Theory of Computing Systems (2021) 65:593–612
https://doi.org/10.1007/s00224-020-09992-7

Forward Looking Huffman Coding

Shmuel T. Klein[1] · Shoham Saadia[2] · Dana Shapira[2]

Published online: 25 June 2020

Abstract
Huffman coding is known to be optimal, yet its dynamic version may yield smaller compressed files. The best known bound is that the number of bits used by dynamic Huffman coding in order to encode a message of n characters is at most larger by n bits than the size of the file required by static Huffman coding. In particular, dynamic Huffman coding can also generate a larger encoded file than the static variant, though in practice the file might sometimes be smaller. We propose here a new variant of Huffman encoding, that provably always performs better than static Huffman coding by at least $m - 1$ bits, where m denotes the size of the alphabet, and may be better than the standard dynamic Huffman coding for certain files. The algorithm is based on reversing the direction for the references of the encoded elements, from pointing backwards into the past to looking forward into the future.

Keywords Lossless data compression · Huffman coding · Dynamic Huffman coding

1 Introduction

Data Compression is a research field that aims at representing some input data using less storage than originally given; this aim may be achieved by removing redundancies.

This article belongs to the Topical Collection: *Special Issue on Computer Science Symposium in Russia (2019)*
Guest Editor: Gregory Kucherov

✉ Shmuel T. Klein
tomi@cs.biu.ac.il

Shoham Saadia
shohamsaadia@gmail.com

Dana Shapira
shapird@ariel.ac.il

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

[2] Department of Computer Science, Ariel University, Ariel, 40700, Israel

To this end, many encoding algorithms have been suggested e.g., [4, 5, 9, 28]. Research in data compression has been extended to several new paradigms in the last few decades. The following partial list gives a few examples:

1. *Compressed Pattern Matching*, that is, searching for strings directly in the compressed form of the text [1, 26]. This has been extended to images [14, 17] as well as to structured files [3, 15];
2. *Variable-to-fixed length codes*, in which the lengths of the codewords are restricted to be equal, which is useful for fast decoding and compressed matching [16, 29];
3. *Compact Data Structures* [22], that help represent the data in reduced space, while still allowing efficient query handling, navigation, and a predefined set of operations to be performed on it [2, 10, 18, 22, 24];
4. *Compressed Cryptosystems* [19, 27, 32], which combine compression and encryption at the same time.

Huffman coding is one of the cornerstones of data compression algorithms, and enjoys popularity in spite of almost seven decades since its invention, probably because of its well-known optimality. Given is a text $T = x_1 \cdots x_n$ over some alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ such that σ_i occurs w_i times in T. The problem is to assign binary codewords of lengths ℓ_i bits to the characters σ_i, such that the set of codewords forms a *prefix code* and such that the total size of the encoded file in bits, $\sum_{i=1}^{m} w_i \ell_i$, is minimized. We use the common notation $T[1, n]$ for T, and $T[i, j]$ for the substring $x_i \cdots x_j$ of T. In particular, $T[1, j]$ is the prefix of length j of T for any $1 \leq j \leq n$, and $T[i, n]$ is the suffix of length $n - i + 1$ of T for any $1 \leq i \leq n$.

The alphabet will often consist of a set of characters, and it is convenient in many applications to use just the basic ASCII set of 128 or 256 letters, but Huffman's algorithm works just as well for larger sets, such a bigrams, k-grams or even words, and ultimately, any set $\mathcal{S}$ of substrings of the text, as long as there is a well defined way to parse the text into a sequence of elements of $\mathcal{S}$. We shall use the term *alphabet* in this broader sense.

Huffman's seminal paper [9] solves this problem optimally, but it should be remembered that Huffman codes are optimal under the following constraints:

1. the alphabet Σ is given and fixed in advance;
2. the codeword lengths ℓ_i are integers.

In particular, the second condition seems self-evident when one considers the binary codewords of known codes like ASCII, Huffman, Shanon-Fano or others, though it may be circumvented by *arithmetic coding* [33]. By encoding the entire input file as a single element rather than each character individually, an arithmetic encoder effectively assigns to each occurrence of the character σ_i of probability $p_i = w_i/n$ an encoding of exactly $-\log_2 p_i$ bits, without rounding. The average codeword length thus reaches the entropy $-\sum_{i=1}^{m} p_i \log_2 p_i$, and is always at least as good as that of Huffman coding based on the same alphabet.

There is, however, another implicit constraint, which is rarely mentioned because it seems obvious, that

3. the encoding is static, i.e., the same codeword is used for a given character throughout the encoding process.

Data compression algorithms are often classified into static or adaptive techniques. The static ones base the coding procedures on a model of the distribution of the encoded elements that is either assumed in advance or gathered in a first pass over the data, whereas the adaptive methods learn the model details incrementally. Therefore, the static models would seem, at first sight, to be better and produce smaller files as they have the ability to exploit more knowledge: not only the distribution of the elements up to the current point, but global occurrence statistics in the entire file. However, in practice, adaptive compression is sometimes better, in particular when there is much variability in the occurrence of patterns of the different elements on which the model is based. Furthermore, if the model is not learned adaptively, a prelude consisting of the details of the chosen model should be prepended to the compressed file, allowing the decoder to be synchronized with the encoder. For the adaptive methods, transmitting the details of the model to the decoder is superfluous, as the model gets updated identically on both encoder and decoder ends. The adaptive methods are often referred to as *dynamic* ones, and we shall use these terms interchangeably.

In this paper we concentrate on Huffman coding and propose a new encoding approach, that provably always performs better than static Huffman coding. In the following section, we discuss related work. Section 3 describes the main idea of the new approach, followed by the details and an extended example in Section 4. Section 5 first presents an extreme case in which the new approach can be twice as efficient as the dynamic algorithm used so far, and then brings a formal proof that the forward looking variant does always improve the compression performance of static coding, a property that does not hold for classical dynamic Huffman coding. Section 6 suggests two block-wise variants of the new algorithm, and Section 7 brings empirical results.

2 Related Work

Static Huffman coding can use a known distribution of the alphabet corresponding to the nature of the file, for example, English text with its well known character distribution, or use accurate probabilities for the specific input file that are computed via a preprocessing stage. An advantage of using known statistics is saving the preprocessing stage, however, one then relies on the assumption that the given file fits the assumed model, which is not always the case. If there is too much discrepancy between the conjectured and the actual model, there might be a loss in the compression gain.

In the adaptive mode of Huffman coding, the encoder and decoder maintain identical copies of a gradually adjusting Huffman tree, which, at each point, corresponds to the frequencies of the elements processed so far. The trivial solution of reconstructing the Huffman tree from scratch after each processed character, is obviously very wasteful, since in most cases, the Huffman tree is not altered: only one of the frequencies is incremented by 1, and the others remain unchanged. This motivated the development of efficient adaptive Huffman coding procedures by Faller [6] and Gallager [8], who propose essentially the same one-pass solution. Knuth extends Gallager's work and also suggests that the frequencies may be decreased as well as increased [20]. These independent adaptive Huffman coding methods are known as the FGK algorithm.

A further enhancement by Vitter [31] also minimizes the external path length $\Sigma_{i=1}^{m} \ell_i$ and the height $\max\{\ell_i\}$ among all Huffman trees. Vitter proved that the number of bits used in his adaptive Huffman procedure in order to encode a message of n characters, is bounded by the size of the compressed file resulting from the optimal two-pass static Huffman algorithm, plus n. That is, the dynamically produced file may be *larger* than the static counterpart, and examples can be given for which this actually happens, though empirical results sometimes show that on the contrary, there might be an improvement in the compression rate of the dynamic version as compared to the static one. One may thus conclude that in certain cases, though not in all, adaptive Huffman coding is *better than the "optimal"* static Huffman coding!

This work is an extended version of a paper that has been presented in 2019 at the 14th International Computer Science Symposium in Russia (CSR'19) in Novosibirsk, and appeared in its proceedings [12]. The additional contributions over those in the conference paper [12] are:

1. It has been shown in the conference version that the proposed Huffman encoding is *at least as good* as the static Huffman encoding. Here we provide a stronger theoretical result, that the new algorithm actually performs *better* than the static Huffman coding. In other words, the best known bound for the size of a compressed file by any dynamic variant of Huffman coding was n bits above the compression of static Huffman. While the conference version showed that this addition may be as low as zero, we reduce this upper bound even further, to $-(m-1)$.
2. In the newly added Section 6, we introduce two block-wise variants of the proposed algorithm, a first that suggests a practical version which provides additional savings on our test files, and an alternative that extends the idea of selectively updating the statistics, similarly to what has been suggested in [11].
3. The empirical results presented here are more involved.

3 Forward looking Huffman Coding

The traditional dynamic Huffman coding, and in fact, practically all adaptive compression algorithms, update the model according to what has already been seen in the file processed so far. The underlying assumption is that the past is a good approximation of the future, according to the biblical postulate that

The thing that hath been, it is that which shall be

Ecclesiastes 1:9, King James Version

More precisely, the distribution of the elements in the prefix of the file up to the current point, serves as an estimate for the distribution of these elements from the current point onwards. Such an assumption seems to be justifiable, especially for homogeneous texts written in some natural language, but there is of course no guarantee that it holds for all possible input files.

The algorithm we suggest corresponds to a scenario in which the exact statistics of the element's occurrences are known, yet we prefer using a model that adapts itself while processing the file. However, contrarily to the classic dynamic methods, which base their current model on what has already been seen in the *past*, our algorithm uses the model's knowledge of what is still to come, i.e., it looks into the *future.* The idea of looking forwards rather than backwards in adaptive compression has already been proposed in a completely different context for files compressed by LZSS [28]: instead of pointing backwards to reoccurring strings, the locations of the pointers were moved and their direction was altered to point forwards, in order to enable streaming compressed pattern matching [13].

The motivation of the suggested amendment is a different approach to what should be done in order to produce a more economical encoding. Traditional dynamic Huffman coding concentrates only on the character that is currently processed: its frequency is incremented, which tends to shorten its codeword length for future usage, but it ignores the fact that these savings may come at the price of having certain other codewords lengthened. This "selfish" behavior is counterbalanced by the more altruistic approach of the forward looking variant, where the frequency of the currently processed element is *decreased*, even though, as a consequence, the corresponding codeword can only become longer, yet this action may shorten the codewords of other symbols that are still present in the tree, yielding an overall gain.

To allow a fair comparison, one has to take into account that the models on which the methods rely require different amounts of storage for their encodings. For instance, static Huffman coding does not need the exact frequencies of the m characters; if a canonical [25] Huffman tree is used, it suffices to transmit its *quantized source* $\langle n_1, n_2, \ldots, n_k\rangle$ as defined in [7], where n_i is the number of codewords of length i, for $1 \leq i \leq k$, and k is the longest codeword length. For example, the quantized source of the canonical tree equivalent to that in Fig. 1a would be $\langle 0, 1, 2, 6, 4\rangle$. By using a canonical tree, one can thus save the transmission of the frequencies, but the sequence of characters must then be sorted by frequency order. If, on the other hand, a non-canonical tree is acceptable, the order of the characters may be implicit, e.g., lexicographic, but then we need to know the length of each of the m codewords.

The forward looking Huffman algorithm requires the exact frequencies of the elements, and not just the corresponding codeword lengths. The standard dynamic Huffman method does not need any frequencies, since they are incrementally learned by both encoder and decoder.

We have therefore included the size of a header describing the model in our experiments below. When the size n of the text to be compressed is large relative to the

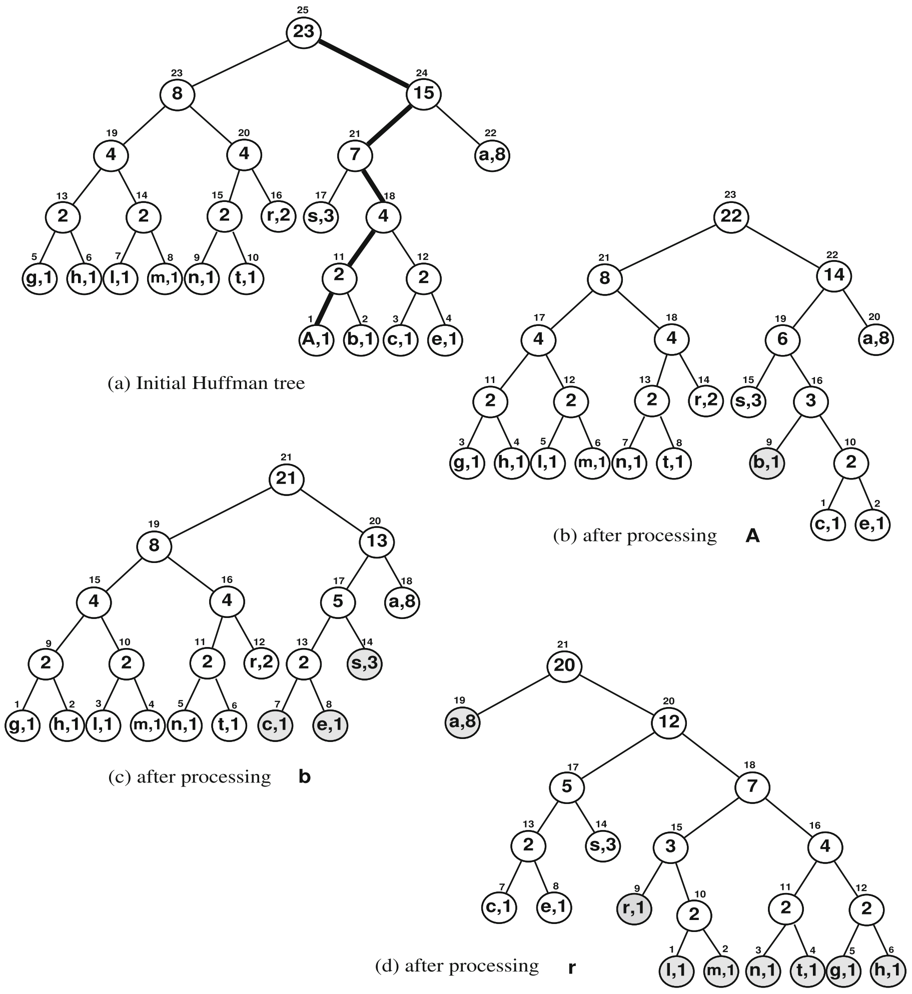

Fig. 1 Illustration of FORWARD-HUFFMAN for T =`Abrahamasantaclaragasse`

size *m* of the alphabet, the amount of storage required to encode the model is often negligible. For larger alphabets, for example when words, instead of just characters, are the elements to be encoded [21], the additional overhead of the forward looking variant may not be justifiable, unless the text to be encoded is very large. For certain applications, like large full-text Information Retrieval Systems, this overhead can be ignored, since the list of different words and their frequencies are usually stored anyway as part of the standard auxiliary data called *inverted files* [34].

4 Implementation Details and Extended Example

For the ease of description, we shall assume that the alphabet of the entire file is known in advance to both encoder and decoder. In practice, adaptive Huffman coding may reserve leaves in the Huffman tree only to characters that have already been processed, plus one leaf, often called NYT for Not Yet Transmitted, to enable the extension of the alphabet whenever a character c that has not been seen previously is encountered. In this case, the codeword of NYT is sent to the output, followed by some standard encoding of c, e.g., in ASCII (though this limits the size of the alphabet to 256, which is often not enough, for example when we wish to encode words rather than single letters). The leaf for NYT is then split, i.e., transformed into a node with two leaf children, one for c and one for NYT.

Our assumption is thus that we initialize a Huffman tree with m leaves. For the classical dynamic Huffman coding, all nodes are assigned a frequency of 0 (or all 1 if zero frequencies may cause trouble), and the frequencies are incremented at each step. The Huffman tree at the end is one that would have been produced by static Huffman coding. For the forward looking approach, the initial frequencies are those corresponding to the entire file, and they are decremented after the processing of each character. If one of the frequencies reaches 0, we know that no further occurrences of the corresponding character will appear in the file, so that its leaf may be removed from the tree.

The main tool for updating the Huffman tree, rather than reconstructing it from scratch after each character, is the *sibling property*, which guarantees that the tree is a Huffman tree, and is defined as follows:

Sibling Property A weighted binary tree with m leaves is said to have the sibling property if and only if

1. the m leaves have nonnegative weights $w_1, \ldots, w_m$, and the weight of each internal node is the sum of the weights of its two children; and
2. the nodes can be numbered in nondecreasing order by weight, so that the nodes indexed $2j-1$ and $2j$ in this numbering are siblings, for $1 \le j \le m-1$.

It has been shown in [8] that a tree is a Huffman tree if and only if it has the sibling property. As example, consider the tree in Fig. 1a, in which the internal nodes contain their weights, the leaf for character x contains the pair $(x, f(x))$, where $f(x)$ is its frequency, and the indexes in the above mentioned numbering are written above the nodes. We use a bottom-up, left to right numbering, but any other one is plausible, as long as it complies with the sibling property.

Given a Huffman tree, the following update procedure is used by the standard dynamic Huffman coding. If the currently processed character is c, the weights of all the nodes on the path from the leaf corresponding to c up to the root have to be incremented by 1. For example, referring again to Fig. 1a, suppose the next character is $c = \mathsf{r}$ whose leaf is the node indexed 16, then the weights of the nodes indexed 16, 20, 23 and 25 are incremented to 3, 5, 9 and 24, respectively. Note that for this example, the sibling property still holds after the updates (with the same numbering), so no further action is needed. However, in other cases, the increments may disrupt the monotonicity of the numbering.

Consider the case in which the next character is c = `A`, rather than `r`. The path from its leaf to the root is emphasized in Fig. 1a, and the weights of the nodes indexed 1, 11, 18, 21, 24 and 25 are incremented to 2, 3, 5, 8, 16 and 24, respectively. If we were to keep the same numbering, the weights of nodes 10–13 would be 1, 3, 2, 2, which is not a monotone sequence. In fact, with the present layout of the tree, no numbering can fulfill the second condition of the sibling property.

The difference between the two examples lies in the fact that in the first one, all the nodes that get updates have indexes which are maximal for their given weights before the increments, a property which does not hold for the second example. Indeed, there are many nodes with weight 2, all those with indexes 11 to 16, and the leaf (`r`,2) is indexed 16; nodes 18 to 20 have weight 4, nodes 22 and 23 have weight 8, and only node 25 has weight 23. On the other hand, for the second example, the leaf (`A`,1) is indexed 1, but there are also other nodes with weight 1 and that have higher indexes (up to 10); node 18 has weight 4, and node 20, which has a higher index, has also weight 4.

Clearly, this rule holds in general and not only for the examples, since a non-decreasing sequence of integers will remain such, even if the highest ranking elements within the sub-sequences of identical integers are incremented by 1. For example, $\cdots 6\ 6\ 7\ 7\ 7\ 7\ 9\ 9 \cdots$ may turn into $\cdots 6\ \mathbf{7}\ 7\ 7\ 7\ \mathbf{8}\ 9\ 9 \cdots$, which is still non-decreasing. To ensure that only such highest ranking nodes are updated, the dynamic Huffman algorithm exploits another property of Huffman trees, namely that nodes with identical weights may be interchanged. More precisely, since swapping the nodes is actually implemented by swapping the pointers to them, not just the nodes are interchanged, but the entire sub-trees rooted by these nodes. As a result, the shape of the tree might change, which yields a different set of codewords, but the weighted total path-length $\sum_{i=i}^{m} w_i \ell_i$ remains the same, so that the transformed tree is also a legitimate Huffman tree minimizing this sum, which represents the size of the compressed file.

Our algorithm adapts the dynamic Huffman procedure to the forward looking variant. After each encoded character, the number of its occurrences is *decremented* by 1. Before doing so, each node on the path from the updated leaf to the root is swapped, if necessary, with the *smallest* numbered node of identical weight. Then the weights of these nodes can be decremented without violating the sibling property. To continue the previous example, $\cdots 6\ 6\ 7\ 7\ 7\ 7\ 9\ 9 \cdots$ may now turn into $\cdots 6\ 6\ \mathbf{6}\ 7\ 7\ 7\ \mathbf{8}\ 9 \cdots$.

Given a text $T = x_1 \cdots x_n$ to be compressed, Forward looking Huffman encoding, or FORWARD-HUFFMAN, for short, compresses T using initially the Huffman tree of the static encoding. After every read character x_k, the corresponding codeword is output to the encoded file, and the Huffman tree gets modified to correspond to the frequencies within $\{x_{k+1} \cdots x_n\}$. The formal algorithm is given in Algorithm 1.

Algorithm 1 *Forward looking Huffman encoding.*

```
Forward-Huffman($T$)
1  Initialize a Huffman tree according to the frequencies within $T$
2  for $k \leftarrow 1$ to $n-1$ do
3      output the codeword for leaf($x_k$)
4      $p \leftarrow$ leaf($x_k$)
5      while $p \neq root$ do
6          $q \leftarrow$ lowest numbered node with same weight as $p$
7          if $q \neq p$ then
8              interchange $p$ with $q$
9          decrement $p$'s weight by 1
10         $p \leftarrow$ parent of $p$
11     $p \leftarrow$ leaf($x_k$)
12     if $p$'s weight is 0 then
13         replace $p$'s parent node by $p$'s sibling
14         delete the leaf $p$
```

Lines 11–14 deal with the case where the weight of leaf(x_k) has been reduced to 0, which means that the last occurrence of the character x_k is encountered. Note that in this case, the leaf is the lowest numbered one and must be the left child of its parent node q. It can thus be eliminated from the tree, by replacing q by its right child, which is the leaf's left sibling. Whenever the structure of the tree is altered (in lines 8 or 13), we assume that the numbering of the nodes, referred to in the sibling property, is updated as well.

For example, consider the text T = `Abrahamasantaclaragasse` (the name of a street in Vienna) over the alphabet { `A`, `a`, `b`, `c`, `e`, `g`, `h`, `l`, `m`, `n`, `r`, `s`, `t`} with corresponding weights {1, 8, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 1}. We started with one of the possible Huffman trees, shown in Fig. 1a, in which the leaves are assigned in lexicographical order. The other trees in the figure are those obtained after the processing, by Algorithm 1, of the first three characters of T, `A`, `b` and `r`, respectively.

When `A` is processed, all the nodes on the emphasized path from the leaf (`A`,1) to the root are the lowest indexed of their corresponding weights, so no interchanges in lines 6–8 are needed; the weight of each node on this path is then decremented, the parent of (`A`,1) is overwritten by the sibling of (`A`,1), which is the leaf (`b`,1), and the leaf (`A`,1) is erased from the tree. The resulting tree is shown in Fig. 1b in which the leaf that corresponds to the only codeword that has been changed, appears in gray. The shape of the tree has changed, therefore, a new numbering is necessary.

When the following input character `b` is processed, the corresponding leaf, numbered 9, is not the node with the lowest index among those with weight 1. Nodes 9 and 1 are therefore interchanged. Climbing up the tree from the new position of the leaf (`b`,1), we get to node 10, which is the lowest numbered node of weight 2. However, its parent node 16 has weight 3, as has also node 15. We therefore interchange the leaf (`s`,3) with the subtree rooted at node 16. The process continues to nodes

19, 22 and 23, whose corresponding weights are the smallest indexed ones for the weights 6, 14 and 22, respectively, so no further updates are needed. The resulting tree is given in Fig. 1c, which again displays the changed codewords in gray.

The processing of character `r` starts by swapping the nodes 12 and 9 in Fig. 1c, and then continues to node 15, which is the lowest indexed of weight 4. Its parent node 19, is interchanged with leaf 18; the updates continue with nodes 20 and 21. Finally, the weights of the nodes on the new path from the root to leaf (`r`,2) are decremented, resulting in the tree given in Fig. 1d.

The decoding algorithm is symmetrical. At the beginning the decoder is given the frequencies of the symbols in the entire file, similarly to the static approach, which is sufficient for correct decoding. The initial tree is constructed based on these frequencies, and the tree gets updated according to the last character that has been processed. That is, once a codeword is decoded, the frequencies of all the nodes on the path from the root to the corresponding leaf have to be decremented by 1. This is performed in a bottom-up scan according to Vitter's algorithm, exchanging subtrees of equal weights to maintain the sibling property, similarly to Algorithm 1. When the frequency of a leaf x reaches zero, the leaf x is eliminated from the Huffman tree by replacing its parent node by the sibling of x.

5 Analysis

Dynamic Huffman coding repeatedly changes the shape of the tree, but there is a delay between the occurrence of a change and when such a change starts to influence the encoding. For encoding the current character we use the tree built in the previous stage, and the changes implied by the processed character do only affect the encoding in the subsequent stages, if at all. This behavior is demonstrated in the following extreme example, comparing its performance to FORWARD. The example shows that the file constructed by traditional dynamic Huffman may be about twice as large as that produced by the FORWARD-HUFFMAN algorithm.

Let T $=$`CAAB`$\{$`BBAA`$\}^k$ for some positive integer k. We initialize our Huffman tree with $\Sigma = \{$`A`, `B`, `C`$\}$ as shown in Fig. 2a for the prefix `CAAB` of T. Consider first the standard dynamic algorithm. When the two `B`s of the first quadruple `BBAA` are processed, only the second `B` causes a change in the structure of the Huffman tree, but this happens *after* the two `B`s have already been encoded using 2 bits for each.

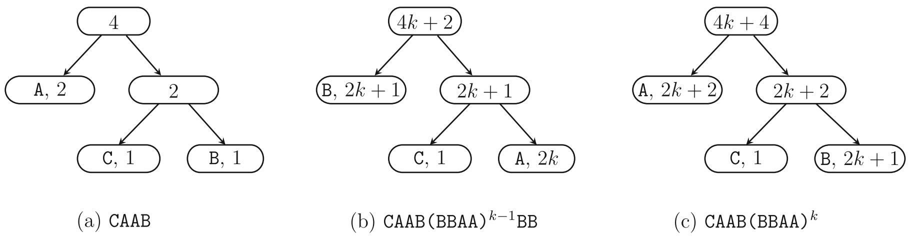

Fig. 2 Example for which classical dynamic Huffman coding produces a file twice the size of that constructed by FORWARD-HUFFMAN

The Huffman tree after reading the prefix CAABBB of T, is depicted in Fig. 2b. When the following two As of the first quadruple BBAA are processed, again the positions of the A and B nodes are swapped only *after* the frequency of A exceeds that of B, so each of the As is also encoded by 2 bits. The resulting Huffman tree after processing CAABBBAA is presented in Fig. 2c, and this is in fact the same Huffman tree as that in Fig. 2a. This alternation between two different structures of the Huffman tree proceeds for each of the BBAA quadruples, and every character of T (except the first two As) uses 2 bits, for a total of $8k + 6$.

When compressing T using the proposed forward looking coding, the Huffman tree may start with the static Huffman tree identical to that of Fig. 2a. Since the weight of the first read character C is 1, its node is deleted, and the Huffman tree is reduced to only two leaves, one for A and the other for B. All the codewords are then of length 1 and the size of the compressed file is exactly $4k + 5$, roughly half the size of the compressed file constructed by traditional dynamic Huffman methods.

If we were to use static Huffman coding, the tree would be the one of Fig. 2c and the size of the encoded file would be $6k + 6$.

Note that this example also shows that the standard dynamic Huffman coding may produce an encoding which is worse than that of static Huffman. The new forward looking algorithm, on the other hand, is at least as good as static Huffman, not only on this example, but in general, as proved in the following Theorem.

Theorem 1 *For a given input file, the size of the encoded file according to* FORWARD-HUFFMAN *is not larger than the size of the encoded file according to static Huffman coding.*

Proof Suppose the input file has n characters and let us inspect the situation after $t < n$ characters have already been encoded. If we knew also the distribution of the characters in the first t characters, we should have built a Huffman code just for these t characters, but lacking this knowledge, we took the global frequencies and encoded accordingly. However, for the following $n - t$ characters, we know the true distribution, which might be different from the global one; thus continuing with the static global distribution, one can only be worse (or at least not better) than changing to another Huffman code according to the frequencies in the last $n - t$ characters, which is exactly what is done in FORWARD-HUFFMAN.

The overall sum of the codeword lengths on the entire file for the static encoding is thus larger or equal to the sum of the codeword lengths of the first t characters, plus the sum of the codeword lengths of the static Huffman encoding based only on the frequencies of the last $n - t$ characters.

However, the same argument applies on this static encoding of the last $n - t$ characters. Thus choosing $t = 1$ and repeating the argument on the suffix gives that the total file size achieved by a static Huffman code is larger than or equal to the total file size achieved by the FORWARD-HUFFMAN presented in Algorithm 1. □

One may actually show a stronger result: not only is the forward looking variant not worse than the static one, we show that it strictly improves. While the size of the compressed file produced by Vitter's dynamic Huffman coding [31] is upper bounded by $|\mathcal{E}_S(T)| + n$, where $\mathcal{E}_S(T)$ denotes the static Huffman encoding of the file T, the bound for our forward variant is $|\mathcal{E}_S(T)| - m + 1$, that is, provably smaller.

Theorem 2 *For a given input file over an alphabet Σ of size m, the size of the encoded file according to* FORWARD-HUFFMAN *is smaller by at least $m-1$ bits than the size of the encoded file according to static Huffman coding.*

Proof The proof of the claim relies on the fact that in the FORWARD-HUFFMAN algorithm, the leaves of the Huffman tree are gradually removed after the last occurrence of the corresponding characters, while for the static variant, the number of leaves remains unchanged. We could have defined the FORWARD-HUFFMAN algorithm without this improvement, leaving the leaves with frequency 0 in the tree.

We construct a sequence of static trees $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$ according to the progress of FORWARD-HUFFMAN while processing the file T. We denote the encoding algorithm using these m static trees by FORWARD1. $\mathcal{S}_1$ is identical to the only Huffman tree of T used by the static Huffman encoding, and $\mathcal{S}_{i+1}$ is constructed from $\mathcal{S}_i$ when the ith character in Σ is removed by FORWARD-HUFFMAN after the last occurrence of $\sigma_i \in \Sigma$ has been encountered in T. The last tree, $\mathcal{S}_m$ consists of a single node and has depth 0.

Refer to the node α that has σ_i as one of its children, and let β be the sibling node of σ_i. $\mathcal{S}_{i+1}$ is obtained from $\mathcal{S}_i$ by replacing the subtree rooted by α by the subtree rooted by β.

There is no codeword in $\mathcal{S}_{i+1}$ that gets lengthened by the transformation from $\mathcal{S}_i$ to $\mathcal{S}_{i+1}$. Moreover, shortening the codewords in the subtree rooted by β implies the savings of a single bit for each occurrence in the remaining portion of T of characters corresponding to one of the leaves of β. Therefore, the encoding by FORWARD1 saves at least $m-1$ bits as compared to $\mathcal{S}_1$ that is identical to the tree used by the static Huffman coding.

On the other hand, at each point in T where $\mathcal{S}_{i+1}$ is constructed from $\mathcal{S}_i$, both the tree of FORWARD and $\mathcal{S}_{i+1}$ hold the same set of leaves. From the optimality of Huffman for the distribution of the characters in the remaining portion of T, it follows that the encoding according to $\mathcal{S}_i$ cannot be better than the encoding of the same portion of T by FORWARD. This implies that FORWARD is at least as good as FORWARD1, which is better than static Huffman by at least $m-1$ bits. □

Note that the bound given in Theorem 2 is tight, even between FORWARD-HUFFMAN and static Huffman coding as can be seen by the following example. Consider any text T at the end of which a single copy of each of the characters in Σ has been concatenated. In that case, the original T is processed identically by the two variants static Huffman coding and FORWARD1. For the remaining m characters of Σ, a single bit is saved by FORWARD1 (and therefore by FORWARD), relative to the static Huffman encoding for each except the first.

Remark that the claim in Theorem 1 refers to the total lengths of the encoded files, and it is not necessarily true that *each* codeword of FORWARD-HUFFMAN, at every position, is of equal length or shorter than the corresponding codeword of the static

Huffman code. Consider as example T = AAABBBCA, for which this claim does not hold. The static Huffman encoding refers to the frequencies 4, 3 and 1 for characters A, B and C, with codeword lengths 1, 2 and 2, respectively, using, e.g., the Huffman tree shown in Fig. 3a.

Figure 3 shows the way the Huffman tree is transformed for FORWARD-HUFFMAN coding while T is processed. FORWARD-HUFFMAN starts with the same tree as static Huffman coding. When the first A is processed, it is encoded by 0, and then its frequency is decreased, as shown in Fig. 3b. The second A is still encoded by 0, but then, once its frequency is updated to 2, in Fig. 3c, its node gets swapped with the node for B, so that the codeword for B becomes shorter, but at the price of the codeword for A becoming longer. The third A is then encoded by 10, a codeword of two bits, while the static version of Huffman coding is still using a single bit to represent A, showing that a codeword of FORWARD, at some position, might be longer than the corresponding codeword assigned by static Huffman. The following three Bs are encoded by a single 0 bit each, followed by decrements of its frequency, to 2, 1, and 0, as shown in Fig. 3d, e and f, when the node corresponding to B is finally removed from the tree, resulting in the tree of Fig. 3g. The following character C generates the

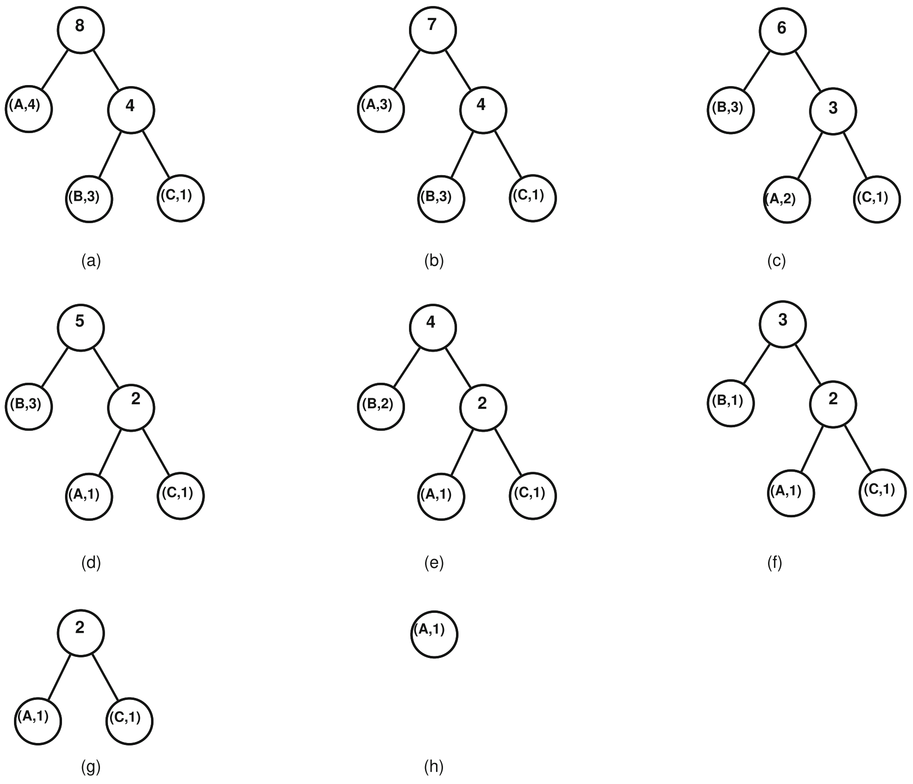

Fig. 3 Dynamically changing Huffman tree in FORWARD-HUFFMAN for T = aaabbbca

codeword 1, followed by a decrement of its frequency, and a removal of C from the tree, yielding the tree of Fig. 3h with a single leaf corresponding to A.

At this point no further bits need to be transferred to the decoder, who already realizes that the remaining suffix of the file may only contain a single character, which must be A, and which repeats the number of times indicated by its frequency (1 in our example). The encoding using FORWARD-HUFFMAN is therefore 0 0 10 0 0 0 1 (spaces are inserted for clarity), resulting in an average codeword length of $\frac{8}{8} = 1$, less than the average codeword length for static Huffman, which is $\frac{4\times1+3\times2+1\times2}{8} = 1.5$.

6 Block-wise FORWARD-HUFFMAN Coding Variants

As mentioned above, a header consisting of the details of the chosen model should be prepended to the compressed file, in case these values are not known to the decoder. The FORWARD-HUFFMAN algorithm requires the exact frequencies of the elements. If these frequencies are encoded according to a predefined order, e.g., that implied by ASCII, the frequencies (including zeroes if necessary) can be listed sequentially. One of the possibilities for the encoding of these frequencies could be Elias's C_δ code [5], a universal encoding method of the integers ≥ 1 using about $\log x + \log\log x$ bits to encode the value x. However, as it might be necessary to encode a frequency 0, we use $C_\delta(x+1)$ to encode the value x.

Both classical static Huffman coding and the FORWARD-HUFFMAN coding use the distribution of the elements to be encoded in the entire file. However, the occurrences of these elements are not necessarily spread uniformly, which might have a negative impact on the compression efficiency.

We propose a block-wise FORWARD-HUFFMAN encoding variant, that considers a limited portion of the file at a time. That is, this version is given a predefined parameter B which denotes the number of elements in a single block. Each block is then compressed independently, having a prelude of its own. The disadvantage of such a version is obviously that the frequency of a specific symbol should then be transferred at the beginning of all the blocks it appears in. On the other hand, the frequency values are smaller, and limited by the size B of the block, thus require fewer bits for their encoding. Moreover, characters that do not appear at all in a certain block need not be encoded in it, thereby possibly reducing the size of the tree. We call this version *Independent block-wise*.

A different block-wise variant, called *Incremental block-wise*, still processes the entire input file as the original FORWARD-HUFFMAN, but updates the frequencies and changes the Huffman tree only selectively at the beginning of each block rather than after each character. That is, the frequencies of the first block are initialized in the same way as in the original algorithm. During the encoding of a block, both encoder and decoder store the number of occurrences of each symbol, locally, without updating the tree, and only at the end of the block, the obtained frequencies are subtracted from the frequencies at the beginning of the block. The special case $B = 1$ is the proposed FORWARD-HUFFMAN algorithm. When B is the size of the input file, this corresponds to the classical static Huffman coding.

7 Empirical Results

To get empirical evidence how the three algorithms behave in practice, we considered texts of different languages and alphabet sizes: *ebib* is the Bible (King James version) in English, in which the text has been stripped of all punctuation signs except blank; *ftxt* is the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [30]; *English* is the concatenation of English text files selected from the etext02 to etext05 collections of the Gutenberg Project, from which the headers related to the project were deleted so as to leave just the real text; and *sources* is formed by C/Java source codes obtained by concatenating the first 50MB of the .c, .h and .java files of the linux-2.6.11.6 distributions. The dataset also includes the executable files `static.o`, `dynamic.o` and `forward.o`, of the static, dynamic and FORWARD-HUFFMAN source codes we used, respectively.

Our goal was to compare the compression performance of the three methods: static Huffman, the traditional dynamic Huffman and the proposed FORWARD-HUFFMAN algorithms. The results are presented in Table 1. The second column gives the original file sizes in MB. The third column gives the size of the encoded alphabet, m. The following three columns, headed STATIC, DYNAMIC and FORWARD, represent the sizes of the compressed output files. More precisely, for STATIC the numbers are given in bytes, while for DYNAMIC and FORWARD the presented values are the differences, in bytes, from the values in STATIC. As mentioned, we included the overhead of the description of the model in the size of the compressed file.[1]

We see that on our test files, the compression results are extremely close, and that our new FORWARD variant gives consistently smaller values than STATIC, as expected. The FORWARD variant is also generally, but not always, better than DYNAMIC, which itself often improves on STATIC. The executable files are examples in which STATIC is strictly better than DYNAMIC.

For our next set of experiments, the results of which are reported in Table 2, we wanted to consider also larger alphabets. An appropriate application area would then be large Information Retrieval Systems, the texts of which are often encoded as a sequence of *words*, rather than just characters, which yields much better compression. As mentioned earlier, a plausible scenario is to assume that the list of different words and their frequencies are needed anyway for the Information Retrieval process and are thus already stored, implying no additional overhead for the compression techniques. We considered datasets taken from the *Canterbury corpus*[2] for this set of experiments. Each file was preprocessed to obtain the underlying list of different words, defined as strings separated by white space. The columns of Table 2 follow the same structure as Table 1.

[1]The numbers differ slightly from those in [12], not only because of the different representation but also since dynamic Huffman coding has been implemented here exactly as suggested by Vitter [31], while a more optimized variant of Nelson [23], which repeatedly performs rescaling of the frequencies, has been used in [12].

[2]http://corpus.canterbury.ac.nz/

Table 1 Compression performance for small alphabets

File	full size	m	Compressed size		
	MB		Static	Dynamic	Forward
ebib	3.5	53	1,941,863	−71	−45
ftxt	7.3	127	4,398,943	−78	−120
English	50	217	29,917,189	−78	−187
sources	50	205	36,429,014	−52	−326
`static.o`	0.022	256	14,684	+31	−124
`dynamic.o`	0.023	254	14,793	+26	−124
`forward.o`	0.031	255	20,132	+14	−139

As can be seen, the compression differences for the larger alphabets are more significant than those in Table 1. Our new method is consistently better than static Huffman by up to a few percent, and the traditional dynamic Huffman is worse than STATIC on all our tests, even up to 9% for the file *ftxt*. We note that the extra space of DYNAMIC over STATIC is consistent with the upper bound of *at most n* additional bits given by Vitter [31], where n is the number of elements in the parsing of the text. For example, *ftxt* is parsed into a sequence of 1,176,190 words, implying an upper bound of that many bits, which are 147,023 bytes. On the other hand, the upper bound on the new FORWARD method implied by Theorem 2 depends on m rather than n; according to it, the savings for the same file *ftxt* are *at least* 75,191 bits, or 9399 bytes.

The graphs of Fig. 4 present our experimental results for the Independent blockwise FORWARD LOOKING variant, using the first four input files mentioned in Table 1. The compression ratio, defined as the size of the compressed divided by the size of the original file, is given as a function of the block size B with $1 \leq B \leq 1000$ KB. Note that the graphs for *English* and *ebib* literally overlap. As can be seen, the compression savings are the best for blocks of sizes up to about 50MB on our datasets, despite the fact that the prelude has to be added at the beginning of each block. However, the compression gain is moderate, and for larger blocks,

Table 2 Compression performance for large alphabets

File	full size	m	Compressed size		
	MB		Static	Dynamic	Forward
asyoulik	0.12	5,317	28,545	+1,254	−1,197
alice29	0.15	5,312	32,103	+1,396	−1,200
lcet10	0.41	9,946	80,303	+2,890	−2,342
bbe.txt	4.3	22,180	1,052,190	+43,071	−6,347
ebib	3.5	11,377	671,866	+22,331	−3,492
ftxt	7.3	75,192	1,568,984	+142,306	−17,983

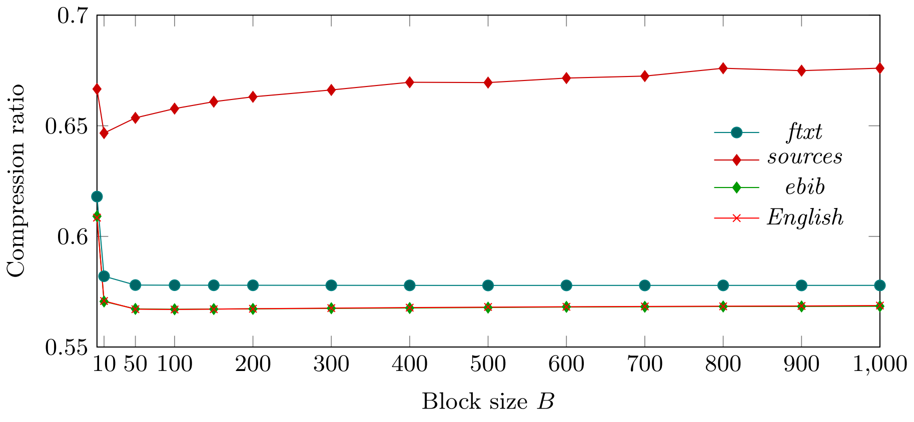

Fig. 4 Compression ratio as a function of the block size B using the Independent block-wise FORWARD LOOKING variant

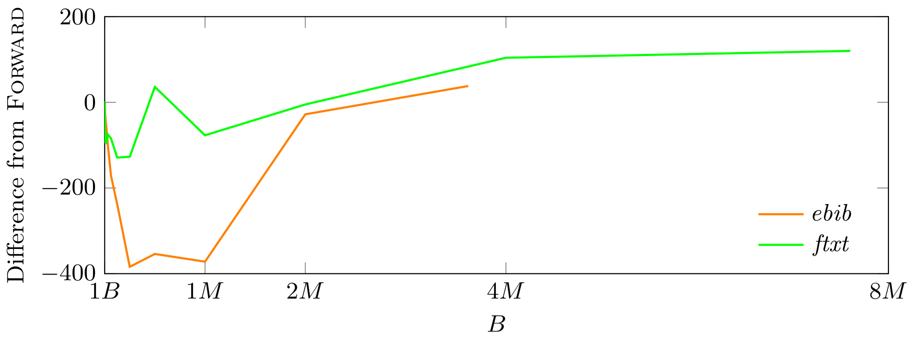

Fig. 5 The difference, in bytes, between the sizes of the compressed files using the Incremental block-wise FORWARD LOOKING variant and FORWARD, as a function of the block size B

Table 3 Encoding execution time

File	Static	Dynamic	Forward
ebib	0.067	0.147	0.209
ftxt	0.071	0.177	0.210
English	0.085	0.194	0.212
sources	0.091	0.212	0.208
`static.o`	0.272	0.574	0.672
`dynamic.o`	0.272	0.549	0.650
`forward.o`	0.227	0.684	0.495

Table 4 Decoding execution time

File	Static	Dynamic	Forward
ebib	0.073	0.167	0.163
ftxt	0.083	0.153	0.161
English	0.085	0.154	0.157
sources	0.095	0.172	0.190
`static.o`	0.321	0.573	0.769
`dynamic.o`	0.308	0.555	0.629
`forward.o`	0.281	0.556	0.556

the compression efficiency is similar to that of the original FORWARD-HUFFMAN algorithm.

Figure 5 presents the difference, measured in bytes, between the sizes of the file obtained from the *Incremental block-wise* FORWARD-HUFFMAN *coding* with blocks of size B and the original FORWARD HUFFMAN coding, which is the special case $B = 1$. The test files were *ebib* and *ftxt*. The x-axis corresponds to the size of the block B in the range from 1 byte (referring to FORWARD) and up to the size of the input file (referring to static Huffman). The y-axis shows the differences between the sizes of the compressed files using a block of B bytes and the compressed file obtained by FORWARD, that is, with $B = 1$. Positive numbers imply a loss in compression, while negative numbers refer to savings. Note that in all cases, the difference is no more than 384 bytes, which is negligible when compared to the compressed file sizes.

The space requirements by all algorithms is the memory used by the corresponding Huffman trees plus $O(1)$ for variables and pointers which is about the same. For our last set of experiments, we were interested in comparing the execution times of all mentioned algorithms. All experiments were conducted on a Virtual Machine with 1GB of memory and a single CPU, running Ubuntu 64 bits, on an Intel Core i5-4300U @ 1.80–2.50 GHz processor. Tables 3 and 4 report the average compression and decompression times, respectively, for 100 runs. The times are presented in seconds per Megabyte. As can be seen, static Huffman is obviously the fastest, and runs typically in about half the time of DYNAMIC. FORWARD is consistently a bit slower than DYNAMIC.

8 Conclusion

The contribution of this paper is twofold, theoretical as well as practical. The theoretical result is that the standard static Huffman coding, well-known for its optimality, may in fact be improved. In practice, this is sometimes achieved by the standard dynamic Huffman coding, but this traditional version can also be worse. The new forward looking Huffman coding, on the other hand, is *provably* better than static Huffman *on all files*, though may at times be outperformed on certain files by the classical dynamic coding. For executables and for large alphabets, when the precise number of occurrences is already known to the decoder such as in Information Retrieval applications, the FORWARD algorithm was also better than DYNAMIC on our tests.

Acknowledgements We wish to thank the anonymous reviewers for their constructive comments.

References

1. Amir, A., Benson, G.: Efficient two-dimensional compressed matching. In: Proc. IEEE Data Compression Conference DCC–92, pp. 279–288 (1992)
2. Baruch, G., Klein, S.T., Shapira, D.: A space efficient direct access data structure. Journal of Discrete Algorithms **43**, 26–37 (2017)
3. Baruch, G., Klein, S.T., Shapira, D.: Applying compression to hierarchical clustering. In: Similarity Search and Applications - 11th International Conference, SISAP 2018, Lima, Peru, October 7-9, 2018, Proceedings, pp. 151–162 (2018)
4. Cleary, J., Witten, I.: Data compression using adaptive coding and partial string matching. IEEE Trans. Commun. **32**(4), 396–402 (1984)
5. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. Inf. Theory **21**(2), 194–203 (1975)
6. Faller, N.: An adaptive system for data compression. In: Record of the 7th Asilomar Conference on Circuits, Systems and Computers, pp. 593–597 (1973)
7. Ferguson, T.J., Rabinowitz, J.H.: Self-synchronizing Huffman codes. IEEE Trans. Inf. Theory **30**(4), 687–693 (1984)
8. Gallager, R.: Variations on a theme by Huffman. IEEE Trans. Inf. Theory **24**(6), 668–674 (1978)
9. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)
10. Jacobson, G.: Space efficient static trees and graphs. In: Proceedings of FOCS, pp. 549–554 (1989)
11. Klein, S.T., Opalinsky, E., Shapira, D.: Selective dynamic compression. In: Proceedings of the Prague Stringology Conference 2019, Czech Technical University in Prague, Czech Republic (2019)
12. Klein, S.T., Saadia, S., Shapira, D.: Forward looking Huffman coding. In: The 14th Computer Science Symposium in Russia, CSR, Novosibirsk, Russia, July 1-5, LNCS 11532, pp. 203–214 (2019)
13. Klein, S.T., Shapira, D.: A new compression method for compressed matching. In: Data Compression Conference, DCC Snowbird, Utah, USA, March 28-30, 2000, pp. 400–409 (2000)
14. Klein, S.T., Shapira, D.: Compressed pattern matching in JPEG images. Int. J. Found Comput. Sci. **17**(6), 1297–1306 (2006)
15. Klein, S.T., Shapira, D.: Compressed matching in dictionaries. Algorithms **4**(1), 61–74 (2011)
16. Klein, S.T., Shapira, D.: On improving Tunstall codes. Inf. Process. Manage. **47**(5), 777–785 (2011)
17. Klein, S.T., Shapira, D.: Compressed matching for feature vectors. Theor Comput. Sci. **638**, 52–62 (2016)
18. Klein, S.T., Shapira, D.: Random access to Fibonacci encoded files. Discret. Appl. Math. **212**, 115–128 (2016)
19. Klein, S.T., Shapira, D.: Integrated encryption in dynamic arithmetic compression. In: Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings, pp. 143–154 (2017)
20. Knuth, D.E.: Dynamic Huffman coding. Journal of Algorithms **6**(2), 163–180 (1985)
21. Moffat, A.: Word-based compression. Softw. Pract. Exper. **19**(2), 185–198 (1989)
22. Navarro, G.: Compact Data Structures: a Practical Approach. Cambridge University Press, Cambridge (2016)
23. Nelson, M., Gailly, J.-L.: The Data Compression Book. 2nd edn. M & T Books (1996)
24. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), 43 (2007)
25. Schwartz, E.S., Kallick, B.: Generating a canonical prefix encoding. Commun. ACM **7**, 166–169 (1964)

26. Shapira, D., Daptardar, A.H.: Adapting the knuth-morris-pratt algorithm for pattern matching in huffman encoded texts. Inf. Process. Manage. **42**(2), 429–439 (2006)
27. Singh, A., Gilhotra, R.: Data security using private key encryption system based on arithmetic coding. International Journal of Network Security & Its Applications (IJNSA) **3**(3), 58–67 (2011)
28. Storer, J.A., Szymanski, T.G.: Data compression via textural substitution. J. ACM **29**(4), 928–951 (1982)
29. Tunstall, B.P.: Synthesis of noiseless compression codes. Georgia Institute of Technology (1967)
30. Véronis, J., Langlais, P.: Evaluation of parallel text alignment systems: the arcade project. In: Véronis, J. (ed.) Parallel Text Processing (2000). Kluwer Academic Publishers, Dordrecht, Chapter 19, pp. 369–388
31. Vitter, J.S.: Design and analysis of dynamic Huffman codes. J. ACM **34**(4), 825–845 (1987)
32. Wen, J., Kim, H., Villasenor, J.: Binary arithmetic coding with key based interval splitting. IEEE Trans. on Signal Processing Letters **13**(2), 69–72 (2006)
33. Witten, I.H., Neal, R.adford.M., Cleary, J.G.: Arithmetic coding for data compression. Commun. ACM **30**(6), 520–540 (1987)
34. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2), 6 (2006)