SN Computer Science (2023) 4:265
https://doi.org/10.1007/s42979-022-01629-5

ORIGINAL RESEARCH

Weighted Burrows–Wheeler Compression

Aharon Fruchtman[1] · Yoav Gross[1] · Shmuel T. Klein[2] · Dana Shapira[1]

Received: 12 September 2022 / Accepted: 20 December 2022 / Published online: 17 March 2023

Abstract

A weight-based dynamic compression method has recently been proposed, which is especially suitable for the encoding of files with locally skewed distributions. Its main idea is to assign larger weights to closer to be encoded symbols by means of an increasing weight function, rather than considering each position in the text evenly. A well known transformation that tends to convert input files into files with a more skewed distribution is the *Burrows–Wheeler Transform* (BWT). This paper proposes to apply the weighted approach on Burrows–Wheeler transformed files. While it is shown that the compression performance is not altered for static and adaptive arithmetic coding by *any* permutation of the symbols, hence in particular for BWT, empirical evidence of the efficiency of the combination of BWT with the weighted approach is provided.

Keywords Adaptive compression · Huffman code · Arithmetic code · Burrows-Wheeler Transform

Introduction

The *Burrows–Wheeler Transform* (BWT) [1] is the basis of the popular compression method bzip2, yielding, on many types of possible input files, better compression than gzip and other competitors. As a matter of fact, BWT itself is not a compression method: its output is a permutation of its input, which has obviously the same size. The usefulness of the transformation is that it has a tendency to reorganize the data into what seems to be a more coherent form, grouping many, though not all, identical characters together. The output is therefore usually more compressible, by applying as simple methods as run-length coding and move-to-front.

This article is part of the topical collection "String Processing and Combinatorial Algorithms guest edited by Simone Faro.

✉ Dana Shapira
shapird@g.ariel.ac.il

Aharon Fruchtman
aralef@gmail.com

Yoav Gross
yodgimmel@gmail.com

Shmuel T. Klein
tomi@cs.biu.ac.il

[1] Department of Computer Science, Ariel University, Ramat HaGolan st. 65, 40700 Ariel, Israel

[2] Department of Computer Science, Bar Ilan University, 52900 Ramat-Gan, Israel

The combination of different methods to be applied one after the other, an action known as *cascading*, is not new to data compression. It can, for example, be found in gzip, which first parses the input using LZ77 [2], and then applies Huffman coding [3] to the parsed elements. It therefore seems natural to try to apply BWT in a pre-processing stage, and then compress the transformed string by means of more sophisticated compression schemes, as done, e.g., in bzip2. This strategy fails, however, for static or dynamic arithmetic coding as shown in the following sections. An additional contribution of this paper is to give empirical evidence that cascading BWT with the recently developed *weighted* compression schemes implies significant savings. A first version of this work has appeared in [4].

Given a text T of length n, occupying $O(n/\log n)$ machine words, the construction of the BWT algorithm proposed by Hon et al. [5] runs in linear time and $O(n/\log n)$ space. Recently, a sub-linear running time for sufficiently small alphabets has been proposed by Kempa and Kociumaka [6], in which the BWT is constructed in $O(n/\sqrt{\log n})$ time and $O(n/\log n)$ space.

There are a number of other reversible transformations that are suitable to be used after BWT instead of *Move To Front* (MTF) of Bentley et al. [7, 8], like the *Inversion Frequencies technique*, introduced by Arnavut and Magliveras [9], or *Distance Coding* proposed by Edgar Binder [10]. Gagie and Mansini [11] analyze the efficiency of MTF, Distance Coding and Inversion Frequencies after BWT and provide simple variants of these techniques that achieve the entropy bound. Their analysis strongly relies on the fact that the output of all the involved techniques is composed of integers, which is not necessarily the case for the output of the weighted technique, to be shown below.

A family of dynamic compression algorithms, named *weighted coding*, has recently been proposed [12], which is especially suitable for the encoding of files with locally skewed distributions. The main idea of the weighted approach is to assign larger weights to closer to be encoded symbols by means of an increasing weight function, rather than considering each position in the text evenly, similarly to the weights assigned in the structured arithmetic coding of Fenwick [13].

Traditional dynamic algorithms use the distribution of the symbols in the already processed portion of the input file as an estimate for the distribution of the elements still to come later in the text. However, the assumption that the past is a good approximation for the future, is not necessarily true. A *Forward-looking* dynamic algorithm, using the true distribution of the remaining portion of the file, was suggested in [14]. In this method the frequencies of the symbols in the entire file are prepended to the compressed file. In the encoding process, these frequencies are gradually updated to reflect the number of occurrences in the remaining part of the file by *decrementing* the frequency of the character that is currently being processed. A hybrid method, combining both traditional and forward-looking approaches, is proposed in [15]: in this method the frequencies for all characters are not transmitted at the beginning of the file but rather progressively, each time a new character is encountered.

Pushing this approach even further, a family of *Forward weighted* coding schemes is proposed in [16], in which the encoding is based on the distribution derived from index-dependent weights. That is, this weighted method assigns higher priorities to positions that are closer to the currently processed one in the encoding process, rather than treating all positions in the input file equally. The weights assigned to the positions are generated by a non increasing function f, and the weight for each symbol σ is the sum of the values of f over all the positions where σ occurs in the portion of the input file that is still to be encoded.

Recently, a specific variant named the *Backward Weighted* coding has been studied [12], which suggests a heuristic based on a weighted distribution, calculated only over positions that have already been processed. The core advantage of such a heuristic approach is a negligible header, relatively to a costly header used in *Forward Looking* and all *Forward Weighted* variants. Empirical tests have shown that backward weighted techniques can improve beyond the lower bound given by the entropy for static encoding. In [17], forward weighted coding has been adapted to work with Cleary and Witten's *Prediction by partial matching* (PPM) [18].

The paper is constructed as follows. The section "Notation and Discussion" reviews the notation and formulation needed for the weighted encodings using a running example and discusses the cascading with BWT. The section "Properties" proves some properties involving the combination of general dynamic coding techniques and BWT. The section "Experimental Results" presents empirical outcomes supporting the compression efficiency of the proposed method even in practice.

Notation and Discussion

Weighted Coding

For the completeness of the paper, we include the definitions given in [16], which formalize entropy based compression methods.

Let $T = T[1, n]$ be an input file of size n over an alphabet Σ of size m. A weight W is defined based on the following parameters,

- A non-negative function g, $g : [1, n] \longrightarrow \mathbb{R}^+$, which assigns a positive real number to integers, seen as an assignment of a weight to each position $i \in [1, n]$ within T;
- A symbol of the alphabet, $\sigma \in \Sigma$;
- An interval $[\ell, u]$, $1 \le \ell \le u \le n$ for restricting the domain of the function g.

The value of $W(g, \sigma, \ell, u)$ is defined for each symbol σ, as the sum of the values of the function g for all positions j in the range $[\ell, u]$ at which σ occurs. Formally,

$$W(g, \sigma, \ell, u) \;=\; \sum_{\{\ell \le j \le u \;\mid\; T[j]=\sigma\}} g(j).$$

As a special case of weighted coding, *Backward Weighted* considers all the positions that have already been processed, that is, the interval is of the form $[\ell, u] = [1, i-1]$, and

$$W(g, \sigma, 1, i-1) \;=\; \sum_{\{1 \le j \le i-1 \;\mid\; T[j]=\sigma\}} g(j).$$

Traditional encoding methods, such as static and dynamic Huffman or arithmetic coding [19], can be reformulated as special instances of W for which $g = \mathbb{1} \equiv g(i) = 1$ for all i. For example, static compression refers to weights for which $W(g, \sigma, \ell, u) = W(\mathbb{1}, \sigma, 1, n)$ is constant for all indices.

As a short illustration for the weighted approach, Table 1 brings a comparative chart for the encoding of a small example of 50 characters:

$$T = x_1 \cdots x_{50} = (\mathtt{at})^7(\mathtt{cg})^{11}(\mathtt{at})^7,$$

shown in the second row of the table for several representative portions of T, just underneath the indices. The static compression for T considers the probabilities $\frac{14}{50}$ for $\mathtt{a}$ and

Table 1 Coding example for $T = (\mathtt{at})^7(\mathtt{cg})^{11}(\mathtt{at})^7$

	i	1	2	3	4	⋯	13	14	15	16	⋯	35	36	37	38	⋯	49	50	avg
	T	a	t	a	t	⋯	a	t	c	g	⋯	c	g	a	t	⋯	a	t	
static	W	14	14	14	14	.	14	14	11	11	⋯	11	11	14	14	⋯	14	14	
	TotIndx	50	50	50	50	.	50	50	50	50	⋯	50	50	50	50	⋯	50	50	1.990
	IC	1.84	1.84	1.84	1.84	⋯	1.84	1.84	2.18	2.18	⋯	2.18	2.18	1.84	1.84	⋯	1.84	1.84	(+0.291)
b-adp	Indx$W$	1	1	1	1	⋯	1	1	1	1	⋯	1	1	1	1	⋯	1	1	
	W	1	1	2	2	⋯	7	7	1	1	⋯	11	11	8	8	⋯	14	14	
	TotIndx	4	5	6	7	⋯	16	17	18	19	⋯	38	39	40	41	⋯	52	53	2.111
	IC	2.00	2.32	1.58	1.81	⋯	1.19	1.28	4.17	4.25	⋯	1.79	1.83	2.32	2.36	⋯	1.89	1.92	
f-adp	Indx$W$	1	1	1	1	⋯	1	1	1	1	⋯	1	1	1	1	⋯	1	1	
	W	14	14	13	13	⋯	8	8	11	11	⋯	1	1	7	7	⋯	1	1	
	TotIndx	50	49	48	47	⋯	38	37	36	35	⋯	16	15	14	13	⋯	2	1	1.820
	IC	1.84	1.81	1.88	1.85	⋯	2.25	2.21	1.71	1.67	⋯	4.00	3.91	1.00	0.89	⋯	1.00	0.00	(+0.291)
b-2	Indx$W$	1	1	1	1	⋯	4	4	4	8	⋯	64	128	128	128	⋯	512	512	
	W	1	1	2	2	⋯	12	13	1	1	⋯	261	281	16	17	⋯	1552	1809	1.981
	TotIndx	4	5	6	7	⋯	27	31	35	39	⋯	575	639	767	895	⋯	4095	4607	
	IC	2.00	2.32	1.58	1.81	⋯	1.17	1.25	5.13	5.29	⋯	1.14	1.19	5.58	5.72	⋯	1.40	1.35	
b-weight	Indx$W$	1.00	1.15	1.32	1.52	⋯	5.28	6.06	6.96	8.00	⋯	111.43	128.00	147.03	168.90	⋯	776.05	891.44	
	W	1.00	1.00	2.00	2.15	⋯	14.39	16.38	1.00	1.00	⋯	327.96	376.58	19.67	22.44	⋯	1988.36	2283.88	1.989
	TotIndx	4.00	5.00	6.15	7.47	⋯	32.77	38.05	44.11	51.08	⋯	746.65	858.08	986.08	1113.11	⋯	5216.21	5992.26	
	IC	2.00	2.32	1.62	1.80	⋯	1.19	1.22	5.46	5.67	⋯	1.19	1.19	5.65	5.66	⋯	1.39	1.39	

t and $\frac{11}{50}$ for c and g. These probabilities can be calculated from the two first rows corresponding to the static method in Table 1, the first entitled W representing the weight of the specific character, and the second entitled TotIndx representing the cumulative weights of all the characters. The ratio of these weights can be considered as a probability p_i, and the corresponding *Information content* in bits, $-\log p_i$ for each position i, is shown in the line entitled by its abbreviation IC. For static, the IC values are 1.84 for a and t and 2.18 for c and g. The last column of the table, headed avg, gives the average of these IC values, which is in fact the *entropy*. In a practical application, the entropy can be reached by arithmetic coding and can be approached by Huffman coding. We concentrate in this paper only on arithmetic coding.

The classic adaptive coding, b-adp [20], is a specific backward weight method in which $g(i) = 1$ for all i, and the backward weights refer to all positions i with $1 \leq i < n$ by:

$$W(\mathbb{1}, \sigma, 1, i-1) = \sum_{\{1 \leq j \leq i-1 \ | \ T[j]=\sigma\}} 1,$$

which is simply the number of occurrences of the current character σ up to that point, i.e., in $T[1, i-1]$. The details appear in the second part of Table 1, headed b-adp. The line entitled W refers now, at position i, to the specific weight of the character $\sigma = T[i]$ up to the given column i of the table, that is, the sum of the index weights Indx$W$ for those indices $j < i$ at which the character σ occurs, including the default values that are set to 1 at initialization. For b-adp, as well as for the following method f-adp, the values of Indx$W$ are just 1 for every i. The cumulative W values for all the characters $\sigma \in \Sigma$ are given in the row entitled TotIndx.

The symmetric counterpart of b-adp is the *forward looking* method, f-adp, which, at each location i, considers the positions yet to come $[i, n]$ rather than those already processed $[1, i-1]$ as the *backward looking* b-adp. That is,

$$W(\mathbb{1}, \sigma, i, n) = \sum_{\{i \leq j \leq n \ | \ T[j]=\sigma\}} 1.$$

For our running example, f-adp initializes the weights of the characters a, t to 14 and c, g to 11. The counts are then gradually decremented reflecting the remaining number of occurrences for each σ from position i to the end of T. The header for f-adp describes the exact frequencies of the involved characters, and its size is 0.291 for our example, as for static.

A simple adaptive weighted coding, denoted by b-2, has been proposed in [12]. It is inspired by Nelson and Gailly [21], who rescale the frequencies in order to cope with hardware constraints like the representation of the number of occurrences as 16-bit integers. b-2 divides all the frequencies at the end of every block of k characters, for a given parameter k, regardless of computer hardware restrictions. That is, the occurrences of characters at the beginning of the input file contribute to W less than those closer to the current position. Furthermore, all positions within the same block contribute equally to W, and their contribution weight is twice as large as the weight assigned to the indices in the preceding block. Formally, the function g for b-2, denoted by $g_{\text{b-2}}$, is defined as

$$g_{\text{b-2}}(i) = 2^{\lfloor \frac{i-1}{k} \rfloor},$$

so that for each pair of indices i and $i+k$, it maintains the equation

$$g_{\text{b-2}}(i+k) = 2\ g_{\text{b-2}}(i).$$

The first line of the block headed b-2 shows the index weight, Indx$W$, chosen here with parameter $k = 5$. Starting with 1, the value doubles after each block of 5 positions. The other lines, entitled W, TotIndx and IC, are then defined as above.

A refinement of b-2, named b-weight, is another special weighted coding [12] inspired by the division by 2, but based on the function

$$g_{\text{b-weight}}(i) = (\sqrt[k]{2})^{i-1} \qquad \text{for} \quad i \geq 1,$$

for a given parameter k. The function $g_{\text{b-weight}}$ still maintains a fixed ratio of 2 between blocks but with a smooth hierarchy between all indices, rather than sharp differences at block boundaries. The ratio of 2 for indices that are k apart can be seen by:

$$\begin{aligned} g_{\text{b-weight}}(i+k) &= (\sqrt[k]{2})^{i+k-1} = (\sqrt[k]{2})^{k} \cdot (\sqrt[k]{2})^{i-1} \\ &= 2 \cdot g_{\text{b-weight}}(i). \end{aligned}$$

The index weight Indx$W$ for the b-weight method consists of real numbers rather than integers as above. The shown values correspond to $k = 5$, so that $g_{\text{b-weight}}(i) = (\sqrt[5]{2})^{i-1} = 1.149^{i-1}$. This yields an average codeword length of 1.989 bits per symbol.

The weighted approach should be applied only on a text that has skewed probability distributions in different portions of the file: there is a price for adjusting the model in the transition between regions of different distributions, and this overhead gets negligible only when the text becomes long enough, or if the difference between the distributions is sufficiently sharp as in this short example.

Table 2 Storage requirements of the encoding methods on $\mathsf{BWT}(T) = \mathtt{t}^7\mathtt{gt}^6\mathtt{a}^{14}\mathtt{g}^{10}\mathtt{tc}^{11}$

	T			$\mathsf{BWT}(T)$		
	k	Header	bps	k	Header	bps
Static	–	0.291	2.281	–	0.404	2.394
b-adp	–	–	2.111	–	0.113	2.224
f-adp	–	0.291	2.111	–	0.404	2.224
b-weight	5	–	1.989	3	0.113	1.567
b-2	5	–	1.981	3	0.113	1.562

Cascading with BWT

One of the features of the BWT is that it has a tendency to reorganize the text T, such that $\mathsf{BWT}(T)$ contains several runs of repeated characters. In particular, for our running example

$$T = \mathtt{atatatatatatatcgcgcgcgcgcgcgcgcgcgcgatatatatatatat} = (\mathtt{at})^7(\mathtt{cg})^{11}(\mathtt{at})^7,$$

after applying the BWT, the transformed text is

$$\mathsf{BWT}(T) = \mathtt{tttttttgttttttaaaaaaaaaaaaaaggggggggggtccccccccccc} = \mathtt{t}^7\mathtt{gt}^6\mathtt{a}^{14}\mathtt{g}^{10}\mathtt{tc}^{11},$$

even though in the original T, there is not a single pair of identical adjacent characters. This tendency implies that the local distributions seem more skewed, which is advantageous to the weighted approach.

We applied the same 5 methods on $\mathsf{BWT}(T)$. A small amendment is necessary because the transformed string $\mathsf{BWT}(T)$ on its own is not reversible—it needs in addition a pointer to a starting point, actually the index of the last character of T within $\mathsf{BWT}(T)$, which requires $\log n$ bits, or $\frac{\log n}{n}$ per symbol. We thus included this additional overhead of $\frac{\log 50}{50} = 0.113$ bits in the header. Note that this overhead is constant for all methods. On the other hand, the static and the forward compression methods require information about the distribution probabilities, which should be prepended to the compressed file independently from applying BWT or not. We use the lower bound approximation given by the IC to express this information in this example. Yet, for our experiments, we encoded the meta data information by means of the *universal* Elias [22] γ-code.

The comparison of the final compression results, before and after applying the BWT, is presented in Table 2 and includes the parameter k that achieves the best performance of b-2 and b-weight, the header size and the corresponding total storage costs including the header in bits per symbol (bps). The first three columns refer to the encoding variants on the original file T, and the last three columns are the results on $\mathsf{BWT}(T)$. As can be seen, while the improvement of the weighted methods with BWT is about 21%, the net encoding, excluding the header size, is the same for static, b-adp and f-adp, regardless whether BWT has been applied or not. The total bps results, including the header size, are identical for b-adp and f-adp, because the gain in the net compression by f-adp is exactly the same as the loss incurred by the additional overhead due to the exact frequencies. In fact, these are not coincidences, and in the following section we show that the compression performance is preserved for static, b-adp and f-adp under *any* permutations of the symbols of T, hence in particular for BWT. Moreover, we show that the size of the compressed file including the header for b-adp and f-adp is the same.

Figures 1 and 2 are visualizations of the differences between the methods, plotting for each method the information content as a function of the position i for our running example. The net encoding results before (Fig. 1) and after (Fig. 2) applying BWT are depicted in matching colors. We see that the fluctuations are much more accentuated after applying BWT. Figure 3 displays the cumulative values of the same data. As can be seen from both figures, the backward weighted methods are more sensitive to fluctuations in the distribution, but adjust faster to changes. The final points of the accumulated values for the traditional methods differ only by the size of the header, and the advantage of BWT for b-2 and b-weight can be seen by the fact that the corresponding plots are below their counterparts without BWT. The differences between b-2 and b-weight, with and without BWT, are so small that their plots seem to be overlapping.

Properties

We show in this section that for the three first mentioned methods, static, b-adp and f-adp, applying BWT, or any other permutation, does not have any influence of the compression by arithmetic coding. The next section then brings empirical evidence that on the other hand, for the weighted methods, a pre-processing stage by BWT does significantly improve the compression performance.

Arithmetic coding starts with an interval [0,1), and repeatedly narrows it as the text T is being processed. The narrowing procedure is a proportional refinement of the present interval into sub-portions according to the probability

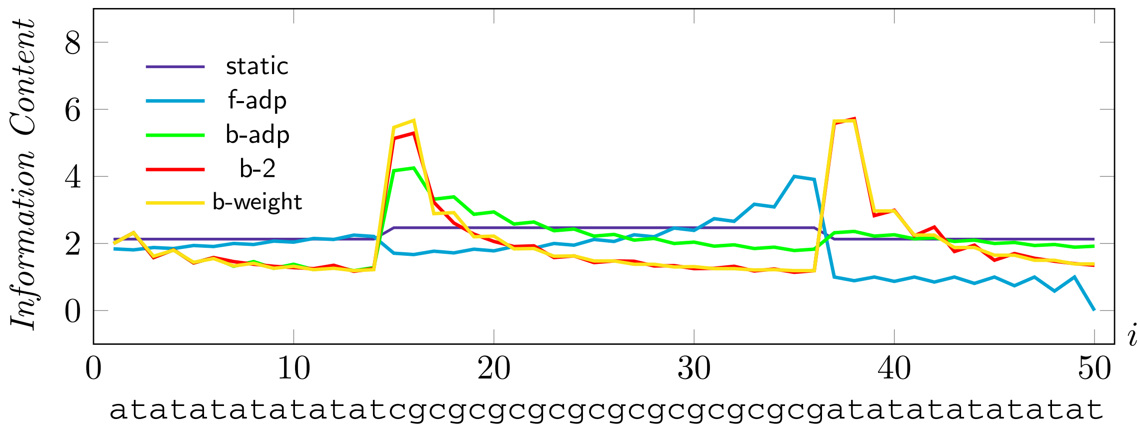

Fig. 1 Information content per index on the original text $T = (\mathtt{at})^7(\mathtt{cg})^{11}(\mathtt{at})^7$

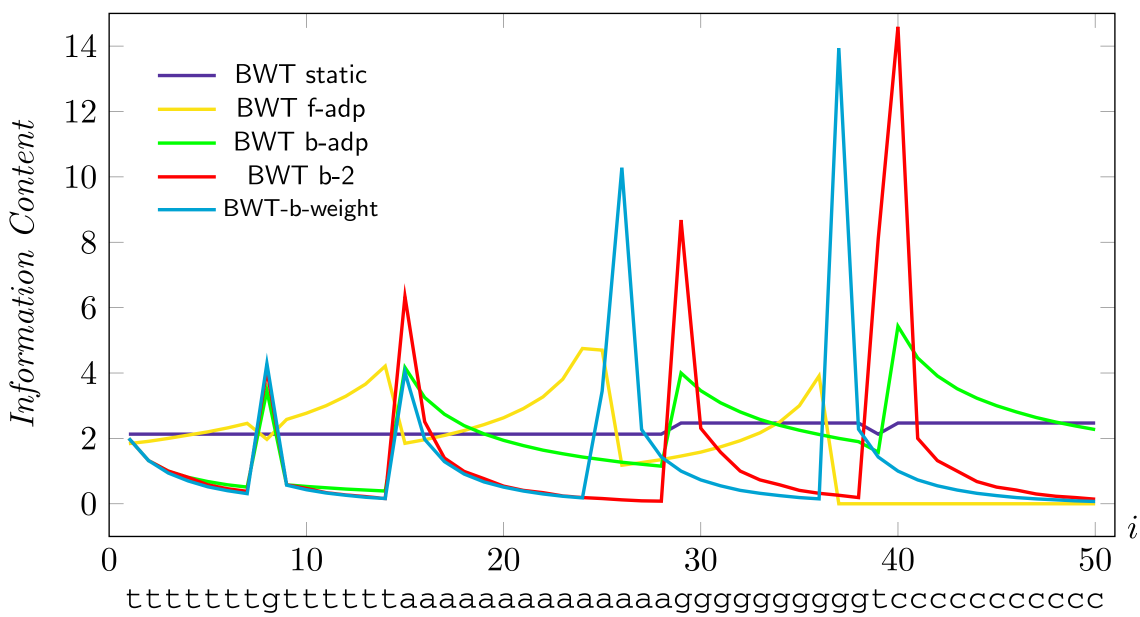

Fig. 2 Information content per index on $\mathsf{BWT}(T) = \mathtt{t}^7\mathtt{g}\mathtt{t}^6\mathtt{a}^{14}\mathtt{g}^{10}\mathtt{t}\mathtt{c}^{11}$

distribution of the symbols of Σ. The encoding is a real number that can be selected randomly within the final interval. Static arithmetic coding uses a fixed probability distribution throughout the process for the interval partition, while the (backward) adaptive method updates the proportions for the corresponding partitions according to what has already been seen. It is well known that the static variant achieves the entropy of order 0. A straightforward corollary of this fact is that permuting T does not change the *size* of the output, though the output file itself will of course be altered. This is stated in the following lemma.

Lemma 1 *The size of the compressed file, after having applied* static *arithmetic coding, is invariant under permutations of the original input.*

Proof Suppose we are encoding the text $T[1, n]$ using the static arithmetic method. The notation of the weight introduced above for $\sigma = T[i]$, $W(\mathbb{1}, T[i], 1, n)$, refers to the number of occurrences of $T[i]$ within $T[1, n]$. For simplicity, we shall use $\mathsf{occ}(\sigma)$ to denote $W(\mathbb{1}, \sigma, 1, n)$.

Each processed letter $T[i]$ narrows the current sub-interval of $[0, 1)$ by a factor equal to the probability of $T[i]$ in T, that is, by $\frac{1}{n}W(\mathbb{1}, T[i], 1, n)$. The size of the range, r_s, of the

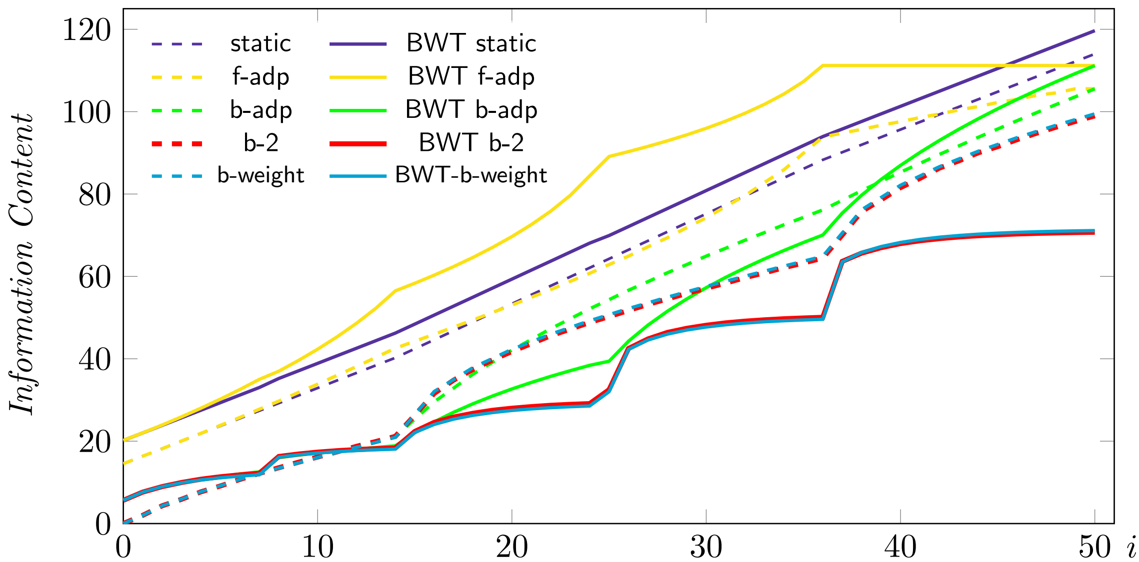

Fig. 3 Accumulated information content as a function of the size of the processed prefix, before and after applying BWT on $T = (\mathtt{at})^7(\mathtt{cg})^{11}(\mathtt{at})^7$

final interval after processing $T = T[1, n]$ by static, is the product of the sizes of these intervals:

$$r_s = \prod_{i=1}^{n} \frac{1}{n} W(\mathbb{1}, T[i], 1, n) = \frac{1}{n^n} \prod_{\sigma\in\Sigma} \prod_{i=1}^{\mathsf{occ}(\sigma)} \mathsf{occ}(\sigma) = \frac{1}{n^n} \prod_{\sigma\in\Sigma} \mathsf{occ}(\sigma)^{\mathsf{occ}(\sigma)}, \tag{1}$$

where the middle equality is obtained by reordering the multiplication factors by character. The size of the compressed file is the information content of arbitrarily choosing a number within an interval of size r_s, which is $-\log_2 r_s$, and it is independent of the order in which the letters appear. □

A similar property can be proven for traditional adaptive arithmetic coding, as follows.

Lemma 2 *The size of the compressed file, after having applied* adaptive *arithmetic coding, is invariant under permutations of the original input.*

Proof To avoid the zero-frequency problem for encoding the first occurrence of a letter σ in T, the number of occurrences for each σ is initialized by 1. For adaptive arithmetic coding, b-adp, each processed symbol $T[i]$, $1 < i \leq n$, narrows the current sub-interval of [0, 1), representing the processed *prefix* $T[1, i-1]$ of T of size $i-1$, by a factor equal to the probability of $T[i]$ in $T[1, i-1]$. This probability is equal to

$$\frac{W(\mathbb{1}, T[i], 1, i-1) + 1}{m + i - 1},$$

where $m = |\Sigma|$ taking the initial 1-values of all the characters into account. Multiplying the sizes of these intervals for all elements $1 \leq i \leq n$, yields the size of the final range r_b, corresponding to b-adp:

$$r_b = \prod_{i=1}^{n} \frac{W(\mathbb{1}, T[i], 1, i-1) + 1}{m + i - 1} = \left(\prod_{i=1}^{n} \frac{1}{m + i - 1}\right) \prod_{\sigma\in\Sigma} \prod_{i=1}^{\mathsf{occ}(\sigma)} i$$
$$= \frac{(m-1)!}{(m+n-1)!} \prod_{\sigma\in\Sigma} \mathsf{occ}(\sigma)!$$

The size of the compressed file is accordingly $-\log_2 r_b$, so that permuting the input text will not change the size of the output. □

A similar proposition can be proven for f-adp, stated as follows.

Lemma 3 *The size of the compressed file, after having applied forward arithmetic coding,* f-adp, *is invariant under permutations of the original input.*

Proof By a similar argument to that of Lemma 2, each processed letter $T[i], 1 < i \leq n$, narrows the current sub-interval, representing the processed *suffix* $T[i, n]$ of T of size $n - i + 1$, by a factor equal to the probability of $T[i]$ in $T[i, n]$, which is equal to

$$\frac{W(\mathbb{1}, T[i], i, n)}{n - i + 1}.$$

Multiplying the sizes of these intervals contributed by the occurrences of $T[i]$, $1 \leq i \leq n$ yields the size of the final range corresponding to f-adp,

$$\begin{aligned} r_f &= \prod_{i=1}^{n} \frac{W(\mathbb{1}, T[i], i, n)}{n - i + 1} = \left(\prod_{i=1}^{n} \frac{1}{n - i + 1} \right) \prod_{\sigma \in \Sigma} \prod_{i=1}^{\mathrm{occ}(\sigma)} i \\ &= \frac{1}{n!} \prod_{\sigma \in \Sigma} \mathrm{occ}(\sigma)! \end{aligned} \tag{2}$$

As before, the size of the compressed file is $-\log_2 r_f$, which is not affected by permutation of the input file. □

Besides proving that the order of the characters does not matter, only their quantity, we also get an exact evaluation of the difference in size between the encoded files:

Corollary 4 *The forward looking encoding,* f-adp, *is better than the backward looking encoding,* b-adp *by* $\log \binom{m+n-1}{n}$ *bits, where n is the size of the input file T, and m is the size of its alphabet.*

Proof The difference between the sizes of the compressed files of b-adp and f-adp is as follows.

$$\begin{aligned} -\log r_b + \log r_f &= \log \left(\frac{1}{n!} \prod_{\sigma \in \Sigma} \mathrm{occ}(\sigma)! \right) \\ &\quad - \log \left(\frac{(m-1)!}{(m+n-1)!} \prod_{\sigma \in \Sigma} \mathrm{occ}(\sigma)! \right) \\ &= \log \left(\frac{(m+n-1)!}{n! \cdot (m-1)!} \right) = \log \binom{m+n-1}{n}. \end{aligned}$$

□

Interestingly, the difference between b-adp and f-adp only depends on the size of the alphabet and the size of the file and is blind to the content itself. In fact, on our datasets with n about 4 million and $m = 257$, including the EOF sign, the difference between the compressed files, discarding the prelude was the constant 494 bytes, as expected. However, as mentioned above, the prelude for f-adp is more costly than that for b-adp , as it must include the exact frequencies of the characters.

The number of sequences $\left(occ(\sigma_1), occ(\sigma_2), \ldots, occ(\sigma_m)\right)$ such that $\sum_{i=1}^{m} occ(\sigma_i) = n$ is equal to the number of ways to select n numbers out of $n + m - 1$, i.e., $\binom{n+m-1}{n}$. The information content of selecting one of these choices, $\log \binom{n+m-1}{n}$, coincides therefore with the difference in encoding size between f-adp *and* b-adp , which shows that the gain in compression efficiency is in fact canceled out by the increased size of the prelude. We can therefore conclude that, even though the net encoding by f-adp is always better than that by b-adp :

Corollary 5 *The total size of the encoding (including the header) by forward looking*, f-adp, *is the same as that of the backward looking encoding,* b-adp.

Another immediate consequence of the above is that f-adp is better than static , a fact that was already proven for Huffman coding in [14], *and here for arithmetic coding.*

Proof The difference between the forward looking and static algorithms may be emphasized by considering both techniques as different variants of the following experiment. Imagine a pool $\mathcal{P}$ of characters of Σ, which initially contains $occ(\sigma)$ copies of the character $\sigma \in \Sigma$; we then apply n iterations in each of which one element is randomly chosen from the pool $\mathcal{P}$. The question is whether or not the already drawn elements are returned to $\mathcal{P}$. If yes, the experiment corresponds to static and the probability of reaching a specific sequence is r_s as shown by Eq. (1); if the elements are not returned to $\mathcal{P}$, the experiment corresponds to forward and the probability of reaching a specific sequence is r_f as given by Eq. (2). Considering the sequences of characters obtained by this experiment as its outcomes, the set of possible sequences in the latter (forward) case is a proper subset of the set of possible sequences in the former (static) case; thus $r_s < r_f$ and therefore $-\log r_s > -\log r_f$. □

Experimental Results

In order to study the performance of the weighted methods on BWT transformed texts on real-life, rather than artificial data, we considered the datasets from the Pizza & Chili corpus,[1] a collection of files of different nature and alphabets. All algorithms were implemented in C++, and the code can be found at https://www.ariel.ac.il/wp/dana-shapira/code/

[1] http://pizzachili.dcc.uchile.cl/.

Table 3 Compression performance on original and BWT transformed files (%) of the different methods

Datasets	Static	adp	BWT + RLE	b-2				b-weight			
				T	(k)	BWT	(k)	T	(k)	BWT	(k)
SOURCES	69.48	69.48	27.54	64.20	(444)	26.44	(24)	64.17	(438)	26.30	(22)
XML	65.54	65.54	16.77	65.27	(9355)	13.11	(24)	65.27	(8799)	13.05	(24)
DNA	24.97	24.98	36.69	24.64	(62)	23.02	(34)	24.64	(60)	23.01	(34)
ENGLISH	57.07	57.07	42.28	56.55	(4352)	29.41	(36)	56.54	(3230)	29.33	(36)
PITCHES	69.32	69.32	59.68	56.17	(85)	53.34	(119)	56.09	(83)	53.26	(120)
PROTEINS	52.64	52.65	44.12	51.69	(329)	46.71	(39)	51.68	(317)	46.62	(38)

weightedBWT/ password: Yoav. We compared all methods before and after BWT has been applied. For comparison, we included the compression results of applying simple run-length encoding, RLE, on the BWT transformed file. RLE considers the text as an alternating sequence of runs of characters and encodes it as (character, length) pairs. We used the *minimal binary code* of [23] (an almost fixed length code with codewords of lengths $\lceil \log_2 m \rceil$ or $\lceil \log_2 m \rceil - 1$) to encode the alphabet symbols. Because the encoding of single character occurrences was too expensive by Elias codes [22], we used the following variant that yielded a significant improvement. A single bit indicated whether the length ℓ of the current run is longer than 1 or not. Only in the former case, $\ell - 1$ was encoded by Elias' γ-code.

Compression Performance

For our experiments we used a 4MB prefix of each of our dataset files. Table 3 shows the compression performance on the original and transformed files, defined as the size of the compressed file divided by the size of the original file, in percent. The column headed adp refers to both b-adp and f-adp, which give identical results, as expected by Corollary 5. As static and adp are not affected by BWT by Lemma 1 and Lemma 3, their results are reported only once, in the second and third column, respectively. Our RLE implementation is reported in the fourth column. The following four columns refer to b-2 and the last four columns to b-weight. The values of k used to achieve the best results of b-2 and b-weight are displayed in parentheses. The optimal values of k were derived empirically for each test file, using a logarithmic search in an initially unbounded range.

As can be seen, b-2 and b-weight outperform the static and adaptive variants on all datasets even on the original file T. While they are better by up to 5% when applied on the original file, they become better by more than 40% in some of the cases, and about 25% on the average, when applied on the BWT transformed file. b-weight is better than b-2 by 0.1% on average on all files. On our tests, b-2 and b-weight slightly improve on RLE, except for PROTEINS. While the k values in Table 3 for b-2 and b-weight are those yielding the optimal performance for each of the methods, Figs. 4 and 5 compare b and b-weight for identical k values, showing the file sizes as a function of k, before and after applying BWT. For the given test file, b-weight is smoother and consistently performs better than b-2.

For our final experiment we have combined several methods: RLE, Move-to-front (MTF) and arithmetic coding on the BWT transformed files, with and without our weighted technique. MTF encodes each input symbol σ by its index, counting from zero, in the dynamic list of "recently used symbols". The initial order of the list is given in advance, often using the alphabetic order of the underlying alphabet. In this case, the first symbol is encoded by its own index in the alphabet. After the encoding of each symbol σ, it is moved to the front of the list before continuing to the next symbol. As example, in case $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, then MTF($\mathtt{cccccaa}$) = 2, 0, 0, 0, 0, 1, 0. MTF + RLE applies MTF followed by RLE, e.g., (MTF + RLE)

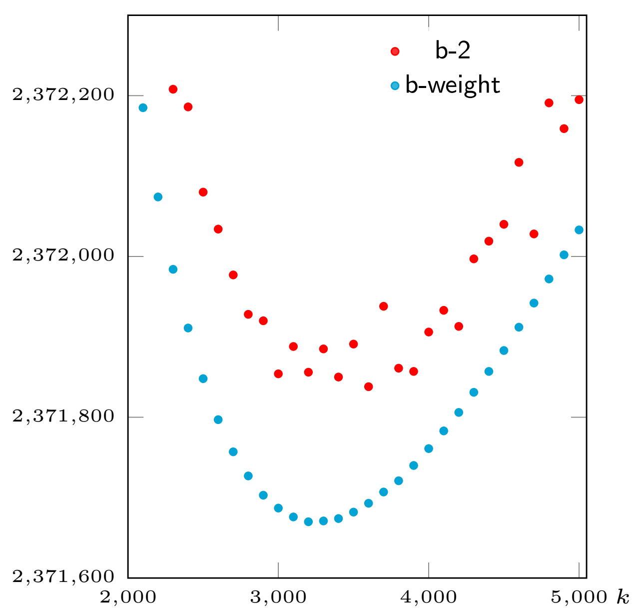

Fig. 4 Comparing b-2 and b-weight for different k values on ENGLISH

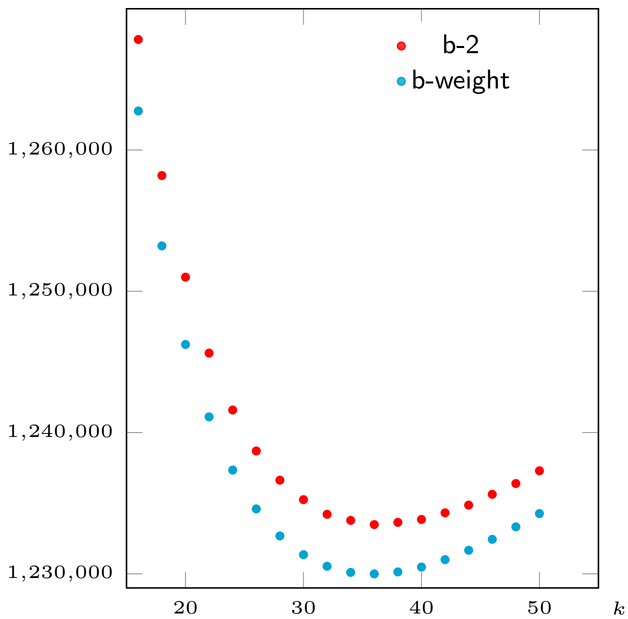

Fig. 5 Comparing b-2 and b-weight for different k values after BWT on ENGLISH

(cccccaa) = (2,1)(0,4)(1,1)(0,1). There are generally long sequences of zeros and ones in the output of MTF [1], which is why in practical applications like bzip2, after having applied BWT and MTF, a variant of RLE known as RLE-0 is used. This variant has been suggested in Burrows and Wheeler's original paper [1] and uses two arithmetic or Huffman codes, the first encoding integers and lengths of runs of zeros, the second encoding the integers immediately following such zero-runs (and thus not including 0 itself). To give an example of RLE-0, suppose the output of MTF starts with the sequence

$$2, 1, 3, \underline{0, 0, 0}, \boxed{4}, 1, \underline{0, 0, 0, 0}, \boxed{1}, 2, \underline{0}, \boxed{5}, \underline{0, 0, 0}, \boxed{7}, 3, \ldots .$$

We use two different Huffman codes: the first $A = \{a_1, a_2, a_3, \ldots\}$ encodes the integers $1, 2, 3, \ldots$, respectively, and the second $B = \{b_1, b_2, b_3, \ldots, R_1, R_2, R_3, \ldots\}$, in which b_i encodes the integer i and R_j represents a run of $j \geq 1$ zeros. Any input can then be parsed into a sequence of elements of A and B, in which a codeword of A is always preceded by a codeword R_j for a run of zeros. The above example sequence, in which runs of zeros have been underlined and elements following such runs are boxed, will then be encoded by

$$b_2, b_1, b_3, R_3, a_4, b_1, R_4, a_1, b_2, R_1, a_5, R_3, a_7, b_3, \ldots .$$

Instead of Huffman coding, one can obviously also use arithmetic coding with the necessary adaptations, as we have done in our experiments.

The compression results for each method are presented in Table 4, using adaptive arithmetic coding alone (adp), or in combination with b-2 and b-weight. Consistently, b-weight slightly improves on b-2, which in turn improves on adp. All the results outperform the RLE based minimal binary code that has been presented in Table 3. The combined (MTF + RLE-0) method gives the best values for all our tests. The compression gain of using RLE-0 instead of RLE was 10–20%.

Weighted BWT **Block Variant**

The Burrows–Wheeler transform is usually applied on blocks, as the name of their technique indicates. Although the division into blocks may improve the running time, it can at the same time significantly damage the compression efficiency. It is indeed possible to perform the transformation on each block and still compress all blocks together. Instead of chaining the transformed blocks in a sequence, we suggest to concatenate original and reversed blocks alternately. BWT approximately sorts the characters in each block. If two consecutive blocks have similarly distributions, applying BWT will produce similar strings. Reversing then every second block will tend to blur the transitions between the blocks.

There are several examples in the data compression literature for alternating original and reversed data in consecutive blocks. As a first example, consider *Gray codes* [24] for generating a cyclic sequence of codewords so that each codeword differs from the previous one by a single bit. The binary code with codewords of length n can be constructed recursively from the code with codeword lengths $n - 1$ bits by reversing the order of the codewords and prefixing the

Table 4 Compression performance on BWT transformed files (%) of the different methods

Datasets	RLE			MTF			MTF + RLE-0		
	adp	b-2	b-weight	adp	b-2	b-weight	adp	b-2	b-weight
SOURCES	24.75	19.99	19.97	21.64	20.12	20.11	18.937	18.519	18.516
XML	14.54	11.22	11.21	15.24	11.38	11.36	10.884	10.399	10.396
DNA	28.65	28.30	28.29	24.37	23.84	28.83	23.680	23.540	23.539
ENGLISH	32.64	26.59	26.57	28.30	26.20	26.19	26.310	25.374	25.367
PITCHES	49.86	45.37	45.34	46.26	45.69	45.69	45.064	44.841	44.836
PROTEINS	40.11	39.96	39.96	41.70	41.31	41.29	38.570	38.542	38.542

Table 5 Block compression methods with block size of 8KB

	SOURCES	XML	DNA	ENGLISH	PITCHES	PROTEINS
b-2						
DBC	48.54	43.56	24.26	49.94	53.80	50.93
DBMC	47.84	43.15	24.12	49.57	53.86	50.15
DBIMC	47.63	42.96	24.11	49.29	53.61	50.13
b-weight						
DBC	48.27	43.19	24.25	49.79	53.67	50.86
DBMC	47.57	42.86	24.12	49.43	53.75	50.06
DBIMC	47.37	42.66	24.10	49.15	53.51	50.05

original and reversed codewords with a 0 and 1 bit, respectively. A second example, in the area of image compression, is the extension of a signal to a periodic wave, using the cosine and sine transforms, see, e.g., Chapter 10.3 in [25].

As a simple example consider the outputs `aacbbcacd–cdd`, `aaabdbacccdd` and `aababbbccddd` of three consecutive blocks. Instead of concatenating them directly and compressing the result, the output of the second block is reversed. For this example we get

`aacbbcacdcdd–ddcccabdbaaa–aababbbccddd`,

and the transition between the blocks, emphasized here in red, becomes smoother. Moreover, similar runs, indicated by matching colors, tend to be moved closer together. Table 5 compares the following block methods with block size 8KB.

- DBC, *Divide-BWT-Compress*, divides the input file into blocks and applies separately BWT followed by b-2 or b-weight compressors on each block;
- DBMC, *Divide-BWT-Merge-Compress*, divides the input file into blocks, applies BWT to each block, merges the BWT outcomes into a single file and compresses the resulting file;
- DBIMC, *Divide-BWT-Inverse-Merge-Compress*. The same as DBMC, except for merging consecutive blocks in which every second block is reversed.

As can be seen, there is mostly a slight improvement of DBMC over DBC, and DBIMC gives consistently an additional small gain.

Conclusion

This paper studies the compression performance of weighted coding on Burrows–Wheeler transformed files. We have shown that statistical methods which treat all positions in the files evenly are indifferent to permutations in the input file, and to BWT in particular. On the other hand, the weighted approach, being more suitable to skewed files, has been shown empirically to gain additional savings when applied after having pre-processed the text by BWT.

Data availability All data generated or analysed during this study are included in this published article.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Burrows M, Wheeler D.J. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
2. Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE Trans Inf Theory. 1977;23(3):337–43.
3. Moffat A. Huffman coding. ACM Comput Surv. 2019;52(4):85–18535.
4. Fruchtman A, Gross Y, Klein ST, Shapira D. Weighted Burrows-Wheeler compression. CoRR abs/2105.10327 (2021)
5. Hon W, Sadakane K, Sung W. Breaking a time-and-space barrier in constructing full-text indices. SIAM J Comput. 2009;38(6):2162–78.
6. Kempa D, Kociumaka T. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: Charikar M, Cohen E, editors. Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23–26; 2019. p. 756–767.
7. Bentley JL, Sleator DD, Tarjan RE, Wei VK. A locally adaptive data compression scheme. Commun ACM. 1986;29(4):320–30.
8. Ryabko BY, Horspool RN, Cormack GV. Comments to: a locally adaptive data compression scheme. Commun ACM. 1987;30(9):792–4.
9. Arnavut Z, Magliveras SS. Block sorting and compression. In: Storer JA, Cohn M, editors. Proceedings of the 7th Data Compression Conference (DCC '97), Snowbird, Utah, USA, March 25–27; 1997. p. 181–190.
10. Binder E. Distance coder. Usenet group: comp.compression. 2000. http://groups.google.com/group/comp.compression/msg/27d46abca0799d12.
11. Gagie T, Manzini G. Move-to-front, distance coding, and inversion frequencies revisited. Theor Comput Sci. 2010;411(31–33):2925–44.
12. Fruchtman A, Gross Y, Klein S.T, Shapira D. Backward weighted coding. In: 31st Data Compression Conference, DCC 2021, Snowbird, UT, USA, March 23–26; 2021. p. 93–102.

13. Fenwick PM. The Burrows-Wheeler transform for block sorting text compression: principles and improvements. Comput J. 1996;39(9):731–40.
14. Klein ST, Saadia S, Shapira D. Forward looking Huffman coding. Theory Comput Syst. 2020;65(3):593–612.
15. Fruchtman A, Klein S.T, Shapira D. Bidirectional adaptive compression. In: Proceedings of the Prague Stringology Conference; 2019. pp. 92–101.
16. Fruchtman A, Gross Y, Klein ST, Shapira D. Weighted forward looking adaptive coding. Theor Comput Sci. 2022;930:86–99.
17. Avrunin RM, Klein ST, Shapira D. Combining forward compression with PPM. SN Comput Sci. 2022;3(3):239.
18. Cleary J, Witten I. Data compression using adaptive coding and partial string matching. IEEE Trans Commun. 1984;32(4):396–402.
19. Witten IH, Neal RM, Cleary JG. Arithmetic coding for data compression. Commun ACM. 1987;30(6):520–40.
20. Vitter JS. Design and analysis of dynamic Huffman codes. JACM. 1987;34(4):825–45.
21. Nelson M, Gailly J-L. The data compression book. New York: M & T Books; 1996. p. 550–1.
22. Elias P. Universal codeword sets and representations of the integers. IEEE Trans Inf Theory. 1975;21(2):194–203.
23. Moffat A, Turpin A. Compression and Coding Algorithms. The international series in engineering and computer science, vol. 669, Kluwer (2002)
24. Gray F. Pulse code communication. U.S. Patent 2,632,058A, Serial No. 785697 (1953)
25. Hankerson DC, Harris GA, Johnson J. Introduction to information theory and data compression. Boca Raton, Florida: CRC; 1998.