Bidirectional Adaptive Compression

Aharon Fruchtman[1], Shmuel T. Klein[2], and Dana Shapira[1]

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
aralef@gmail.com, shapird@g.ariel.ac.il

[2] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

Abstract. A new dynamic Huffman encoding has been proposed in earlier work, which instead of basing itself on the information gathered from the already processed portion of the file, as traditional adaptive codings do, uses rather the information that is still to come. The current work extends this idea to *bidirectional* adaptive compression, taking both past and future into account, and not only performs at least as good as static Huffman, but also provably improves on the future-only based variant. We give both theoretical and empirical results that support the enhancement of the new compression algorithm.

1 Introduction

Data compression techniques are often classified according to various criteria. One of the popular partitions is into static and adaptive variants. Alternatively, they can be arranged by whether being statistical or dictionary based methods. We focus in this paper on adaptive statistical compression such as Huffman [7] and arithmetic coding.

Research in data compression has evolved in several directions over the years, e.g., *compressed pattern matching* in texts [1,19], in images [11,14] and in structured files [12,3], *compact data structures* [8,18,15,2] and *cryptosystems* [16]. The current paper is yet another research regarding the core of encoding such as [5,23,7,4,20,17,13].

The traditional approach to adaptive coding updates the model dynamically according to what has already been seen in the file processed so far. The distribution of the following item to be encoded at some current location in the file is determined according to the distribution of the elements that have occurred up to that point. A different adaptive approach is suggested in [9], assuming that the exact statistics of the number of occurrences of each element in the entire file are known. For example, these statistics could have been collected in a first, preprocessing, pass over the file. In this approach, the dynamic model adapts itself while processing the file by using its knowledge of what is still to come, i.e., it looks into the *future*, rather than what is done by the traditional dynamic methods, which base their current model on what has already been seen in the *past*.

This "looking into the future" paradigm has also been suggested in [10] for performing stream compressed matching in LZSS [20], where instead of letting the Ziv-Lempel type (offset, length) pairs point backward to reoccurring strings, the locations of these pointers were moved and their direction was reversed to point forward.

Classical dynamic compression algorithms focus only on the item that is currently processed and increments its frequency, which may consequently imply a shorter corresponding codeword in future usage. However, as a consequence these savings may come at the price of having certain other codewords lengthened. The forward approach improves upon this "egoistic" behavior by a more social approach of the

Aharon Fruchtman, Shmuel T. Klein, Dana Shapira: Bidirectional Adaptive Compression, pp. 92–101.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8

forward looking variant, where the frequency of the currently processed element is *decreased*, even at the price that the corresponding codeword can become longer. However, this operation may shorten the overall codeword length of those elements that are still present in the tree, yielding a better space savings.

2 A new hybrid adaptive coding method

The performance of both the classical static Huffman coding and its dynamic variants suffers from the fact that they use some information about the distribution of the characters to be encoded which is not necessarily needed. Moreover, this additional information has a negative impact on the compression ratio that may be achieved. More precisely, static Huffman coding uses throughout the frequencies of the characters in the entire text, yet the occurrences of these characters are not necessarily spread uniformly. In an extreme case in which a certain character x is clustered locally and does not appear elsewhere, the appearance of a leaf assigned to x in the Huffman tree is a burden, reducing the compression gain for the parts of the text outside of these clusters.

As another example, consider Huffman's dynamic variant, as proposed by Vitter [22], and an input text T consisting first of a long string of characters from $\{a, b, \ldots, z\}$, followed by a long string of numbers from $\{0, 1, \ldots, 9\}$. While for the beginning of the file, the Huffman tree will only have 27 leaves (including one for the not yet transmitted elements), the Huffman tree for the second part will have 37 leaves, including the newly encountered digits. However, the fact that non-numeric characters do not appear in the second part is not taken advantage of, which could have reduced the average codeword length.

The forward looking dynamic Huffman coding [9] starts with the full frequencies, as for the static algorithm, and then decreases them gradually after the occurrence of each of the characters. The extreme case of the input file T will then imply a symmetric behavior, in which the encoding of the first part of the file will be based on a full Huffman tree with 37 leaves and therefore be wasteful, and only in the second part will the tree be reduced to 11 nodes for the set of digits alone.

To remedy this potential source of inefficiency, we propose the following hybrid method taking advantage of both forward and backward looking dynamic Huffman coding and thereby correcting some of their drawbacks.

The idea is to start with the same tree as the traditional dynamic Huffman encoding. That is, at the beginning, the Huffman tree contains only a special leaf, symbolizing the set of characters that has not yet occurred in the text read so far. This special character is often labelled NYT for *Not Yet Transmitted*. The corresponding codeword is the empty string. The forward looking dynamic variant requires the entire distribution of the alphabet to be known at the beginning of the process, whereas the classical backward looking dynamic method of Vitter transmits the alphabet incrementally, each character immediately after its first occurrence. For Vitter's method, there is no need to transmit the character frequencies, which are updated on the encoder and decoder sides in synchronization.

For the hybrid method, we again assume that the whole distribution of the characters in the entire input file is known to the encoder, as for static encoding, but that this distribution will not be transferred to the decoder in advance as is done in the forward looking dynamic method. We now suggest transmitting a newly encountered character y immediately at its first occurrence, for example by issuing the

Huffman codeword for NYT, followed by some encoding of the new character y, e.g., in ASCII. The innovation is that at this point, we also transmit the frequency of y in the remaining part of the file.

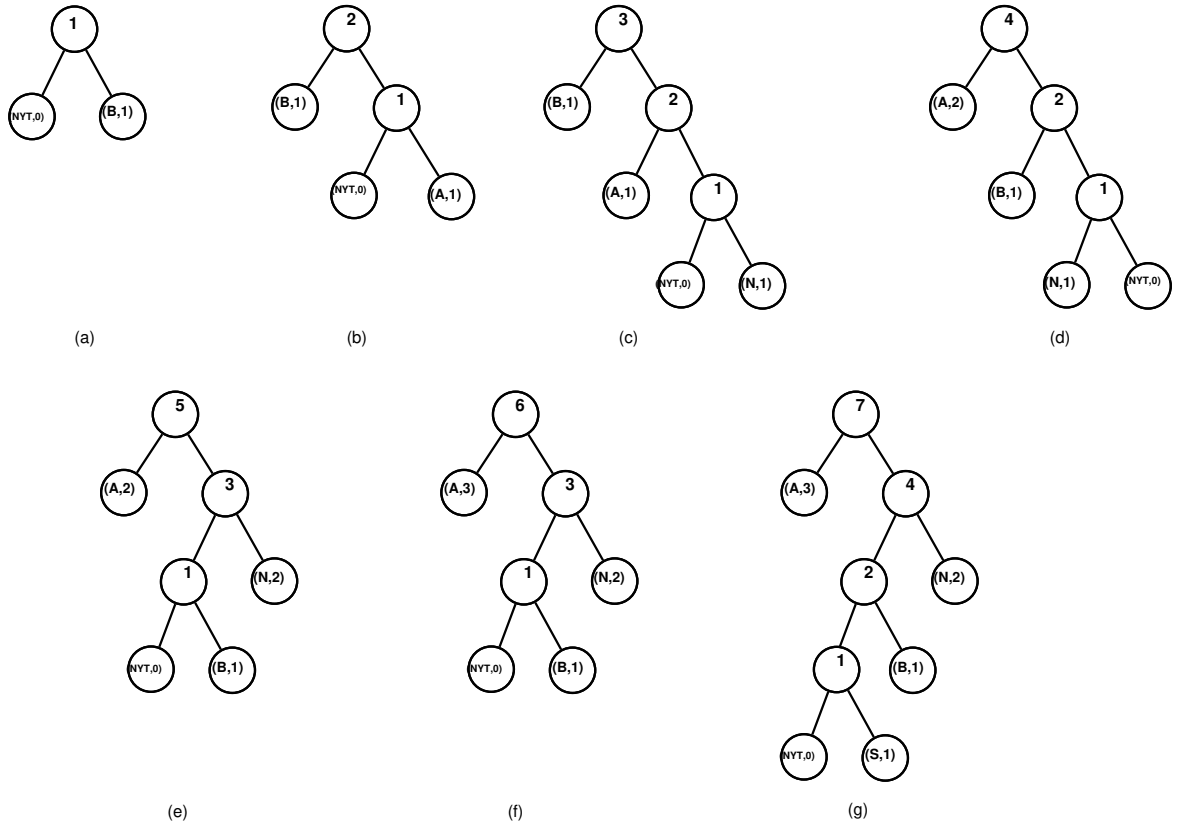

Figure 1. Illustration of the standard VITTER adaptive Huffman algorithm for $T =$ `BANANAS`.

The expected savings are of course not in the transmitted statistics of the characters, since instead of sending them as a bulk in a header at the beginning, they are spread within the encoded file. The real advantage lies in the fact that on the one hand, at each point, the current Huffman tree reflects only the set of characters in the prefix of the file so far, as in the backward looking variant, but that the frequencies are according to the suffix of the file, as in the forward looking variant, which is better from the compression point of view. Moreover, characters are altogether eliminated from the tree after their last occurrence, which in extreme cases, when the suffix only includes a small subset of the original alphabet, may yield significant savings.

3 Bidirectional Adaptive Coding

The bidirectional adaptive compression can be adapted to any statistical dynamic compression such as dynamic Huffman, adaptive arithmetic coding and Prediction by Partial Matching (PPM) [4]. The generic encoding algorithm, named HYBRIDENCODE, is given in Algorithm 1. Decoding is done symmetrically. In a preprocessing stage, the text T is scanned in order to gather the underlying statistics storing the frequency $\text{freq}(\sigma_i)$ for each character σ_i in the alpahbet Σ. The tree is initialized by NYT having as frequency the size of the alphabet, since this is the number of times it

will be used. The text $T = x_i \cdots x_n$ is rescanned. In case a character x_i is encountered for the first time, the codeword for NYT is transmitted followed by the ASCII encoding of x_i, and some encoding of the frequency $\mathsf{freq}(x_i)$. One of the possible choices for the encoding method 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. Otherwise, x_i is already in the model and encoded accordingly, its frequency is decremented by 1 and the model is updated.

Algorithm 1: HYBRID-ENCODE

```
HYBRID-ENCODE(T = x_1 ··· x_n )
preprocess T to get freq(σ_i),   ∀σ_i ∈ Σ
initialize the model with a single element, for NYT, with freq(NYT)) ← |Σ|
encode freq(NYT)
for i ← 1 to n do
    if x_i has already appeared earlier then
        encode x_i according to current model
        freq(x_i) ← freq(x_i) − 1
    else // first occurrence
        encode NYT according to the current model
        freq(NYT) ← freq(NYT) − 1
        output ASCII(x_i)
        encode freq(x_i)
        Update the model with x_i, freq(x_i) and freq(NYT)
```

Consider as example the text $T =$ BANANAS over the alphabet $\{\mathtt{A}, \mathtt{B}, \mathtt{N}, \mathtt{S}\}$ with corresponding weights $\{3, 1, 2, 1\}$. Figure 1 shows the way the Huffman tree alters for the standard adaptive Huffman coding of [22] while T is processed. The tree is initialized with the NYT node with 0 weight. When B is encountered, ASCII(B) is output and the letter B is inserted as a new leaf into the Huffman tree with weight 1 resulting in the tree presented in Figure 1(a). Note that for this standard algorithm, there is no need to transmit the frequency of B, which will be learned by the decoder incrementally while processing the remaining part of the file. When A is processed, the NYT with codeword 0 is output, followed by ASCII(A), and a new leaf for A is inserted resulting in Figure 1(b). The following character N is dealt with similarly, the codeword of NYT is now 10, followed by ASCII(N) (Figure 1(c)). To process the second A, and then the second N, the corresponding codewords 10 and 110 are output, their weights are incremented, and the tree is updated to Figure 1(d) and then to Figure 1(e). The codeword for the last A is 0, and while the structure of the tree remains the same, A's weight is incremented to 3 (Figure 1(f)). For the last character S, NYT is encoded by 100 and followed by ASCII(S), and the process stops (Figure 1(g)).

The FORWARD algorithm basically works in the opposite way, starting with the final tree of the dynamic variant, and ending with the empty tree. The main difference is that a special node for NYT and the transmission of the frequencies is not needed, as the entire alphabet and its statistics are assumed known to the decoder. Our running example for FORWARD-HUFFMAN is presented in Figure 2.

The initial tree, shown in Figure 2(a), contains statistics for the entire alphabet, exactly as for the static version of Huffman coding. When the only appearance of B is processed, it is encoded by 101, and then B is removed from the tree, resulting in

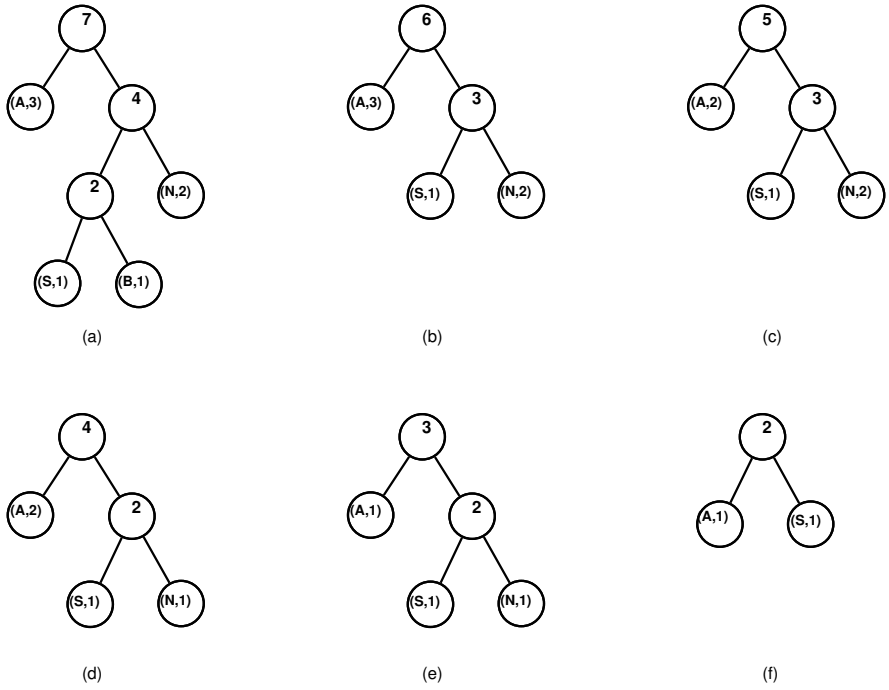

Figure 2. Illustration of FORWARD adaptive Huffman algorithm for $T =$ BANANAS.

the tree of Figure 2(b). The following characters A, N, and A generate 0, 11, and 0 as output, each followed by a decrement of their correnponding frequencies, with no other changes, yielding the trees Figure 2(c), Figure 2(d) and Figure 2(e), respectively. When the last N is processed, 11 is output again, and N is removed from the tree, resulting in the tree of Figure 2(f). When the last A is processed, 0 is output, and A is removed from the tree, resulting in a tree with a single node corresponding to S, thus 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 S, and which repeats freq(S) times.

The proposed hybrid algorithm starts with a tree containing the NYT node, but unlike for the standard dynamic Huffman algorithm, its frequency is initialized by $|\Sigma|$, which is 4 in our case. Each time a new character is encountered, the procedure uses its knowledge of the "future" and updates the exact frequency, which, subsequently, alters the model. When B is processed, it does not need to be inserted into the tree, as it only occurs once, and ASCII(B) is output, along with a single bit 1, which is the C_δ encoding of its frequency 1, resulting in the tree of Figure 3(b). When the following character A is encountered, having frequency 3, there is no need to encode NYT, which is still the only leaf in the tree, so only ASCII(A) is output, followed by $C_\delta(3) = 0101$, and A is inserted into the tree with frequency 2 (Figure 3(c)). To process N, NYT is now encoded by 0, followed by ASCII(N) (Figure 3(d)), followed by $C_\delta(2) = 0100$. A is then encoded by 0 (Figure 3(e)), N is encoded by 11 and removed from the tree (Figure 3(f)), and A is encoded as 0, and the tree is updated to contain again only NYT (Figure 3(g)). For S, which occurs only once in T, only its ASCII code is output, followed by its frequency $C_\delta(1) = 1$ (and not preceded by NYT).

Figure 4 summarizes this example in a comparative chart, showing the binary output sequences produced by the different approaches. For the standard VITTER

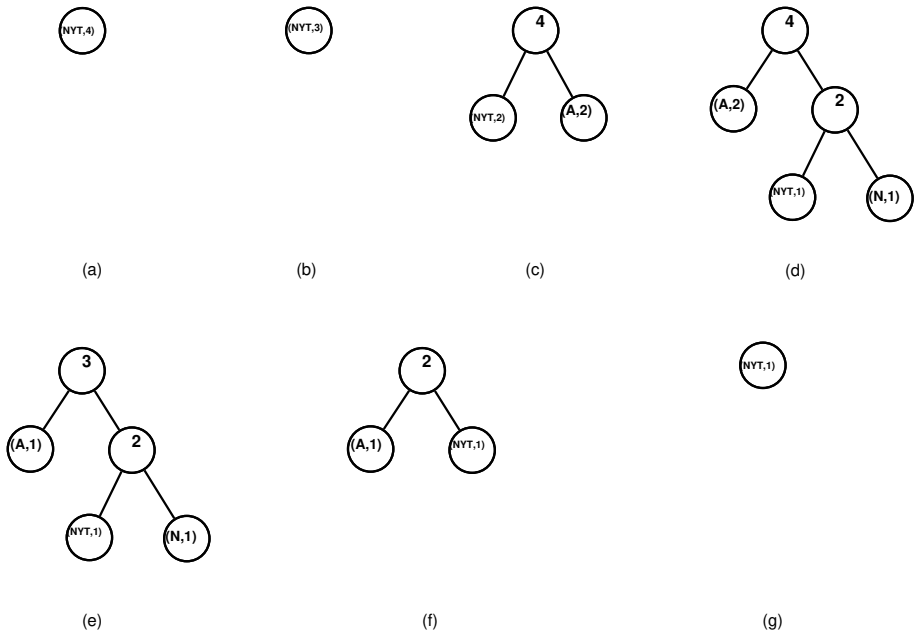

Figure 3. Illustration of HYBRID adaptive Huffman algorithm for $T =$ `BANANAS`.

algorithm, the output consists of Huffman codewords, and if the special codeword for NYT has been emitted, it is followed by the ASCII representation of the newly encountered character. For FORWARD, there are only Huffman codewords in the output, since the alphabet is known in advance, so there is no need for NYT. Note that there is no encoding for the last letter `S`, because in this particular example, `S` appears only once, and after all other characters have been eliminated from the tree, so the corresponding codeword is the empty string. In the HYBRID technique, the output is a sequence of elements of four different kinds: Huffman codewords of elements of the alphabet, the Huffman codeword for NYT, newly encountered characters in ASCII, and frequencies in C_δ, all of which can be uniquely identified according to their position in the compressed text.

VITTER	0100 0010	0	0100 0001		10	0100 1110	10	110	0	100	0101 0011	
	B	NYT	A		NYT	N	A	N	A	NYT	S	
FORWARD	101		0			11	0	11	0			
	B		A			N	A	N	A		S	
HYBRID	0100 0010	1	0100 0001	0101	0	0100 1110	0100	0	11	0	0101 0011	1
	B	$C_\delta(1)$	A	$C_\delta(3)$	NYT	N	$C_\delta(2)$	A	N	A	S	$C_\delta(1)$

Figure 4. Comparative chart for the output of the three adaptive encoding procedures.

4 Analysis of the proposed algorithm

The contribution of the forward looking dynamic Huffman coding algorithm proposed in [9] is that it provably always achieves compression savings that are at least as good as those of the static Huffman algorithm, which is often claimed to yield "optimal" performance, and practically also improves upon the dynamic compression efficiency. The contribution of the current paper is yet a new twist on the implementation, with both theoretical and practical improvements over the forward algorithm.

Theorem: The expected performance of the HYBRID algorithm is at least as good as that of the FORWARD algorithm.

Proof: Note that the FORWARD and HYBRID algorithms only differ in their behavior at the beginning of the processing of a file, up to the moment where the entire alphabet to be used is already known. From that point on, the two algorithms are identical and generate exactly the same codewords. Define as i_0 the index of the character in the input file from which on the two models coincide, that is, i_0 is the index of the first occurrence of the last character of the alphabet that is encountered in a left to right scan of the input. Denote by f_j and h_j the lengths of the codewords generated for the j-th input character by the FORWARD and HYBRID algorithms, respectively. We have that

$$h_j = f_j \qquad \text{for} \quad j \geq i_0. \tag{1}$$

Define the sequence $k_1 = 1, k_2, \ldots, k_{|\Sigma|} = i_0$, as the indices of the first occurrences of all the characters of the alphabet Σ, in the order of their appearances. At any point $k < i_0$, the set of characters seen so far, denoted by Σ_k, is a proper subset of Σ. When processing the input characters with indices between k_r and k_{r+1}, for $1 \leq r < |\Sigma|$, the HYBRID algorithm bases its encoding on an alphabet of only r characters, just as VITTER's variant would do. The FORWARD algorithm, on the other hand, uses already the entire alphabet, and therefore works with another set of *probabilities* for the given subset Σ_{k_r}, in spite of using the same *frequencies*.

Denote by H^{k_r} the Huffman code based on the frequencies of the characters in Σ_{k_r} and by $H^{k_r}_\sigma$ the codeword length for a character $\sigma \in \Sigma_{k_r}$ in this Huffman code. At point k_r (and up to the characters indexed $k_{r+1} - 1$), the HYBRID algorithm uses the Huffman code H^{k_r}, but the FORWARD algorithm uses other probabilities, and therefore the corresponding codeword lengths $F^{k_r}_\sigma$ cannot be better, because of the optimality of Huffman's procedure, that is

$$\sum_{\sigma \in \Sigma_{k_r}} p_\sigma H^{k_r}_\sigma \leq \sum_{\sigma \in \Sigma_{k_r}} p_\sigma F^{k_r}_\sigma, \tag{2}$$

where p_σ is the probability of occurrence of the character σ at the given index k_r.

In other words, looking at the character indexed k_r, we encode it for HYBRID using a Huffman code which has been built for $P = \{p_\sigma \mid \sigma \in \Sigma_{k_r}\}$, which is optimal at that specific point. But for FORWARD, we base ourselves on other probabilities, so the resulting Huffman code is *not* necessarily optimal for P. Therefore, when averaging with the same set of probabilities, we get (2), that is, the former set of lengths yields an average which is not larger than that obtained by the latter set.

Summing over all r for the indices up to i_0 and adding eq. (1) for the larger indices, we conclude that the *expected* length of a codeword for HYBRID is $\leq$ than for FORWARD. ∎

We remark that this does not mean that necessarily $h_j \leq f_j$ for all $j < i_0$, and that the claim is only for their expected values. Indeed we found at least one example for which $h_j = f_j + 1$ for some j.

Another remark concerns the expected savings by using HYBRID instead of FORWARD. These will be rather modest for many "typical" large files. The reason is that there is a possible advantage of the former only before the entire alphabet has been discovered in a scan from the beginning. But this will often happen fairly soon in typical natural language texts.

Indeed, the distribution of the characters in standard English texts is well known and can be found in many books, see, e.g., [6]. Though the details vary from one source to another, it is well known that the letters J and Q, for example, are rare and appear with probability about 0.002. But that means that the expected number of characters to scan until the first appearance of one of these characters is about $1/0.002 = 500$, if we assume for simplicity that the text is generated by independently appearing characters. Therefore it will only rarely occur that after a few thousand letters, there is still a part of the alphabet that has not been seen.

We therefore see the main contribution of this paper as a theoretical one, showing that one can improve, even if only slightly, on a method which already seems better than one considered generally as optimal. Useful applications might be restricted to special files with sharply varying statistics, e.g., a file consisting first only of digits, then of characters, and similar settings.

5 Experimental Results

In order to compare the compression savings of the herein suggested HYBRID method relative to the three other algorithms, the dynamic VITTER and FORWARD Huffman encodings, as well as the STATIC Huffman variant, we have considered several datasets of different sizes and nature, and using different alphabets. *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 [21]; *eng* is the concatenation of English text files, downloaded from the Pizza & Chili Corpus, selected from 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; *exe* is the executable file produced by compiling the source code we used for the hybrid Huffman algorithm; and *nt* is a dataset constructed from *ebib* in order to obtain a file composed of two parts having disjoint sets of alphabets — digits, followed by characters. The first part of the file takes the ten thousand first characters of the *ebib* file and writes them as decimal digits; the second part consists of the remaining characters of *ebib* in their original form. Table 1 summarizes the information regarding the used datasets.

In many encoding algorithms, the description of the underlying model on which the method relies on, is given in the header of the encoded file, and varies for each method. For static arithmetic coding the exact frequencies of the characters are not needed, and approximate probabilities suffice. On the other hand, the forward looking coding requires the exact frequencies of the items, while the traditional adaptive arithmetic method does not need any frequencies, since they are incrementally learned by both encoder and decoder. As the size of the prelude describing the model does not usually grow as a function of the size of the underlying file, it can be treated as a constant

File	full size (bytes)	$\lvert\Sigma\rvert$
ebib	3,711,020	53
exe	48,640	256
ftxt	7,648,930	132
eng	52,428,800	176
nt	3,726,683	63

Table 1. Information about the used datasets

sized header. Alternatively, we decided to compare the core of the encoding, neglecting the technical issue of transferring the model itself (which can be probably squeezed to be much shorter by a fine tuning of its encoding). The figures presented in Table 2 are the net sizes of the encodings, that is, the sizes of the entire codewords of the file without the description of the model, given in bytes. As can be seen from the results, the compression efficiencies of all methods are very close, which justifies the omission of the headers, as the description of the model is not negligible in the size of the difference.

File	Size of Encoded file (bytes)			
	Static	Adaptive	Forward	Hybrid
ebib	1,940,573	1,941,321	1,940,527	1,940,268
exe	31,296	31,851	31,132	28,930
ftxt	4,443,525	4,444,660	4,443,419	4,442,447
eng	29,914,197	29,915,562	29,914,021	29,912,644
nt	1,969,884	1,970,694	1,969,830	1,945,310

Table 2. Compression performance

Table 2 presents our experimental results on each file of our dataset in which the figures are given in bytes. The first column is the file's name. The second to fifth columns correspond to the size of the encoded files of static Huffman (Static), adaptive Huffman of Vitter (Adaptive), forward (Forward), and our proposed method (Hybrid), respectively. As theoretically expected, Hybrid consistently slightly improves Forward which itself improves both the static and dynamic versions, except for the file, *nt*, based on two different alphabets, for which the improvement is more significant.

References

1. A. Amir and G. Benson: *Efficient two-dimensional compressed matching*, in Proc. IEEE Data Compression Conference DCC–92, 1992, pp. 279–288.
2. G. Baruch, S. T. Klein, and D. Shapira: *A space efficient direct access data structure.* Journal of Discrete Algorithms, 43 2017, pp. 26–37.
3. G. Baruch, S. T. Klein, and D. Shapira: *Applying compression to hierarchical clustering*, in Similarity Search and Applications - 11th International Conference, SISAP 2018, Lima, Peru, October 7-9, 2018, Proceedings, 2018, pp. 151–162.
4. J. Cleary and I. Witten: *Data compression using adaptive coding and partial string matching.* IEEE Transactions on Communications, 32(4) 1984, pp. 396–402.
5. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.

6. H. Heaps: *Information Retrieval, Computational and Theoretical Aspects*, Academic Press, 1978.
7. D. A. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9) 1952, pp. 1098–1101.
8. G. Jacobson: *Space efficient static trees and graphs*, in Proceedings of FOCS, 1989, pp. 549–554.
9. S. T. Klein, S. Saadia, and D. Shapira: *Forward looking Huffman coding*, in The 14th Computer Science Symposium in Russia, CSR, Novosibirsk, Russia, July 1-5, 2019.
10. S. T. Klein and D. Shapira: *A new compression method for compressed matching*, in Data Compression Conference, DCC 2000, Snowbird, Utah, USA, 2000, pp. 400–409.
11. S. T. Klein and D. Shapira: *Compressed pattern matching in* jpeg *images.* Int. J. Found. Comput. Sci., 17(6) 2006, pp. 1297–1306.
12. S. T. Klein and D. Shapira: *Compressed matching in dictionaries.* Algorithms, 4(1) 2011, pp. 61–74.
13. S. T. Klein and D. Shapira: *On improving Tunstall codes.* Inf. Process. Manage., 47(5) 2011, pp. 777–785.
14. S. T. Klein and D. Shapira: *Compressed matching for feature vectors.* Theor. Comput. Sci., 638 2016, pp. 52–62.
15. S. T. Klein and D. Shapira: *Random access to Fibonacci encoded files.* Discrete Applied Mathematics, 212 2016, pp. 115–128.
16. S. T. Klein and D. Shapira: *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, 2017, pp. 143–154.
17. S. T. Klein and D. Shapira: *Context sensitive rewriting codes for flash memory.* Comput. J., 62(1) 2019, pp. 20–29.
18. G. Navarro: *Compact Data Structures: A Practical Approach*, Cambridge University Press, Cambridge, UK, 2016.
19. D. Shapira and A. H. Daptardar: *Adapting the knuth-morris-pratt algorithm for pattern matching in huffman encoded texts.* Inf. Process. Manage., 42(2) 2006, pp. 429–439.
20. J. A. Storer and T. G. Szymanski: *Data compression via textural substitution.* J. ACM, 29(4) 1982, pp. 928–951.
21. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The* arcade *project*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, Chapter 19, 2000, pp. 369–388.
22. J. S. Vitter: *Design and analysis of dynamic Huffman codes.* J. ACM, 34(4) 1987, pp. 825–845.
23. J. S. Vitter: *Algorithm 673: Dynamic Huffman coding.* ACM Trans. Math. Softw., 15(2) 1989, pp. 158–167.