Best Practices in Adaptive Encoding

Igor Zavadskyi and Maksym Kovalchuk

Taras Shevchenko National University of Kyiv
2d Glushkova ave.
Kyiv, Ukraine
ihorzavadskyi@knu.ua, max.koval4uk@knu.ua

Abstract. We discuss and experimentally evaluate a range of classical and recently developed adaptive data compression algorithms. Key features of each method are highlighted, and their effectiveness in real-world data compression scenarios is assessed. By combining these techniques, we construct a family of simple adaptive encoding methods that achieve an excellent space-time trade-off, particularly for text with small alphabets such as ASCII. Compared to classical Vitter's algorithm, our method is several times faster while maintaining a similar compression ratio. Compared to Gagie's worst-case optimal solution, our algorithm generates compressed files that are about 10% smaller, while also delivering faster encoding and decoding speeds.

1 Introduction and Related Work

Adaptive encoding is a well-established data compression technique that enables "online" compression, meaning the encoder does not require prior knowledge of the entire file's statistical characteristics. Instead, it infers this information from the portion of the file it has already encoded and uses it to estimate the properties of the remaining data. Compared to non-adaptive methods, this approach allows the encoder to make a single pass through the text, and often leads to more accurate probability estimates for the symbols. Classical statistical compression algorithms using adaptive techniques were introduced by Faller [1], Gallager [5], Knuth [7] (FGK), and Vitter [11].

Both the FGK and Vitter algorithms adapt the Huffman tree dynamically, updating it after processing each symbol. The FGK method has been shown in [9] to use, in the worst case, $\delta + 2$ bits per symbol more than the zero-order entropy H, where δ represents the overhead of static Huffman coding over the entropy. Vitter's more advanced algorithm reduces this overhead to $\delta + 1$ bits. In practice, however, both the FGK and Vitter methods often outperform the theoretical entropy limit H. In [2], Gagie revised the FGK algorithm to implement adaptive Shannon encoding rather than Huffman coding, which eliminated the δ term and reduced the worst-case space usage to $H + 1$ bits per symbol. This estimation is near-optimal as in [4] Gagie and Nekrich proved the lower bound for the worst-case space, $H + 1 - o(1)$ bits per symbol.

Updating the Huffman tree takes $O(H)$ time per symbol for both encoding and decoding. However, it is not likely that its structure will change with every next symbol, especially if the alphabet is small. To improve time efficiency, the natural idea is to update the code over intervals. With the appropriate relation between the interval length, the length of a text, and the alphabet size, the code may remain space-optimal. This idea was first implemented by Karpinsky and Nekrich [6]. The authors introduced an advanced algorithm using the canonical Shannon code with quantized code updates. Although each update still takes $O(l_{max})$ time (where l_{max} is

Igor Zavadskyi, Maksym Kovalchuk: Best Practices in Adaptive Encoding, pp. 3–12.
Proceedings of PSC 2025, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07461-9 © Czech Technical University in Prague, Czech Republic

the longest codeword length), updates are spaced, distributing the cost over chunks of symbols and achieving $O(1)$ amortized time per symbol. In addition, their method reduces the decoding time to $O(\log H)$ per symbol. The worst-case space complexity is guaranteed to be $H + 1$ bits per symbol as long as $\sigma = o(n/\log^2 n)$, where σ is the alphabet size, and n is the total length of the text.

Shannon's prefix-free code is convenient for estimating worst-case space, as it guarantees that a symbol having probability p is assigned to a codeword of length at most $\lceil \log 1/p \rceil$ bits (hereinafter log denotes the binary logarithm). In [3], Gagie proposed an algorithm based on Shannon code that maintains a space usage of $H + 1$ bits per symbol and supports constant-time encoding and decoding for the alphabets consisting of $o(\sqrt{n}/\log n)$ symbols. Similarly to [6], Gagie's method performs code updates periodically over intervals of length $\lceil \sigma \log n \rceil$. Before every code update, character probabilities are *smoothed*, i.e., replaced with the weighted average of the character's probability with weight $(\log n - 1)/\log n$, and uniform distribution with weight $1/\log n$. This technique restricts the maximal codeword length, reducing the size of a decoding lookup table and its construction time. Although Gagie's algorithm [3] is the first worst-case optimal method in terms of space and time complexity that can be easily implemented in practice, it was published as a theoretical contribution only. Its experimental testing is one of the goals of our research.

Yet another approach to accelerate adaptive encoding and decoding was proposed in [12]. The core idea is to decouple the update of the codeword set from the update of the mapping between alphabet symbols and codewords. For Huffman codes, updating the codeword set takes $O(H)$ time per symbol, while updating the symbol-to-codeword correspondence can be done in constant time by simply swapping two entries in the (symbol, codeword) map, which is ordered by symbol frequencies. In this method, an $O(1)$-time swap is performed after processing each symbol, whereas the full codeword set is updated at geometrically increasing intervals.

This approach is particularly effective for large alphabets, such as word-based representations of natural language texts. In such cases, the data typically contains many symbols with small frequencies, e.g. 1 or 2, causing frequent changes in the relative order of symbols by frequency. However, the set of codewords itself changes much less often. As a result, the algorithm [12] achieves significant speedups − by orders of magnitude over Vitter's algorithm − at the cost of only about a 1% reduction in compression efficiency.

In this paper, we focus on the compression of text over small, e.g. character-based, alphabets. In this case, neither the frequent swapping of the (symbol, codeword) map elements nor the symbol-after-symbol updating of the codeword set is necessary. In such scenarios, the frequency order of characters changes less frequently, making the method [12] less time-efficient. Nevertheless, the key idea of updating the codeword set at geometrically increasing intervals proves valuable even for small alphabets, as shown by the experiments in Section 3. This strategy results in only a few code updates per file.

We refer to algorithms that require a small number of code updates as *low-adaptive*. The concept of low adaptivity, along with other ideas discussed above, can be explored in various combinations to further improve the trade-off between compression efficiency and encoding/decoding speed. Namely,

- update the code over fixed-length or variable-length intervals;
- use smoothed or non-smoothed symbol probabilities;

– use Shannon or Huffman codes, canonical or non-canonical.

All these options can be implemented on the basis of a very simple lookup table-based approach:

– during the encoding, get codewords from the lookup table indexed by characters;
– during the decoding, use several bits from the encoded bitstream as indices of lookup tables allowing us to get the output character and the bit length of the decoded codeword; shift the bitstream read position by this length;
– update the codeword set together with the lookup tables over some intervals.

In Section 2, we implement this baseline approach in encoding and decoding algorithms featuring some memory tricks that reduce the number of time-consuming bit-level operations. In Section 3, we discuss the results of experiments introducing the above-listed techniques into the baseline algorithms. Gagie's original algorithm is one of the options in this more general schema. Conclusions on the practical applicability of different approaches to adaptive encoding are given in Section 4.

2 Encoding and Decoding Algorithms

The baseline low-adaptive encoding approach is implemented in Algorithm 1. The code itself, as well as the correspondence between characters and codewords, are updated in variable-length intervals (line 20). In homogeneous files, it is reasonable to increase the interval length geometrically with some constant *rate* (line 21). A substring between two code update points is processed in lines $7-19$. If we use fixed-length intervals between code updates, set $rate = 1$.

We store two precomputed lookup tables indexed by characters: $\mathsf{Codewords}[c]$ consists of the codeword corresponding to the character c, while $\mathsf{Lengths}[c]$ stores its bitlength. In lines $8-10$ we read a character, increase its frequency, and move the read position forward. In lines $11-18$, the codeword of the character c is appended to the output bitstream. To accelerate the bit-wise operations, we output 32-bit words, which are constructed inside the 64-bit variable *buffer*, starting from its leftmost bit. Each codeword is shifted by *shift* bits to the left to find its appropriate bit position in the buffer. The variable *shift* initially equals 64 and at each iteration is decreased by the length of a codeword (line 11). A codeword is appended to the buffer in line 12. When the left half of the buffer is full (line 13), we output it (lines $14-15$) and shift the buffer by 32 bits to the left (lines $16-17$). It is assumed that there are no codewords longer than 32 bits, which is realistic for character-based compression.

In the reverse decoding Algorithm 2, the code is updated in the same intervals as during the encoding (lines $20-21$). To reduce the number of time-consuming bit-level operations, we use the 64-bit *buffer*, which is initialized with the first 8 bytes of the code in line 3. The current codeword is aligned with the leftmost bit of the buffer. We decode it and get its length using the function DecodeCodeword in line 8, which can be implemented differently depending on the underlying code. In lines $9-11$, we output the decoded character and increase its frequency. In line 12, we shift the buffer by the length of the codeword to the left to align the next codeword with the leftmost bit of the buffer. In line 13, we update the total shift length in the variable *shift*. When it exceeds 32 bits, we insert the next 32-bit word to the buffer and decrease the shift value by 32 (lines $14-18$).

Algorithm 1: Adaptive encoding

```
   input  : - String Text[1..n];
            - Position of the first code update initialIntervalLength;
            - Ratio of interval length between code updates rate;
            - Precomputed codewords array Codewords;
            - Precomputed codeword length array Lengths.
   output: The bitstream of codewords Out, indexed by bytes.
 1 shift ← 64;
 2 buffer ← 0;
 3 foreach c ∈ A do Freq[c] ← 0;
 4 intervalLength ← initialIntervalLength;
 5 inPos ← 1; outPos ← 1;
 6 while inPos < n do
 7 |   for i ← 1 to intervalLength do     // Process block of characters
 8 |   |   c ← Text[inPos];                          // Read the character
 9 |   |   Freq[c] ← Freq[c] + 1;                    // Increase frequency
10 |   |   inPos ← inPos + 1;                  // Shift the read position
11 |   |   shift ← shift − Lengths[c];        // Fill the buffer from left
12 |   |   buffer ← buffer + (Codewords[c] << shift);
13 |   |   if shift ≤ 32 then          // Output 32 bits from the buffer
14 |   |   |   Out[outPos..outPos + 3] ← buffer >> 32;
15 |   |   |   outPos ← outPos + 4;
16 |   |   |   buffer ← buffer << 32;
17 |   |   |   shift ← shift + 32;
18 |   |   end
19 |   end
20 |   UpdateCode(Freq);          // Update the code and interval length
21 |   intervalLength ← min{n − inPos, intervalLength · rate};
22 end
```

A possible smoothing of character probabilities before the code update does not affect encoding/decoding algorithms as it is encapsulated in the function UpdateCode.

The decoding of a noncanonical codeword is simple and shown in Algorithm 3. In line 1, we get the *maxLen*-bit value, which consists of the current codeword and is used as the index of lookup tables to get the decoded character in line 2 and the codeword length in line 3.

This approach works well unless *maxLen* exceeds $12 - 13$ bits. The tables Decode and DecodeLen then become too large for the cache memory, which dramatically slows down decoding. To address this issue, one can use the *Canonical Huffman Codes* (CHC), first introduced in [10]. For a given source symbol distribution, codewords of CHC have the same lengths as the classical Huffman codes, and thus, the compression ratios of these code classes are the same. In addition, the adaptive encoding and general decoding algorithms are the same as for non-canonical codes (Algorithms 1 and 2); the only difference is the content of the lookup tables, which are filled in the function UpdateCode in line 20. However, the DecodeCodeword function is quite different and is shown in Algorithm 4.

Consider a list of codewords sorted by increasing length, which corresponds to a list of characters sorted by decreasing frequency. To decode a codeword, find its index

ind in this list. The decoded character is then given by Dict[*ind*], where Dict is the dictionary.

Algorithm 2: Adaptive decoding

input : - Encoded bitstream Code, indexed by bytes;
- Decoded string length n;
- Position of the first code update $initialIntervalLength$;
- Ratio of interval length between code updates $rate$.

output: Decoded string Text.

```
1  shift ← 0;
2  foreach c ∈ A do Freq[c] ← 0;
3  buffer ← Code[1..8];
4  intervalLength ← initialIntervalLength;
5  inPos ← 5; outPos ← 1;
6  while outPos < n do
7      for i ← 1 to intervalLength do
8          (c, length) ← DecodeCodeword(buffer);
9          Text[outPos] ← c;                          // Output the character
10         outPos ← outPos + 1;                  // Shift the output position
11         Freq[c] ← Freq[c] + 1;                      // Increase frequency
12         buffer ← buffer << length;          // Remove codeword from buffer
13         shift ← shift + length;
14         if shift ≥ 32 then              // Insert 32-bit word to the buffer
15             shift ← shift − 32;
16             inPos ← inPos + 4;
17             buffer ← buffer + (Code[inPos..inPos + 3] << shift);
18         end
19     end
20     UpdateCode(Freq);              // Update the code and interval length
21     intervalLength ← min{n − outPos, intervalLength · rate};
22 end
```

Algorithm 3: Function DecodeCodeword – Decoding the current codeword and obtaining its length for a noncanonical code.

input : - Maximal codeword length $maxLen$;
- Precomputed decoding table Decode;
- Precomputed array of codeword lengths DecodeLen;
- 64-bit chunk of the input stream $buffer$.

output: Decoded character c and the $length$ of its codeword.

```
1  x ← buffer >> (64 − maxLen);                  // Get the maxLen-bit value
2  c ← Decode[x];
3  length ← DecodeLen[x];
4  Return c, length.
```

The most important property of canonical codes is that codewords of the same length form a contiguous set of integers. Let baseCwd[l] be the smallest value of the codeword of length l, and baseSym[l] be the result of its decoding. Then, if we know that the buffer bit-vector starts from the codeword cwd of length l, the result of its decoding can be calculated as baseSym[l] + cwd − baseCwd[l] (line 6). The codeword cwd can be obtained by shifting the 64-bit buffer by $64 - l$ bits to the right (line 5).

The only problem that remains is to get the length of the codeword 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 minimum possible value, until the buffer value is greater or equal to the threshold $\mathsf{Limit}[l]$ – this is the smallest buffer value, which bit representation starts from the smallest codeword of length l. This search can be faster than traversing the Huffman tree; however, it remains too slow. In [8], Liddell and Moffat propose to get the initial length value from the lookup table Start of limited size (line 1). They use p leftmost bits from the buffer as the index x of that table (values p in the range $8 - 10$ are a good choice). $\mathsf{Start}[x]$ is the length of the shortest codeword cwd, which is consistent with x. The word 'consistent' here means that either some prefix of x coincides with cwd or some prefix of cwd coincides with x. Then, the obtained length value l is incremented until the buffer value is not less than the threshold $\mathsf{Limit}[l]$ (lines $2 - 4$).

As a result, we avoid using lookup tables with more than 2^p elements versus 2^{maxLen} elements in noncanonical codes.

Algorithm 4: Function DecodeCodeword – Decoding the current codeword and obtaining its length for canonical Huffman codes.

input : - 64-bit chunk of the input stream *buffer*;
- Lookup table for the initial codeword length values Start;
- Bit length p of the lookup table Start index;
- Threshold for buffer values Limit;
- The smallest value of a codeword of a given length baseCwd;
- The result of the decoding of the smallest codeword of a given length baseSym;
- Dictionary Dict.

output: Decoded character c and the length l of its codeword.

```
1 l ← Start[buffer >> (64 − p)];                  // Get the initial length
2 while buffer ≥ Limit[l] do                      // Find the true length
3 |  l ← l + 1;
4 end
5 cwd ← buffer >> (64 − l);          // Get l-bit codeword from the buffer
6 c ← Dict[baseSym[l] + cwd − baseCwd[l]];          // Decode the codeword
7 Return c, l.
```

3 Experiments

Compression experiments were conducted on 4 texts of different sizes and nature.

1. *Harry Potter, vol. 1.* 439,741 bytes. $H_0 = 251,221$ bytes = 4.57 bits/character.
2. *The Bible, King James version.* 4,047,392 bytes. $H_0 = 2,197,350$ bytes = 4.343 bits/character.
3. *The Bible, King James version*, preprocessed with Burrows-Wheeler + Move-To-Front transforms. 4,047,392 bytes. $H_0 = 1,550,143$ bytes = 3.064 bits/character.
4. *Source program code* from Pizza&Chili corpus. 20,055,515 bytes. $H_0 = 13,915,769$ bytes = 5.551 bits/character.

Table 1 presents the compressed file sizes in bits per character with the percentage of excess over the entropy given in brackets. In each pair of lines in Table 2, the encoding (upper) and decoding (lower) times are shown. They are averaged over 100

runs of each algorithm on a PC with an AMD Athlon 3000G processor, 3.50 GHz, 2 cores, 32 KB L1 data cache per core, L2 cache – 512 KB per core, L3 cache – 4 MB, 16 GB RAM. Algorithms implemented in C++ and compiled using the g++ compiler with full optimization for speed.[1]

Let us compare the results in different dimensions.

- **Code type.** Vitter's algorithm updates the code after processing each character and consistently produces the smallest compressed files. However, our low-adaptive Huffman code-based method performs nearly as well − less efficient by a fraction of a percent − and even surpasses Vitter's algorithm in one case (Text 3). In terms of speed, Vitter's approach is always several times slower than the low-adaptive encoding with variable-length update intervals, regardless of whether they use Shannon or Huffman codes. All adaptive methods based on Shannon codes exhibit significantly lower compression efficiency, producing files approximately 10 % or more above the entropy, compared to about 1 % overhead for Huffman-based methods. Nevertheless, the Shannon-based variant with variable-length update intervals (column 5 in Table 2) achieves the fastest encoding and decoding times. Although its encoding speed is similar to that of Huffman-based methods (columns 8 and 10), its decoding is nearly twice as fast as that of Huffman canonical codes.
- **Update interval.** For the first three texts, variable-length update intervals yield compression ratios comparable to or better than those achieved with fixed-length intervals (as defined in Gagie's algorithm). This is attributed to the relative homogeneity of these texts. In contrast, the fourth text − comprising a mix of source code snippets in different programming languages, large constant arrays, and other heterogeneous components − benefits more from frequent updates, making fixed-length intervals advantageous. However, the improvement in compression ratio is modest (less than 1.5 %), while the encoding and decoding are significantly slower across all texts. This indicates that the fixed-length update strategy, as proposed by Gagie, introduces a major performance bottleneck. The slowdown becomes especially critical when the decoding table is large enough to exceed cache capacity (see columns 4, 7, and 9 of Table 2).
- **Smoothing.** The smoothing technique proposed by Gagie adjusts character probabilities to reduce the size of the decoding table and thereby accelerate both its construction and overall decoding performance. This effect is validated experimentally (see column 3 vs. column 4, and column 5 vs. column 6 in Table 2). The trade-off is a slight loss in compression efficiency − less than 1.5 % in most cases, except for the BWT+MTF preprocessed text, where it approaches 7 %. This exception is due to frequent character repetitions in that text, where small changes in probability distributions over short intervals can substantially alter the codeword set. Notably, smoothing using the formula from [3] does not affect the structure of the Huffman tree in our experiments and therefore does not affect the results for Huffman-based encodings. For this reason, smoothed and non-smoothed variants of Huffman-based methods are not distinguished in Tables 1 and 2.
- **Canonical codes.** Canonical Huffman codes are used to speed up the decoding of long codewords (typically those exceeding 12–13 bits) without affecting compression quality. Long codewords often appear when character frequencies differ significantly, as can happen in large texts or when statistics are gathered over longer intervals. This is supported by the data in columns 8 and 10 of Table 2,

[1] The source code can be found at https://github.com/zavadsky/low-adaptive.

where canonical decoding outperforms its non-canonical counterpart for three of the larger texts and underperforms only on the shortest one. The benefit is especially noticeable in the BWT+MTF preprocessed text, where character frequency differences within blocks are particularly large.

Table 1: Comparison of the compression efficiency, bits/character
(FLI - Fixed Length Intervals, VLI - Variable Length Intervals).

Text	Vitter	Shannon-based				Huffman-based	
		Gagie original	*FLI, non-smoothed*	*VLI*	*VLI, non-smoothed*	*FLI*	*VLI*
1	4.618 (1.04%)	5.149 (12.65%)	5.099 (11.57%)	5.137 (12.4%)	5.091 (11.4%)	4.629 (1.29%)	4.625 (1.21%)
2	4.385 (0.97%)	4.872 (12.19%)	4.852 (11.7%)	4.87 (12.12%)	4.824 (11.06%)	4.387 (1.02%)	4.39 (1.08%)
3	3.079 (0.49%)	3.625 (18.31%)	3.415 (11.47%)	3.443 (12.36%)	3.418 (11.54%)	3.085 (0.69%)	3.078 (0.49%)
4	5.57 (0.34%)	6.139 (10.6%)	6.064 (9.24%)	6.22 (12.06%)	6.136 (10.54%)	5.572 (0.38%)	5.614 (1.13%)

Table 2: Encoding and decoding time, seconds
(FLI - Fixed Length Intervals, VLI - Variable Length Intervals).

Text	Vitter	Shannon-based				Huffman-based		Canonical Huffman	
		Gagie original	*FLI, non-smoothed*	*VLI*	*VLI, non-smoothed*	*FLI*	*VLI*	*FLI*	*VLI*
1	2	3	4	5	6	7	8	9	10
1	0.063	0.0169	0.017	0.0051	0.0051	0.017	0.0051	0.017	0.0053
	0.176	0.0197	0.234	0.0058	0.04	0.28	0.0084	0.28	0.011
2	0.578	0.11	0.117	0.04	0.041	0.136	0.042	0.131	0.042
	0.582	0.14	15.52	0.049	0.12	25	0.098	24.16	0.093
3	0.423	0.134	0.129	0.045	0.047	0.16	0.051	0.152	0.048
	0.533	0.166	10.55	0.053	0.192	15.92	0.279	15.55	0.102
4	3.44	0.692	0.642	0.22	0.215	0.695	0.222	0.724	0.236
	4.51	0.723	147	0.26	0.787	227	0.675	215	0.48

Let us note that all codes in research satisfy the $H + 1$ compressed size limit, proved in theory for Gagie's algorithm. However, for a character-based alphabet, this one extra bit per character gives more than 20% of the compressed file size. Therefore, the worst-case optimality limit is too weak for practical applicability.

The experimental results are summarized in the plane (compressed size, decoding speed) shown in Figure 1. Each algorithm is represented with four markers that correspond to four tested texts. Compression rate is given as the percentage of excess over the entropy, while the decoding speed is shown in microseconds per character. Among algorithms that update the code over fixed-length intervals, only Gagie's original algorithm is presented, as all the others have extremely high decoding times (columns 4,7, and 9 of Table 2).

All markers in Figure 1 can be divided into two groups: the right group corresponds to algorithms based on Shannon codes, while the left group corresponds to algorithms based on Huffman codes. The wide gap between the two groups highlights the low compression efficiency of Shannon codes in practical applications. However, two series of round markers occupy the most attractive Pareto-optimal areas. The round filled markers correspond to Shannon-based encoding with smoothed character probabilities and variable-length update intervals. This method provides the worst compression ratio (though not significantly worse than other Shannon-based encodings), but the best decoding time. The round empty markers correspond to adaptive encoding with variable-length update intervals based on Huffman canonical codes. It combines the compression rate almost on a level with Vitter's algorithm, with good decoding speed, which is inferior only to the fastest version of Shannon-based codes.

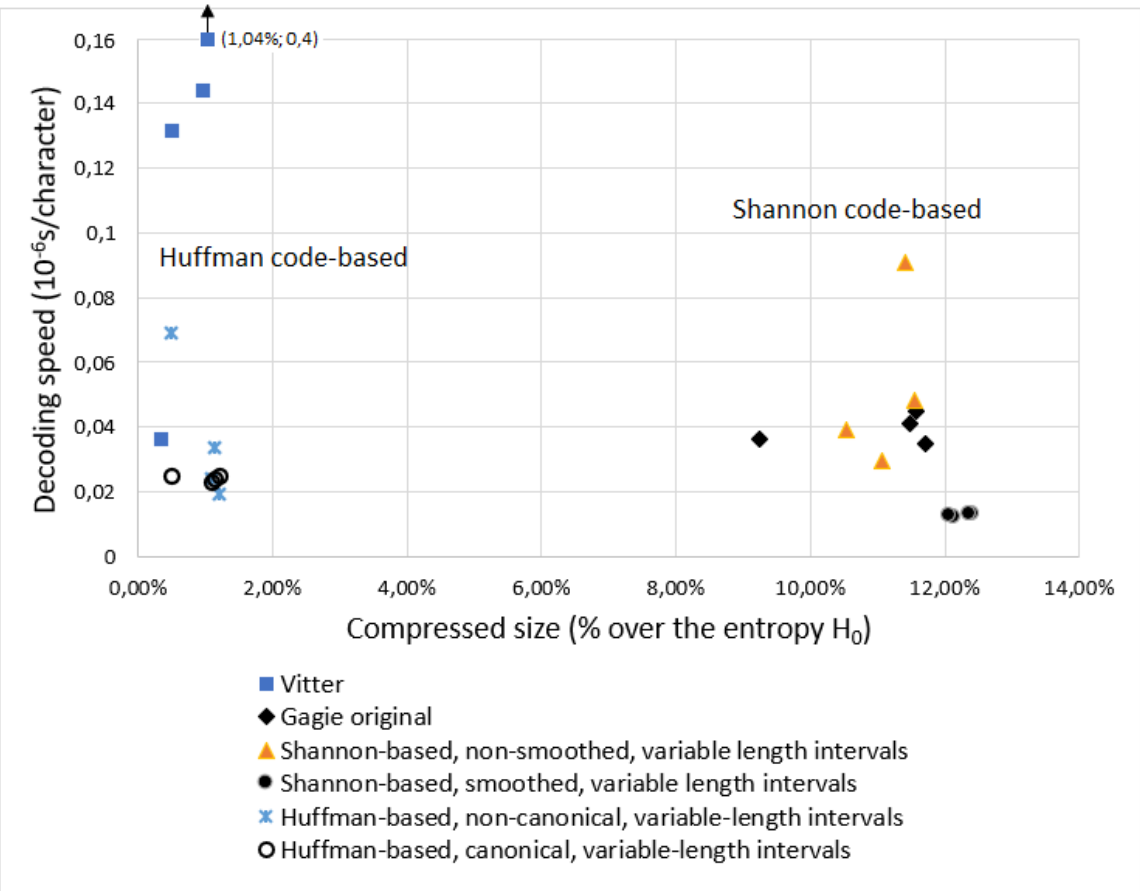

Figure 1: Compression efficiency and decoding speed of tested algorithms

4 Conclusions

We investigate the practical applicability of various classical and recently proposed adaptive encoding algorithms for character-level text compression. Among the evaluated methods, Vitter's classical algorithm consistently achieves the best compression ratio. However, alternative approaches significantly outperform it in terms of encoding and decoding speed. Gagie's algorithm, which is provably optimal in both space and time in the worst case, based on properties of the underlying Shannon code, appears

to be space-inefficient in practice – an issue shared by other Shannon code-based adaptive methods. Nonetheless, when the code is updated at geometrically increasing intervals rather than at the originally proposed $\lceil \sigma \log n \rceil$ intervals, the algorithm attains the highest observed encoding and decoding speeds. Furthermore, replacing the Shannon code with a canonical Huffman code yields a highly practical encoding scheme, achieving a compression ratio comparable to that of Vitter's algorithm while offering significantly superior performance in terms of speed. The theoretical investigation of worst-case optimality for Huffman code-based adaptive compression methods remains an open direction for future research.

References

1. N. Faller: *An adaptive system for data compression*, in Record of the 7th Asilomar Conference on Circuits, Systems and Computers, 1973, pp. 593–597.
2. T. Gagie: *Dynamic Shannon coding*, in Proceedings of the 12th European Symposium on Algorithms (ESA), 2004, p. 359–370.
3. T. Gagie: *Simple worst-case optimal adaptive prefix-free coding*, in Proceedings of 30th Annual European Symposium on Algorithms (ESA), vol. 244, 2022, pp. 57:1–57:5.
4. T. Gagie and Y. Nekrich: *Worst-case optimal adaptive prefix coding*, in Proceedings of the 11th Symposium on Algorithms and Data Structures (WADS), 2009, p. 315–326.
5. R. Gallager: *Variations on a theme by Huffman.* IEEE Transactions on Information Theory, 24(6) 1978, pp. 668–674.
6. M. Karpinski and Y. Nekrich: *A fast algorithm for adaptive prefix coding.* Algorithmica, 55(1) 2009, p. 29–41.
7. D. Knuth: *Dynamic Huffman coding.* Journal of Algorithms, 6(2) 1985, p. 163–180.
8. M. Liddell and A. Moffat: *Decoding prefix codes.* Software Practice and Experience, 36 2006, pp. 1687–1710.
9. R. L. Milidiú, E. S. Laber, and A. A. Pessoa: *Bounding the compression loss of the FGK algorithm.* Journal of Algorithms, 32(2) 1999, p. 195–211.
10. E. S. Schwartz and B. Kallick: *Generating a canonical prefix encoding.* Communications of the ACM, 7 1964, pp. 166–169.
11. J. S. Vitter: *Design and analysis of dynamic Huffman coding*, in Proceedings of the 26th Symposium on Foundations of Computer Science (FOCS), 1985, p. 293–302.
12. I. O. Zavadskyi: *Fast practical adaptive encoding on large alphabets*, in Data Compression Conference, DCC 2025, Snowbird, Utah, USA, 2025, pp. 141–150.