Huffman Coding with Non-Sorted Frequencies

Shmuel T. Klein and Dana Shapira

[1] Department of Computer Science
Bar Ilan University, Ramat Gan, Israel
tomi@cs.biu.ac.il

[2] Department of Computer Science
Ashkelon Academic College, Ashkelon, Israel
shapird@ash-college.ac.il

Abstract. A standard way of implementing Huffman's optimal code construction algorithm is by using a sorted sequence of frequencies. Several aspects of the algorithm are investigated as to the consequences of relaxing the requirement of keeping the frequencies in order. Using only partial order may speed up the code construction, which is important in some applications, at the cost of increasing the size of the encoded file.

1 Introduction

Huffman's algorithm [6] is one of the major milestones of data compression, and even though more than half a century has passed since its invention, the algorithm or its variants find their way into many compression applications to this very day. The algorithm repeatedly combines the two smallest frequencies, and thus stores the set of frequencies either in a heap or in sorted form, yielding an $\Omega(n \log n)$ algorithm for the construction of the Huffman code, where n is the size of the alphabet to be encoded.

Working with a sorted set of frequencies is indeed a sufficient condition to get an optimal code, but the condition is not necessary. In certain cases, one can get optimal results even if the frequencies are not fully sorted, in other cases the code might not be optimal, but very closely so. On the other hand, relaxing the requirement of keeping the frequencies in order may yield time savings, as the generation of the code, if the frequencies are already given in order, or if their order can be ignored, takes only $O(n)$ steps.

One might object that since the alphabet size n can often be considered as constant relative to the size of the text to be encoded, there is no much sense in trying to improve the code construction process, and any gained savings will only marginally affect the overall compression time. But there are other scenarios for which the above mentioned effort may be justifiable: the ratio between the sizes of the text and the code is not always very large; instead of using a single Huffman code, better results are obtained when several such codes are used. For example, when the text is considered as being generated by a first order Markov process, one might use a different code for the successors of the different characters. When dynamic coding is used, the code is rebuilt periodically, sometimes even after each character read.

The loss incurred by not using an optimal (Huffman) code is often tolerable, and other non-optimal variants with desirable features, such as faster processing and simplicity have been suggested, for example Tagged Huffman codes [4], End-Tagged Dense codes [2] and (s, c)-Dense codes [1]. Similarly, the loss of optimality caused by moving to not fully sorted frequencies can also be acceptable in certain applications,

Shmuel T. Klein, Dana Shapira: Huffman Coding with Non-Sorted Frequencies, pp. 46–53.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

for example when based on estimations rather than on actual counts. In a dynamic encoding of a sequence of text blocks $B_1, B_2, \ldots$, block B_t is often encoded on the basis of the character frequencies in $B_1, \ldots, B_{t-1}$. The encoder could use the frequencies from block B_t itself, but deliberately ignores them because they are yet unknown to the decoder. By using the frequencies gathered up to block B_{t-1} only, decoding is possible without transmitting the code itself. The accuracy, however, of these estimates is based on the assumption that block t is similar to the preceding ones as to the distribution of its characters. If this assumption does not hold, the code may be non-optimal anyway, so an additional effort of producing an optimal code for a set of underlying frequencies that are not reliable, may be an overkill.

In the next section, we investigate some properties of the Huffman process on non-sorted frequencies. Section 3 then deals with a particular application, designing an algorithm for the dynamic compression of a sequence of data packets, and report on some experiments. In Section 4 we investigate whether a similar approach may have applications to other compression schemes than Huffman's.

2 Using non-sorted frequencies

The following example shows that working with sorted frequencies is not a necessary condition for obtaining optimality. Consider the sequence of weights $\{7, 5, 3, 3, 2, 2\}$, yielding the Huffman tree in Figure 1a. If we start with a slightly perturbed sequence $\{7, 5, 3, 2, 3, 2\}$ and continue according to Huffman's algorithm, we get the tree in Figure 1b, which is still optimal since its leaves are on the same levels as before, but it is not a Huffman tree, in which we would not combine 2 with 3. The tree of Figure 1c corresponds to starting with the sorted sequence, but not keeping the order afterwards, working with the sequence $\{7, 5, 6, 4\}$ instead of $\{7, 6, 5, 4\}$ after two merges.

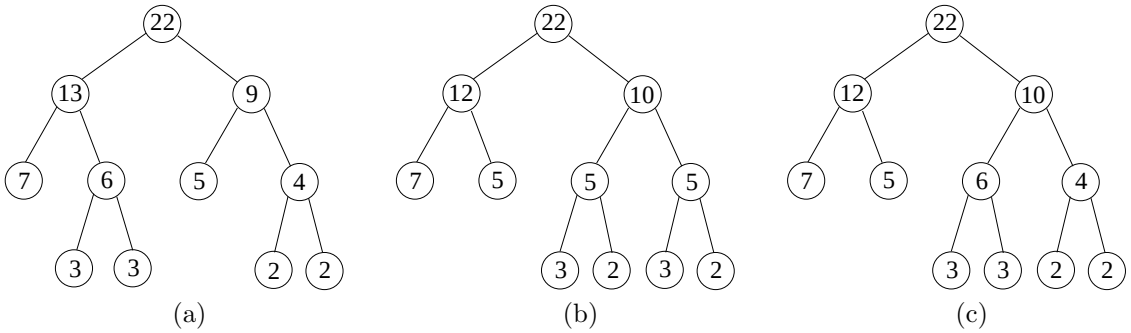

FIGURE 1: *Optimal trees*

Obviously, not paying at all attention to the order of the weights can yield very bad encodings. Consider a typical sequence of weights yielding a maximally skewed tree, that is, a tree with one leaf on each level (except the lowest level, on which there are two leaves). The Fibonacci sequence is known to be the one with the slowest increasing pace among the sequences giving such a biased tree [7], but for the ease of description we shall consider the sequence of powers of 2, more precisely, the weights 1, 1, 2, 4, $\ldots, 2^n$, for some n.

Applying regular Huffman coding to this sorted sequence, we get

$$S_{\text{Huf}} = (n+1) + \sum_{i=0}^{n}(n-i+1)2^i = 2^{n+2} - 2$$

as total size of the encoded file. If one uses the same skewed tree, but assigns the codewords in reverse order, which can happen if the initial sequence is not sorted and the tree is built without any comparisons between weights, the size of the encoded file will be

$$S_{\text{rev}} = 1 + \sum_{i=0}^{n}(i+2)2^i - 2^n = (n+1)2^{n+1} - 2^n + 1.$$

The ratio $S_{\text{rev}}/S_{\text{Huf}}$ may thus increase linearly with n, the size of the alphabet.

We therefore turn to a more realistic scenario, in which some partial ordering is allowed, but requiring an upper bound of $O(n)$ order operations, as opposed to $\theta(n \log n)$ for a full sort. Indeed, the simplest implementation of Huffman coding, after an initial sort of the weights, is keeping a sorted linked list, and repeatedly removing the two smallest elements and inserting their sum in its proper position, overall a $\theta(n^2)$ process. Using two queues Q_1 and Q_2, the first for the initial weights and the other for those created by adding two previous weights, the complexity can be reduced to $O(n)$ because the elements to be inserted into Q_2 appear in order [9]. If one starts with a sequence which is inversely sorted, the first element to be inserted into Q_2 will be the largest; hence if one continues as in the original algorithm by extracting either the two smallest elements of Q_1, or those of Q_2, or the smallest from Q_1 and that of Q_2, the first element of Q_2 will be used again only after the queue Q_1 has been emptied. The resulting tree is thus a full binary tree, with all its leaves on the same level if n is a power of 2, or on two adjacent levels if not. The depth of this tree, for the case $n = 2^k$, will be k. Returning to the above sequence of weights, the total size of the encoded file will thus be

$$S_{\text{fixed}} = \log n \left(1 + \sum_{i=0}^{n} 2^i\right) = 2^{n+1} \log n.$$

The ratio $S_{\text{fixed}}/S_{\text{Huf}}$ still tends to infinity, but increases only as $\log n$ as opposed to n above.

One of the ways to get some useful partial ordering in linear time is the one used in Yao's Minimum Spanning tree algorithm [12]: a parameter K is chosen, and the set of weights W is partitioned into K subsets of equal size $W_1, \ldots, W_K$, such that all the elements of W_i are smaller than any element in W_{i+1}, for $i = 1, \ldots, K-1$, but without imposing any order within each of the sets W_i. The total time for such a partition is only $O(n \log K)$, using repeatedly an $O(n)$ algorithm for finding the median first of the whole set W, then of its two halves (the $n/2$ lower and the $n/2$ upper values), then of the quarters, etc. Starting with such a partition and continuing with the help of two queues, one gets an overall linear algorithm, since K is fixed. On the other hand, K can be used as a parameter of how close the initial ordering should be to a full sort.

To empirically test this partition approach, we chose the following input files of different sizes and languages: the Bible (King James version) in English, and the

	1-grams	2-grams	3-grams	4-grams
English	52	808	6026	21886
French	131	2965	18864	56078

TABLE 1: *Alphabet sizes*

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 [10]. To get also different alphabet sizes, the Bible text was stripped of all punctuation signs, whereas the French text has not been altered. We then also considered extended alphabets, consisting of bigrams, trigrams and 4-grams, that is, the text was split into a sequence of k-grams, $1 \leq k \leq 4$, and for fixed k, the set of the different non-overlapping k-grams was considered as an alphabet. Table 1 shows the sizes of the alphabets so obtained.

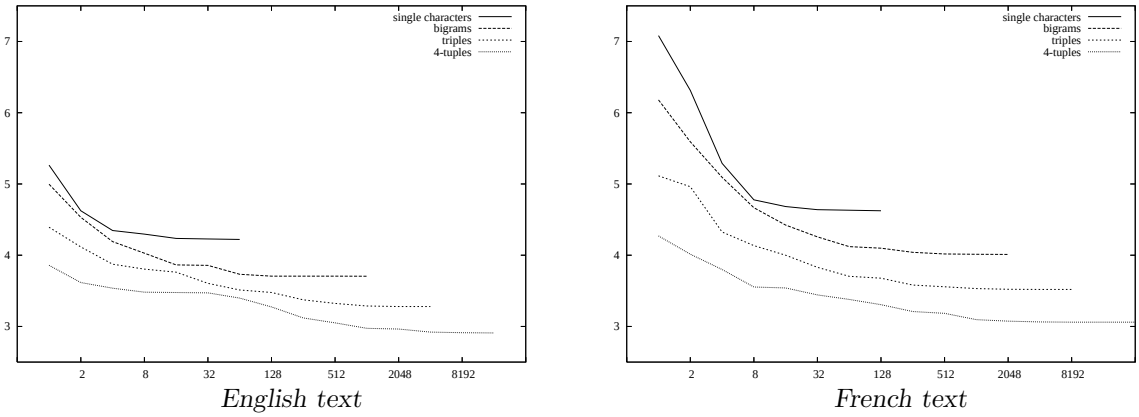

FIGURE 2: *Average number of bits per char as function of number of blocks in partition*

Each sequence of weights was then partitioned as explained above into K equal parts, with $K = 1, 2, 4, 8, \ldots$, where in each part the original lexicographic order of the elements has been retained. Figure 2 plots the average number of bits needed to encode a single character as function of the number of partition parts K. All the plots exhibit a decreasing trend and obviously converge to the optimum when K reaches the alphabet size, but it should be noted that the convergence pace is quite fast. For example, for the 4-tuple alphabets, using $K = 1024$ corresponding to 10 partition phases, there is a loss of only 1.1% for the English and 2.2% for the French texts over the optimal Huffman code.

Another kind of partial ordering relates to a dynamic environment where the Huffman trees to be used are constantly updated. An application of this idea to a packet transmission system is discussed in the next section.

3 Dynamic compression of a sequence of data packets

Consider a stream of data packets $P_1, P_2, \ldots$ of varying sizes, which should be transmitted in compressed form over some channel. In practice, the sizes have great variability, ranging from small packets of several bytes up to large ones, spanning Megabytes. Compression of packet P_t will be based on $P_{t-k}, P_{t-k+1}, \ldots, P_{t-1}$, where

k could be chosen as $t-1$ if one wishes to use the full history, or as some constant if the compression of each packet should only depend on the distribution in some fixed number of preceding packets.

Normally, after having processed P_t, the distribution of the weights should be updated and a new Huffman tree should be built accordingly. The weights of elements which did not appear earlier are treated similarly to the appearance of new elements in dynamic Huffman coding. We suggest, however, to base the Huffman tree reconstruction not on a full sort of the updated frequencies, but on a partial one obtained from a single scan of a bubble-sort procedure. For the formal description, let s_i, $1 \leq i \leq n$, be the elements to be encoded. These elements can typically be characters, but could also be pairs or triplets of characters as in the example above, or even words, or more generally, any set of strings or more general elements, as long as there is some unambiguous way to partition the text into a sequence of such elements. Let $f(s_i)$ be the frequency of s_i and note that we do not require the sequence $f(s_1), f(s_2), \ldots$ to be non-decreasing. The update algorithm to be applied after each block is:

```
Update after having read P_t:
    for i  ⟵  1 to n
        add frequency of s_i within P_t to f(s_i)
        subtract frequency of s_i within P_{t-k} from f(s_i)
    for i  ⟵  1 to n − 1
        if f(s_i) > f(s_{i+1})  swap(s_i, s_{i+1})
    Build Huffman tree for sequence (f(s_1), f(s_2), ..., f(s_n)) using two queues
```

The gain of using only a single iteration of possible swaps is not only in processing time. It also allows a more moderate adaptation to changing character distributions in the case of the appearance of some very untypical data packets. Only if the changed frequencies persist also in several subsequent packets, will the Huffman tree gradually change its form to reflect the new distributions. On the other hand, if the packets are homogeneous, the procedure will zoom in on the optimal order after a small number of steps.

To simulate the above packet transmission algorithm, we took the English and French texts mentioned earlier, and partitioned them into sequences of blocks, each representing a packet. For simplicity, the block size has been kept fixed. The tests were run with single character and bigram alphabets. The following methods were compared:

1. Blocked – Block encoding: each block uses the Huffman tree built for the cumulative frequencies of all the preceding blocks to encode its characters.
2. Bubble – Using one bubble-sort iteration: each block uses the cumulative frequencies of all previous blocks as before, but after each block, only a single bubble-sort iteration is performed on the frequencies instead of sorting them completely. Huffman's algorithm is then applied on the non-sorted sequence of weights.
3. Bubble-For-k – Forgetful variant of Bubble: each block uses the cumulative frequencies not of all, but only the k previous blocks ($k \geq 0$). The frequencies of blocks that appear more than k blocks earlier are thus not counted for building the Huffman tree of the current block. This allows a better adaptation in case of heterogeneous blocks, at the price of slower convergence in the case of a more uniform behavior of the character distributions within the blocks.

For the last case we considered both Bub-For-1 and Bub-For-5, using the frequencies of the preceding block only and of the last five blocks, respectively. The first block was encoded with a fixed length code using the full single character or bigram alphabet. After each block read, the statistics were updated and a new code was generated according to the methods above. The recorded time is that of the average code construction time per block, not including the actual encoding of the block.

Single characters		Block size	Blocked Bubble		Bubble For-1	Bubble For-5
English	Compression	200	4.112	5.532	5.697	5.607
		2000	4.114	5.532	5.553	5.541
		10000	4.123	5.533	5.536	5.533
	Time	200	0.13	0.06	0.06	0.06
		2000	0.63	0.44	0.27	0.27
		10000	2.56	1.32	1.13	1.26
French	Compression	200	4.699	6.020	5.901	5.875
		2000	4.700	6.020	5.877	5.825
		10000	4.705	6.022	5.834	5.865
	Time	200	0.27	0.09	0.09	0.11
		2000	0.49	0.30	0.30	0.31
		10000	1.47	1.26	1.28	1.28

TABLE 2: *Dynamic compression of data packets using single characters*

Single characters		Block size	Blocked Bubble		Bubble For-1	Bubble For-5
English	Compression	2000	3.805	5.061	5.061	5.061
		10000	3.805	5.061	5.061	5.062
		20000	3.806	5.062	5.062	5.062
	Time	2000	30.1	7.3	9.0	11.6
		10000	34.9	9.2	10.8	13.4
		20000	37.4	11.1	12.9	15.2
French	Compression	2000	4.109	6.343	6.345	6.345
		10000	4.109	6.342	6.344	6.344
		20000	4.108	6.342	6.345	6.342
	Time	2000	286.2	9.9	11.3	14.0
		10000	286.6	11.1	12.9	16.1
		20000	290.4	13.4	15.1	17.6

TABLE 3: *Dynamic compression of data packets using bigrams*

Table 2 brings the results for the single character alphabets and Table 3 the corresponding values for the bigram alphabets. The block sizes used were 200, 2000 and 10000 for the single characters and 2000, 10000 and 20000 for the bigrams. The compression figures are given in bits per character and the time is measured in milliseconds.

As can be seen, there is a significant loss, on our data, in compression efficiency, when using non-sorted frequencies. The block size seems not to have an impact on the compression. For the bigrams, there is also no difference between the forgetful variants and that using all the preceding data blocks, but for the smaller single

character alphabets, the compression using only the information of the few last blocks is marginally better on the French text, and worse on the English one. This can be explained by the different nature of the texts: The English Bible is one homogeneous entity, and its partition into blocks is purely artificial. We may thus expect that using more global statistics will yield better compression performance. The French text, on the other hand, consists of many independent queries and their answers, covering a very large variety of topics. Using the distribution of one block to compress a subsequent one may thus not always yield good results, so a variant which is able to "forget" a part of what it has seen, may be advantageous in this case.

The loss in compression is compensated by savings in sorting time. These savings are more pronounced for the larger bigram alphabets, but also noticeable for the character alphabets. The time is increasing with the size of the blocks, because a larger block gives more possibilities for a larger variability of the frequencies. The exception here is for the bigrams of the French text: the alphabet in this case is so large, that the block size has only a minor impact on the processing time. On the other hand, it is in this case that the savings using partial order are the most significant.

4 Relevance of partial sort to other compression schemes

We check in this section whether the idea of not fully sorting the frequencies could be applicable to other compression methods.

4.1 Arithmetic coding

In fact, for both encoding and decoding using an arithmetic coder [11], the weights need not be in any specific order, as long as encoder and decoder agree upon the same. This has the advantage for the dynamic variant, that the same order of the elements can be used at each step, for example that induced by the lexicographic order of the elements to be encoded. Partial ordering is thus not relevant here.

4.2 256-ary Huffman codes, (s, c)-dense codes, Fibonacci codes

All these codes can be partitioned into blocks of several codewords having all the same length. For 256-ary Huffman, the codeword lengths are multiples of bytes, so that even for very large alphabets, it is very rare to get codewords longer than 3 or 4 bytes; the same is true for (s, c)-dense codes. It follows that, almost always, all the codewords can be partitioned into 3 or 4 groups, so a full sort is not even necessary. It suffices to partition the weights into these classes, as suggested above, just that the sizes of the blocks of the partition are not equal, but rather derived from the specific code.

For Fibonacci codes [5,8], there are F_n codewords of length $n + 2$, where F_i are Fibonacci numbers, and this set is fixed, just as for (s, c)-codes. The number of blocks here is larger, but even for an alphabet of one million characters, there are no more than 29 blocks, and the partition can be done in 5 iterations.

4.3 Burrows-Wheeler Transform (BWT)

At first sight, partially sorting seems to be relevant to BWT [3], as the method works on a string of length n and applies all the n cyclic rotations on it, yielding an

$n \times n$ matrix which is then lexicographically sorted by rows. The first column of the sorted matrix is thus sorted, but BWT stores the *last* column of the matrix, which together with a pointer to the index of the original string in the matrix lets the file to be recovered. The last column is usually not sorted, but it often is very close to be sorted, which is why it is more compressible than the original string. The BWT uses a move-to-front strategy to exploit this nearly sorted nature of the string to be compressed.

One could think that since the last column is anyway only nearly sorted, then if the initial lexicographic sort of the matrix rows is only partially done, the whole damage would be that the last row will be even less sorted, so we would trade compression efficiency for time savings. However, the reversibility of BWT is based on the fact that the first column is sorted, so a partial sort would invalidate the whole method and not just reduce its performance.

5 Conclusion

We have dealt with the simple idea of not fully sorting the weights used by Huffman's algorithm, expecting some time savings in applications where the sort is a significant part of the encoding process. This may include large alphabets, or using several alphabets like in dynamic applications, or when encoding according to a first order Markov chain. The tests showed that by using partial sorts, the execution time can be reduced at the cost of some loss in compression efficiency.

References

1. Brisaboa, N. R., Fariña, A., Navarro, G., and Esteller, M. F.: *(S,C)-dense coding: an optimized compression code for natural language text databases.* Proc. Symposium on String Processing and Information Retrieval SPIRE'03, 2857 2003, pp. 122–136.
2. Brisaboa, N. R., Iglesias, E. L, Navarro, G., and Paramá, J. R: *An efficient compression code for text databases.* Proc. European Conference on Information Retrieval ECIR'03, 2633 2003, pp. 468–481.
3. Burrows, M. and Wheeler, D. J.: *A block-sorting lossless data compression algorithm.* Technical Report SRC 124, Digital Systems Research Center, 1994.
4. de Moura, E. S., Navarro, G., Ziviani, N., and Baeza-Yates, R.: *Fast and flexible word searching on compressed text.* ACM Trans. on Information Systems, 18 2000, pp. 113–139.
5. Fraenkel, A. S. and Klein, S. T.: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.
6. Huffman, D.: *A method for the construction of minimum redundancy codes.* Proc. of the IRE, 40 1952, pp. 1098–1101.
7. Katona, G. H. O. and Nemetz, T. O. H.: *Huffman codes and self-information.* IEEE Trans. on Information Theory, IT–11 1965, pp. 284–292.
8. Klein, S. T. and Kopel Ben-Nissan, M.: *Using Fibonacci compression codes as alternatives to dense codes.* Proc. Data Compression Conference DCC–2008, 2008, pp. 472–481.
9. Van Leeuwen, J.: *On the construction of Huffman trees.* Proc. 3^{rd} ICALP Conference, 1976, pp. 382–410.
10. Véronis, J. and Langlais, P.: *Evaluation of parallel text alignment systems: The ARCADE project.* Parallel Text Processing, J. Véronis, ed., 2000, pp. 369–388.
11. Witten, I. H, Neal, R. M., and Cleary, J. G.: *Arithmetic coding for data compression.* Comm. of the ACM, 30 1987, pp. 520–540.
12. Yao, A. C. C.: *An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees.* Inf. Processing Letters, 4 1975, pp. 21–23.