding 8 32-bit integers. The $simd\_load8$ operation loads 8 consecutive integers from memory into a SIMD register. The $simd\_subtract8$ operation subtracts corresponding integers in one SIMD register from another. The $simd\_shiftRight8$ operation shifts each integer in one register to the right by the number of bits specified in the corresponding integer of another register. The $simd\_lookup8$ operation fetches 8 values simultaneously from the $Base$ table, using integers in a SIMD register as indices. Finally, the $simd\_add8$ operation adds corresponding integers from two registers element-wise. These operations allow efficient parallel processing on 8 integers at once. Let us note that Optimization 2 is incompatible with Optimization 4. On processors limited to 128-bit SIMD without the required shift and table-lookup instructions, this optimization may actually degrade performance due to overhead. Conversely, on newer architectures with 512-bit SIMD (e.g., AVX-512), we expect higher speedup.

input : Encoded bit sequence $E[1..n]$;
Lookup tables $Start$, $Limit$, $Base$ defined above;
Dictionary D;
Maximal codeword length $maxLen$;
Length of the start lookup bit vector b.
output: Array of numbers Out.

$pos \leftarrow 1$;
$outPos \leftarrow 1$;
$maxLen8 \leftarrow simd\_fill8(maxLen)$;
while $pos < n$ **do**
  for $i \leftarrow 1$ **to** 512 **do**
    $buffer \leftarrow E[pos..pos + maxLen - 1]$;
    $prefix \leftarrow buffer[1..b]$;
    $l \leftarrow Start[prefix]$;
    while $buffer \geqslant Limit[l]$ **do**
      $l \leftarrow l + 1$;
    end
    $tmpBuffer[i] \leftarrow buffer$;
    $tmpL[i] \leftarrow l$;
    $pos \leftarrow pos + l$;
  end
  for $i \leftarrow 1$ **to** 512 ***by*** 8 **do**
    $buffer8 \leftarrow simd\_load8(tmpBuffer[i..i + 7])$;
    $l8 \leftarrow simd\_load8(tmpL[i..i + 7])$;
    $codeword8 \leftarrow simd\_shiftRight8(buffer8, simd\_subtract8(maxLen8 - l8))$;
    $base8 \leftarrow simd\_lookup8(Base, l8)$;
    $symbol8 \leftarrow simd\_add8(codeword8, base8)$;
    for $j \leftarrow 0$ **to** 7 **do**
      $Out[outPos] \leftarrow D[symbol8[j]]$;
      $outPos \leftarrow outPos + 1$;
    end
  end
end

Algorithm 4: Canonical decoding with SIMD optimization. 256-bit SIMD operations are assumed.

3. Experimental results

We have implemented both the baseline algorithm and its optimizations in C++ language and compiled them with the Visual Studio compiler[1]. We conducted testing on a Windows OS using an Intel i7-7700HQ processor (2.80 GHz) with 32 KB of L1 data cache per core and 16 GB of DDR4 RAM. Algorithms were tested by decoding five English texts encoded with the Canonical Huffman Codes on word-based alphabets. Text parameters are listed in Tab. 2.

The optimal value of the parameter b in all algorithms has been determined experimentally as $b = 12$, while the optimal value of the parameter p in Algorithm 2 has been determined as $p = 10$.

In Tab. 3, we give the total time of 1000 runs of different algorithms and their combinations. We did not evaluate the efficiency of optimizations 2, 3, and 4 without optimization 1, since the latter is straightforward and does not adversely affect any of the other improvements. To compare the productivity fairly, we did not measure the time of common text postprocessing operations, such as converting integers to strings or concatenating words to form the resultant text in memory.

Table 2

Text parameters

File	Size	Total symbols	Different symbols	Entropy H0 (bytes)	Encoded size	Average codeword length (bits)
Alice's Adventures in Wonderland by L. Carrol	145KB	26,446	3241	29,747	29,860	9.03
Harry Potter and the Philosopher's Stone	427KB	78,419	7067	95,450	95,792	9.77
The Bible, King James version	3593KB	766,111	13,754	850,668	853,812	8.91
Shakespeare's complete works	5320KB	899,822	36,693	1,189,654	1,192,828	10.6
50MB English text from Pizza&Chilie corpus	51200KB	9,435,637	148,865	12,389,798	12,421,116	10.53
Shakespeare's complete works - character-based alphabet	5320KB	5,447,171	84	3,131,931	3,149,568	4.63

[1]The source code can be downloaded from https://github.com/reeWorlds/Fast-Large-Huffman/tree/main/Test3.

Table 3

Decoding time comparison in seconds per 1000 iterations

File	State-of-the-Art	Opt. 1	Opt. 1 + 2	Opt. 1 + 3	Opt. 1 + 4	Opt. 1 + 2 + 3	Opt. 1 + 3 + 4
Alice	0.136	0.124	0.113	0.101	0.118	0.106	0.098
Harry Potter	0.385	0.362	0.348	0.309	0.341	0.328	0.293
Bible	3.738	3.516	3.071	3.051	3.279	2.836	2.855
Shakespeare	4.322	4.114	4.172	3.507	3.873	3.828	3.260
English.50MB	50.47	45.04	42.54	41.93	43.30	41.74	38.58
Shakespeare - char	1.165	2.281	1.574	2.079	2.171	1.369	2.017

Combining different optimizations allows us to accelerate the baseline algorithm significantly. As optimizations 2 and 4 are incompatible, we tested optimizations $1+2+3$ and $1+3+4$. The latter option demonstrates the best results for all word-based texts except for The Bible, where optimizations $1+2+3$ are the best. This is explained by the fact that the average codeword length for The Bible is the shortest, which means that two codewords can most often be processed within the same iteration of the decoding loop in Algorithm 2. By contrast, for Shakespeare's complete works, where the average codeword length is the largest, optimization 2 actually reduces the effect of optimization 1. The state-of-the-art decoding (Moffat, & Turpin, 1997) is $31-39$% slower than the methods that use the full stack of our optimizations.

We also included one character-based text in our experimental set (the last line in all tables). Its average codeword length is roughly half that of the word-based texts. Consequently, Algorithm 3 from (Liddell, & Moffat, 2006) proves to be the most time-efficient, even for moderate values of the lookup bitvector length p. Thus, for $p=10$, the condition $Ncwd[buffer[1..p]] \geqslant 1$, which is checked in line 14 of Algorithm 3 in (Liddell, & Moffat, 2006), is almost always satisfied, which makes the corresponding 'if' statement predictable and removes the main source of inefficiency in that algorithm. The Ext-10 method from (Liddell, & Moffat, 2006) outperforms not only the baseline algorithm from (Moffat, & Turpin, 1997), but also our most efficient stack of optimizations $1+2+3$.

Discussion and conclusions

We offer optimizations of Huffman Canonical Codes that improve state-of-the-art decoding time for texts on word-based alphabets by more than 30%. The decoding algorithms with our optimizations are faster than any other known decoding method for large alphabets, where codeword lengths may exceed 11-12 bits, except for BCT codes (Zavadskyi, 2022), which are faster but produce about 5% larger files. The proposed methods can be used in a wide range of data compression and information retrieval systems.

Authors' contribution: Igor Zavadskyi – conceptualization, methodology, Algorithm 2; Maksym Kovalchuk – software, empirical data collection and validation, Algorithms 3 and 4.

Sources of funding. Funding is partially provided by Taras Shevchenko National University of Kyiv.

References

Anisimov, A. V., & Zavadskyi, I. O. (2017). Variable-length prefix codes with multiple delimiters. *IEEE Transactions on Information Theory*, 63(5), 2885–2985. https://doi.org/10.1109/TIT.2017.2674670

Brisaboa, N. R., Fariña, A., Navarro, G., & Esteller, M. F. (2003). (s,c)-dense coding: An optimized compression code for natural language text databases. In Nascimento, de Moura, & Oliveira (Eds.), *String Processing and Information Retrieval* (pp. 122–136). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-39984-1_10

Choueka, Y., Klein, S. T., & Perl, Y. (1985). Efficient variants of huffman codes in high level languages. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (p. 122–-130). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/253495.342777

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

Liddell, M., & Moffat, A. (2006). Decoding prefix codes. *Software Practice and Experience*, 36, 1687–1710.https://doi.org/10.1002/spe.741

Milidiu, R., Laber, E., Moreno, L., & Duarte, J. (2003, March 18–20). A fast decoding method for prefix codes. In Storer & Cohn (Eds.), *Data Compression Conference* (p. 438). Snowbird, Utah, USA. https://doi.org/10.1109/DCC.2003.1194057

Moffat, A., & Turpin, A. (1997, October). On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10), 1200–1207. https://doi.org/10.1109/26.634683

Schwartz, E., & Kallick, B. (1964). Generating a canonical prefix encoding. *Communications of the ACM*, 7(3), 40–50. https://doi.org/10.1145/363958.363991

Zavadskyi, I. (2022, March 19–21). Binary-coded ternary number representation in natural language text compression. In Bilgin, Marcellin, Serra-Sagristà, & Storer (Eds.), *Data Compression Conference* (pp. 419–428). Snowbird, Utah, USA: IEEE. https://doi.org/10.1109/DCC52660.2022.00050

Zavadskyi, I., & Kovalchuk, M. (2023, September 26-28). Binary mixed-digit data compression codes. In Nardini, Pisanti, & Venturini (Eds.), *String Processing and Information Retrieval* (pp. 381–392). Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-43980-3_31

Отримано редакцією журналу / Received: 31.01.25
Прорецензовано / Revised: 24.09.25
Схвалено до друку / Accepted: 10.10.25

Ігор ЗАВАДСЬКИЙ, д-р фіз.-мат. наук, проф.
ORCID ID: 0000-0002-4826-5265
e-mail: ihorzavadskyi@knu.ua
Київський національний університет імені Тараса Шевченка, Київ, Україна

Максим КОВАЛЬЧУК, асп.
ORCID ID: 0009-0002-8450-905X
e-mail: max.koval4uk@knu.ua
Київський національний університет імені Тараса Шевченка, Київ, Україна

ОПТИМІЗАЦІЯ КАНОНІЧНИХ КОДІВ ХАФМАНА ДЛЯ ПОСЛІВНОГО СТИСНЕННЯ ПРИРОДНОМОВНИХ ТЕКСТІВ

Канонічні коди Хафмана – це метод стискального кодування джерела даних, що дає можливість швидкого декодування, не зменшуючи ефективності стиснення порівняно з класичними кодами Хафмана. Якщо максимальна довжина кодового слова не перевищує 11–12 бітів або довгі кодові слова трапляються дуже рідко, будь-який код змінної довжини, зокрема коди Хафмана, можна ефективно декодувати наївним "табличним" методом. Канонічні коди Хафмана доцільно використовувати насамперед тоді, коли стиснутий текст містить достатньо багато довгих кодових слів. Наприклад, це характерно для природномовних текстів, якщо елементами алфавіту вважати слова. Найшвидший на сьогодні алгоритм декодування канонічних кодів розроблено М. Лідлем та А. Мофатом. Ми пропонуємо кілька оптимізацій цього алгоритму, що, як свідчать проведені нами експерименти, дають змогу прискорити його на 20–30 %, не знижуючи коефіцієнта стиснення. Деякі з цих оптимізацій базуються на низькорівневому розпаралелюванні коду з використанням SIMD-архітектури сучасних процесорів, в інших пропонуємо поєднувати класичне декодування канонічних кодів із універсальним "табличним" методом. Крім згаданих оптимізацій, у статті детально розглянуто канонічні коди Хафмана, включно з базовим та оптимізованим М. Лідлем і А. Мофатом алгоритмами, а також універсальний "табличний" метод декодування кодів змінної довжини. Представлено результати експериментального порівняння швидкодії базового та оптимізованих алгоритмів. Результати дослідження можуть бути використані в системах інформаційного пошуку й архівування даних.

Ключові слова: ***Хафман, кодування, коди, канонічні, стискання тексту.***

Автори заявляють про відсутність конфлікту інтересів. Спонсори не брали участі в розробленні дослідження (у зборі, аналізі чи інтерпретації даних, якщо це мало місце), у написанні рукопису та в рішенні про публікацію результатів.

The authors declare no conflicts of interest. The funders had no role in the design of the study (in the collection, analyses or interpretation of data if applicable), in the writing of the manuscript as well as in the decision to publish the results.