Synchronizing Dynamic Huffman Codes

Shmuel T. Klein[1], Elina Opalinsky[2], and Dana Shapira[2]

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

[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@g.ariel.ac.il, elina.opalinsk@ariel.ac.il

Abstract. Traditional dynamic Huffman algorithms update the frequencies adaptively after every character, according to the assumption that better compression can be achieved when all previous characters are taken into account, justifying the slow processing time. This, however, turns the encoded file into an extremely vulnerable one in the case of even a single bit error. Since the above mentioned assumption is not necessarily true, we explore blockwise dynamic Huffman variants, where the Huffman tree is periodically, rather than constantly, updated. Experiments show that avoiding the updates at every character and choosing larger blocks does not hurt the compression performance, and may even improve it at times. Moreover, the new scheme seems to be more robust against single errors introduced in the encoded file.

1 Introduction

One of the oldest, yet still popular, data compression techniques is Huffman coding [5]. The focus of the current research is on its dynamic version, where the code gets repeatedly updated while more characters of the input file are being processed. In particular, we consider the case that the compressed file has been transmitted over a network, and a communication error occurred within the encoded file, causing a change in one of its bits.

Given an input file which we shall call a text, even though the algorithm applies also to non-textual data, a first step is to decide how to parse the text into a set of *elements* to be encoded. Typically, these elements can be the characters of some standard alphabet Σ, like ASCII, but one could as well encode character pairs or triplets, or even words or phrases or word fragments, as long as there is a well-defined way to perform the parsing unambiguously.

This parsing can be used to derive the set of frequencies $\{w_1, \ldots, w_n\}$ of the n distinct elements of the text. Huffman's algorithm then assigns codeword lengths $\{\ell_1, \ldots, \ell_n\}$ to the corresponding elements of Σ, so that the average weighted length $\sum_{i=1}^{n} w_i \ell_i$ is minimized, yielding a minimum redundancy code. The code may be represented by a binary tree known as a *Huffman tree*, whose leaves are associated with the elements of the alphabet Σ. Edges in the tree pointing to a left or right child are labeled by 0 or 1, respectively, and the concatenation of the labels on the path from the root to a given leaf yields the corresponding codeword.

Static Huffman encoding thus requires a double pass over the data: the first for gathering the statistics on the distribution of the elements, on the basis of which the code can be built, and the second for the actual encoding process, once the code is given.

An alternative to this two-pass procedure could be an adaptive method in which both encoder and decoder maintain, independently, a copy of the current Huffman

Shmuel T. Klein, Elina Opalinsky, Dana Shapira: Synchronizing Dynamic Huffman Codes, pp. 27–37.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

tree, which is based on the frequencies of the elements processed so far. The trivial, and very costly, solution would be to reconstruct the Huffman tree from scratch in each iteration. Faller [1], Gallager [3] and Knuth [10] proposed, independently, essentially the same improved one-pass method, known as the FGK algorithm, for generating a Huffman tree according to the probabilities of the characters in the already processed portion of the file. The FGK algorithm was later enhanced by Vitter [15], who maintains the adaptive model and creates the encoding "on the fly".

One of the issues of transferring data over a network, is the ability to synchronize when communication errors occur. In [6], the problem of searching directly within a Huffman compressed file is investigated. Given a pattern P and a compressed file, instead of decompressing and searching for P in the decompressed file, the compression algorithm is applied to the pattern, and the resulting encoding is sought within the given encoded file. To announce a match of the pattern in the original file, the problem is to verify whether the detected occurrences are aligned on codeword boundaries. The algorithm proposed in [7] is based on the tendency of Huffman codes to resynchronize quickly after errors, even if decoding does not start at the beginning of a codeword [9].

Consider the case of a single bit error occurring during the transmission of the encoded file. In case the single error is a bit flip, and the file was encoded using some fixed length code, only a single codeword is affected. But in the case of a variable length code, like Huffman's, or if the error is the addition or deletion of a bit, synchronization with the correct decoding may also be lost. For example, let a part of the alphabet be $\{\ldots, \mathtt{M}, \mathtt{N}, \mathtt{O}, \mathtt{S}, \mathtt{U}, \ldots\}$ with codewords $\{\ldots, 0001, 1011, 010, 0010, 10010, \ldots\}$, respectively, and suppose the first bit of the encoding representing the string `MOON` has been lost. The correct decoding, is given in the lower part of Figure 1, whereas the erroneous decoding, `SUN`, is given in the upper part. Note that the codeword boundary just before `N`, in both decodings, is a synchronization point. We see that even a single bit error may have dramatic consequences, literally changing night into day...

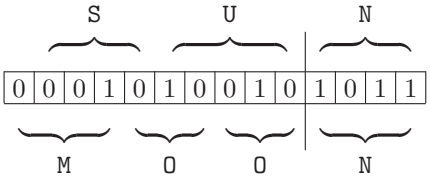


Figure 1. Synchronization in Static Huffman Encoded Files

Generally, suppose an error has occurred in one of the bits of the encoded file, which causes the decoding to be incorrect, and denote by c_1 the last codeword of the resulting wrong decoding before the synchronization point. Let c_2 be the last codeword of the correct decoding before the synchronization point. Then either c_1 must be a proper suffix of c_2 or vice versa. A code is said to have the *affix* property, if none of its codewords is either a prefix or a suffix of any other codeword. Therefore, the scenario described above about the codewords c_1 and c_2 ending both at a synchronization point is not possible if the code is affix, which is why such codes are also called *never-self-synchronizing* in [4]. But such affix codes are extremely rare [2].

Previous research [6] empirically shows that synchronization is regained after typically a few tens of bits. However, the situation is much more involved when dynamic Huffman coding is considered, due to the adaptive nature of the underlying model.

This paper modifies the dynamic Huffman coding procedure, so that the resulting algorithm is more robust against errors when compared to the original dynamic encoding. In fact, we generalize the dynamic Huffman compression of Vitter so that the resulting encoding can mostly synchronize with the correct decoding after only a few codewords. The paper is structured as follows. Section 2 presents the difficulty of the dynamic Huffman algorithm of Vitter [15] to synchronize after a single bit error. Section 3 proposes a generalized dynamic version which is more robust. Section 4 presents experimental results, and Section 5 concludes.

2 Synchronization in Dynamic Huffman Encoding

Dynamic Huffman encoding is highly vulnerable to occurrences of bit errors. In fact, even a single incorrect bit might change the dynamic Huffman tree in a way that will damage the remaining decoding completely.

Figure 2 visually presents an example of the effect of a bit flip introduced in the dynamic Huffman encoded file. The left side of Figure 2 is the decoding of a correctly encoded file (an excerpt from *Alice in Wonderland*), and the right side of Figure 2 is the decoding of the same file in which a *single* bit has been flipped. Characters that were decoded correctly appear in the same font as their correct counterparts, whereas incorrectly decoded characters are colored in red and `boldfaced`. As can be seen in this example, an error may trigger a snowball effect.

The single bit flip causes the character `x` to be mistakenly decoded as `F`, but then synchronization is seemingly regained, as for static Huffman coding. However, at this stage, the frequency counts of the two decodings already differ for certain characters, albeit only slightly. This small difference will trigger, 38 decoded characters later, another erroneous decoding, in which `xt_thing_w` will be substituted by `pnlocilk`, where we use the underscore `_` to visualize a blank. There are thus even more different updates, and the following wrong decoding is `plained_that_they_could_not_tas` which is replaced by `wh_iothn_o-n_oW nTay_ndtnlnha`. Clearly, the initially small changes have a cumulative impact, which materializes as gradually shorter correct stretches separating increasingly longer wrong ones, until the decoding becomes completely erroneous. In several other examples we tested, the situation is even worse, and the divergence is immediate rather than gradually as in Figure 2.

The following small example illustrates the difficulty of Vitter's algorithm to cope with errors. Consider the bit stream $\cdots 101111 \cdots$. The dynamically changing Huffman tree after having read these bits is presented in Figure 3. A dynamic Huffman tree is represented with a pair (*freq*, *char*) in its leaves, where *char* is the character represented by the leaf and *freq* is its frequency in the file so far. Each internal node contains the sum of the frequencies stored in its two children. A special leaf, labeled NYT, is used when a new, Not-Yet-Transmitted, symbol is detected, and its frequency is defined as zero.

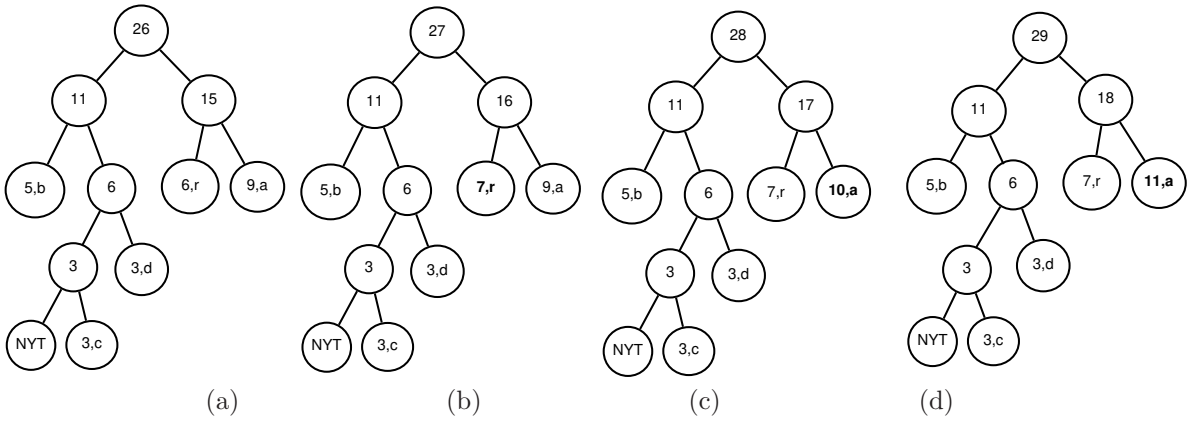
Suppose that at the beginning of the decoding process of these bits, the tree is the one given in Figure 3(a). Processing the first codeword 10, causes an increment of the number of occurrences of `r` from 6 to 7, as shown in Figure 3(b). The following bits are parsed into two consecutive codewords, 11, incrementing the number of the occurrences of `a` from 9 to 10 in Figure 3(c) and finally from 10 to 11, given in Figure 3(d).

Suppose that the encoded file has been corrupted and the second of the shown bits was flipped, so that the entire bit stream consists of 1s. The erroneous decoding

There was exactly one a-piece, all 'round.
The next thing was to eat the comfits; this caused some noise and confusion, as the large birds complained that they could not taste theirs, and the small ones choked and had to be patted on the back. However, it was over at last and they sat down again in a ring and begged the Mouse to tell them something more.
"You promised to tell me your history, you know," said Alice, "and why it is you hate–C and D," she added in a whisper, half afraid that it would be offended again.
"Mine is a long and a sad tale!" said the Mouse, turning to Alice and sighing.
"It_is_ a long tail, certainly," said Alice, looking down with wonder at the Mouse's tail, "but why do you call it sad?" And she kept on puzzling about it while the Mouse was speaking, so that her idea of the tale was something like this:–

There was e**F**actly one a-piece, all 'round.
The ne**Pnloci lr**as to eat the comfits; this caused some noise and confusion, as the large birds com**wh iothn o-n oWnTay ndtnlnha**te theirs, and the small ones choked and had to be patted on the back. **Ot"**ever, it was over at last and they sat down again in a ring and begged the **fmnrihdo** tell them something more.
"Wdnr v omised to tell me your history **uinrwan-heqe hint** Alice-**ks**nd why it is you hate**N ohe** and **Rkw** she added in a **pccwen cssd, a,rail** that it **baDh rfthfgnojs**gain**;e " o,othac a dv dita a sao laleAk hioa** the **o,** use-**l**urning **tll oose ano ecighingOe:e RNiisAhs d. lnint-Nn niorodkp aaio u tice- o lewi .n mr eane r.ton Urthe Aa Eov- nint'k e vrbad or aa jaadd** it sa**okcflttlea hwketr.genvhn ding aeY, it bcys olh h-ha rae frciSi nawl o,l hsdsi,a ve th etals se m msoci dtifc oceng"le eeed Sfu. y esil t,e a mdyhac j hoed**

Figure 2. Bit Flip in Dynamic Huffman Encoded Texts



(a) (b) (c) (d)

Figure 3. Correct decoding

is given in Figure 4. The parsing is now 11-11-11, thus incrementing the number of occurrences of a from 9 to 10, then from 10 to 11, and finally from 11 to 12, presented in Figures 4(a), (b) and (c), respectively. Before the number of occurrences of a changes to 12, the shape of the Huffman tree gets updated, and its underlying structure changes: before incrementing the contents of the leaf y with frequency 11, and that of all its ancestors, Vitter's algorithm calls for interchanging this leaf with the highest ranked node also containing the frequency 11, if there is one. The rank of a node is its index in a bottom-up, and within each level left-to-right, enumeration of all the nodes. In our case, the left child of the root has also frequency 11, so the left subtree and the leaf y are swapped, yielding the tree in Figure 4(d). Obviously, the following bits are completely out of synchronization.

Although the erroneous bit belongs to a single codeword, it directly affects the frequencies of *two* different codewords to start with, unlike for static Huffman coding, and might also trigger two snowball effects, so that the entire decoding may be

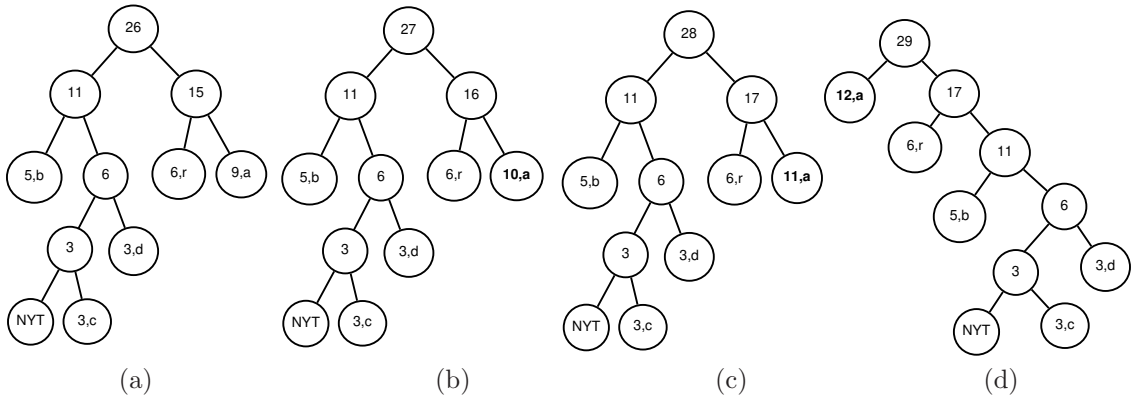


Figure 4. Incorrect decoding

damaged even faster. Going back to our example, the single error flipping 10 into 11 did not only reduce the number of occurrences of the symbol r corresponding to 10, but also increased the number of appearances of the codeword 11, corresponding to a. Since even for static Huffman coding a few codewords might be lost until synchronization is regained, each such mistaken codeword boundary again might cause an incorrect calculation of the frequencies of two codewords. Empirically checking this phenomenon on our datasets, we found that the structure of the Huffman tree is often changed before synchronization is attained, causing a complete loss of the remaining decoding. To overcome the problem we propose a relaxed variant of the dynamic Huffman compression that has the ability to cope with such errors, and generally synchronizes with the correct decoding after only a few codewords.

3 Proposed Algorithm

The main idea of our algorithm is to blur the frequencies with the objective of turning them to less sensitive to isolated errors. This is done on the one hand, by spacing out the updates to be done at the end of a *block* of several characters, and on the other hand by periodically rescaling the accumulated frequencies, for example, dividing each of them by 2. To avoid zero-frequencies, which would force a special treatment for disappearing and reappearing characters, one could initialize all frequencies with 1 and use upward rounding in the scaling, so that no frequency will fall below 1.

The rationale is the following. Spacing out the updates creates blocks within which the algorithm behaves essentially like static Huffman coding. There is thus a chance, if the error is not too close to a block boundary, that synchronization is regained even before it has changed the shape of the tree. As to scaling, if at a certain point, the distribution of the frequencies is $\{w_1, w_2, \ldots, w_n\}$, then considering $\{\lceil w_1/2 \rceil, \lceil w_2/2 \rceil, \ldots, \lceil w_n/2 \rceil\}$ instead will generally produce practically the same Huffman tree. Indeed, the shape of a Huffman tree is mainly determined by the *relative* sizes of the frequencies, rather than by their absolute values, which is why probabilities can be used instead of frequencies. Rescaling can thus achieve a triple goal:

1. it helps keeping the involved frequencies within bounded limits;
2. it gives higher weights to recently seen characters, as the last frequencies are divided by 2, but those accumulated in the block before are divided by 4, and,

generally, the number of occurrences of any character in block i when counted backwards from the last one, is divided by 2^i;

3. small fluctuations of ± 1 in the frequencies may either be corrected by the rounding, but even if not, there are good chances that the resulting Huffman trees are identical. If so, the error will not propagate and its effect may be corrected at the next synchronization point.

The fact that different, yet similar, frequency distributions may yield the same Huffman tree has been investigated by Longo and Galasso [12]: the set of probability distributions over a finite alphabet is given a "pseudometric", and an upper bound is derived for the distance from any probability distribution to the *dyadic* distribution (in which all the probabilities are powers of $\frac{1}{2}$) giving the same Huffman tree.

On the other hand, the disadvantage of rescaling seems to be that the construction of the Huffman tree will then not rely on true frequencies, but on approximate ones, which, would one think, might hurt the optimality of the Huffman procedure. It should however be kept in mind that Huffman codes are optimal only for static frequencies. In the adaptive variant, actually not only for Huffman, but also for arithmetic coding and for intrinsically adaptive methods like those of Ziv and Lempel [16,17], the basic assumption is that the distribution of elements in the text seen so far (the past) is identical, or at least a good estimate for, the distribution after the current point (the future), but there is no guarantee for such an assumption to be true!

The orthodox adherence to the exact frequencies is thus not necessarily justifiable, and it could well be that an approximation can produce results that are not inferior, and maybe even better at times. A similar observation has been mentioned in [8] in an application basing adaptive arithmetic coding on randomly chosen n out of $2n$ preceding characters, rather than the most recent n, without incurring any noticeable loss.

Once it has been agreed that our knowledge of the frequencies of the characters in the processed text does not need to be perfect, this reinforces the idea that we may change the constant updates after each read character, as advocated, e.g., in Vitter's algorithm, by periodic ones, to be performed only at the end of each block of b characters, for some fixed block size b. This has the obvious advantage of speeding up both compression and decompression, and our empirical results show that the expected loss is very low, much less than 1% on all our tests, even with large blocks like $b = 16K$. There were even instances in which the blockwise processing gave better compression than Vitter's variant, which corresponds to $b = 1$.

A first guess would be that the larger the blocksize b will be chosen, the less accurate our estimate will be, which is consistent with our assumption of dealing with a tradeoff: since a larger block implies obvious time savings, it is reasonable to assume that this gain in time comes at the price of a certain loss in compression efficiency. Our results, however, show that often, just the opposite is true! While for large blocks, the increase, if there was one, in the size of the compressed file was very small, it was for the small blocksizes that a significant loss occurred, increasing the file size by tens of percents. For example, for the two test files mentioned in the next section, the file size grew, for a block of size $b = 16$, by 22.4% and 44.8%, while for $b = 1K$, the increase was only by 0.2% and 2.2%, respectively.

To understand this behavior, recall that we start the dynamic encoding by assuming a uniform distribution of the characters, a quite arbitrary initialization which does not really matter if the blocksize is large enough. Recall also that all frequen-

cies are rescaled at the end of each block, so that the influence of the distribution of characters which are several blocks behind the current point is quickly vanishing. If the blocksize itself is small, only a part of the alphabet will appear in these few preceding blocks, and the frequencies on which the current encoding will be based will still include many elements with the initial frequency 1. For large enough blocks, such low frequency elements will practically have no influence, but for small blocks, they might be dominant, yielding an overall distribution which is still close to uniform, and therefore far from the real distribution within the input text.

Algorithm 1: Compression Algorithm

GENERALIZED DYNAMIC HUFF(T, b)
1 $HT \leftarrow$ Huffman Tree for uniform distribution of Σ
2 **while** input T is not exhausted **do**
3 **encode** the following b characters according to HT
4 **update** frequencies of Σ according to the last b characters
5 **divide** all frequencies by 2, rounding up
6 **update** HT according to updated frequencies

The formal algorithm, with parameters T, the input text, and b, the block size measured in number of characters, is given below. The tree reconstructed in the decompression phase maintains the same distribution as in the compression phase, and by agreeing to construct canonical trees [13] and keeping the symbols at each level in some agreed order, e.g., lexicographically, encoder and decoder are able to reconstruct the same tree.

4 Experimental Results

We applied our experiments on two text files:

1. the Bible (King James version) in English, denoted by *ebib*, over an alphabet of 52 characters, in which the text has been stripped of all punctuation signs except blank;
2. the French version of the European Union's JOC corpus, denoted by *ftxt*, which is a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [14], over an alphabet of 127 characters.

Table 1 presents the compression performance of the various algorithms on both data files. The second, third and fourth columns present the original file size, the size of the compressed file, using static Huffman, and the file size after applying Vitter's algorithm. The last columns give the sizes of the generalized dynamic algorithm for block sizes 2^iK, $0 \leq i \leq 4$, and the results when also division is applied is given in the line below. All figures are given in Bytes. The best compression results for each file are highlighted in bold. As can be seen, the efficiency of the blockwise compression algorithm is not inferior, on these examples, to the original one updating after each processed character, in both variants, with and without division. As fewer Huffman trees are constructed throughout the execution of the algorithms when larger blocks are considered, the savings in processing times are also obvious.

In order to assess the synchronization abilities of the different algorithms, several measures should be considered. The problem lies in the fact that the damage caused

File	**Original**	**Static**	**Vitter**	**Generalized**				
				K	2K	4K	8K	16K
ebib	3711020	1942474	1942008	1942247	1942628	1943383	1944885	1947814
				Generalized & Division				
				1945709	1942012	**1941187**	1941847	1948863
ftxt	7610765	4399422	4379161	**4378724**	4379065	4379748	4381080	4383696
				Generalized & Division				
				4476947	4429719	4411830	4406138	4406566

Table 1. Compression performance - ebib

by a single erroneous bit during the decoding process may be judged on different levels, each of which requires another definition. Indeed, assume a single such error has occurred, then at least one, and possibly several, codewords will be falsely interpreted. However,

1. the fact that there were wrong interpretations means that some frequencies will be incorrect, but this does not necessarily mean that the corresponding Huffman trees have changed; the damage might thus be locally restricted and have no long range impact.
2. Even if the changes in the frequencies are extended enough to trigger also a change in the shape of the trees, the set of codeword lengths, and at times, even the set of codewords themselves, may still remain unchanged, so that one *could* use other Huffman trees which are still identical. For instance, Vitter's algorithm constructs a very specific form of the Huffman tree, and a small perturbation in the frequencies may change its shape altogether, as in the example in Figure 4(d), but had canonical Huffman trees been used instead of Vitter's, these small alterations might have a lesser or no impact at all, and the original and altered frequencies might yield the same canonical tree.
3. Finally, the frequency fluctuations may be significant enough to imply even different canonical Huffman trees.

To deal with the first option, we need a measure D_1 evaluating some "distance" between the erroneous frequency distribution caused by the bit flip, and the correct distribution, according to which the file has been encoded. A well-known such measure is the *Kullback-Leibler* (KL) divergence [11]: for two probability distributions $E = \{e_1, \ldots, e_n\}$, which is possibly erroneous, and $T = \{t_1, \ldots, t_n\}$, which we assume to be the true one, define

$$D_1(E, T) = D_{KL}(E\|T) = \sum_{i=1}^{n} e_i \log \frac{e_i}{t_i}\,.$$

The KL divergence is a one-sided, asymmetric, non-negative distance from E to T, which equals zero if and only if $E = T$.

However, D_1 is not an appropriate measure for our application. First, it depends on the location of where the error has occurred. If this happened close to the beginning

of the file, the impact of a change of ± 1 on the yet small accumulated frequencies will be larger than if the error had occurred significantly later. Moreover, the involved probabilities are all very small, their ratios are close to 1 and overall, on all our tests, the values of the KL divergence were of the order of 10^{-5} to 10^{-10}, from which hardly any conclusion could be derived.

This suggests using a more descriptive measure for the distance between distributions, based on the absolute difference between corresponding frequencies, rather than on the relative one implied by the probabilities. If the erroneous and true frequency vectors are denoted $EF = \{ef_1, \ldots, ef_n\}$ and $TF = \{tf_1, \ldots, tf_n\}$, respectively, we define

$$D_2(EF, TF) = \sum_{i=1}^{n} |ef_i - tf_i|.$$

Table 2 presents the results of comparing the D_2 distances between the distributions produced by erroneous and true decodings by the three algorithms considered herein: Vitter's dynamic Huffman coding, with updates after each processed character, the blockwise dynamic algorithm using cumulative frequencies and blocksize $b = 1024$ encoded characters, and the same but using rescaling by dividing the frequencies by 2 after each block. The table gives the numbers for a typical example, presenting in the column headed i the distance D_2 as measured at the end of the ith block after the error. The test file was *ebib*, in which the first bit of codeword number 380245 was flipped.

File	1	2	3	4	5	6	7	8	9	10	20	30	40	50
Vitter	10	12	12	12	10	12	38	60	60	60	210	2344	4000	6114
blocks – cumulative	8	8	8	8	8	8	8	8	8	8	8	8	8	8
blocks – scaled	5	4	3	2	2	1	0	0	0	0	0	0	0	0

Table 2. Comparing the distance $D_2(EF, TF)$ between erroneous and true decoding

We see that for Vitter's algorithm, the sum of the absolute differences is about 10–12 at the beginning, but then increases exponentially yielding the snowball effect mentioned above. A large distance means that the distributions are completely different, in accordance with the example in Figure 2. On the other hand, the distance for the algorithm processing blocks stays constant at 8 for all the considered blocks, and even for hundreds thereafter. The frequencies, though, do increase gradually, just their difference remains constant, which indicates that synchronization has been regained. For the block variant with scaling, not only is there synchronization, but the difference also is zeroed by the repeated divisions.

As a different number of occurrences of the characters does not necessarily refer to an incorrect decoding, our final experiment is to measure the percentage of successful decodings for the various algorithms. We repeated the bit-flip test as the one reported in Table 2 one hundred times, with different flip positions, and checked not only the distance, but also whether there was ultimately synchronization after the error or not. To avoid a bias in the choice of the bit position of the error, the 2000th bit of 100 different blocks of b characters has been flipped, with $b \in \{1K, 2K, 4K, 8K\}$.

Table 3 presents the number of unsynchronized decodings as a function of block size for all three algorithms. Synchronization has been assessed in this experiment by

Block Size	**1K**	**2K**	**4K**	**8K**
Vitter	100	100	100	100
blocks – cumulative	23	17	17	16
blocks – scaled	22	15	13	12
intersection	4	2	3	7

Table 3. Number of unsynchronized decodings out of 100 tests as a function of block sizes.

comparing the last blocks at the end of the decoded files and checking that they are identical. The number appearing on the intersection line reports the number of cases that are unsynchronized in both blockwise algorithms.

In none of our experiments did Vitter's decoding synchronize with the correct decoding, whereas the block variants did get back on track in about 80% of the cases or more. This figure seems to be improving with growing block sizes. There seems also to be a small improvement of the variant using scaling over that using cumulative frequencies, though the full extent of the improvement might not be caught by our measure: the major advantage of periodically dividing the numbers is that erroneous fluctuations are ultimately "forgotten". This amnesic behavior allows the decoding to synchronize not only the text sent to the output file, but also the Huffman data structure used to enable the decoding. In the cumulative variant, the texts might match even for long stretches, but there is no guarantee that the differing Huffman trees will not, at some later stage, induce errors again.

5 Conclusion

Motivated by the lack of synchronization in the case of the occurrence of even a single bit error in a file that has been compressed by a standard dynamic Huffman code, we explored generalizations that get updated periodically, not necessarily after each character. Two blockwise dynamic variants were considered, with and without scaling, suggesting an improvement in processing time and robustness against single bit errors, without hurting the compression performance.

A synchronization point of a correct and incorrect decoding of static Huffman coding may be defined as the first position after that of the error in the encoded file, for which both decodings reach the root of the tree at the same time. However, locating the synchronization point in the dynamic variants was found to be a bit tricky, as two different Huffman trees should be compared in parallel. Moreover, small fluctuations in the frequencies may trigger later divergences of the decoding, even if temporarily synchronization has been restored. We therefore approximated the distances between both decodings by summing the frequency differences.

References

1. N. Faller: *An adaptive system for data compression*, in Record of the 7-th Asilomar Conference on Circuits, Systems and Computers, 1973, pp. 593–597.
2. A. Fraenkel and S. Klein: *Bidirectional Huffman coding.* The Computer Journal, 33(4) 1990, pp. 296–307.
3. R. Gallager: *Variations on a theme by Huffman.* IEEE Transactions on Information Theory, 24(6) 1978, pp. 668–674.
4. E. Gilbert and E. Moore: *Variable-length binary encodings.* The Bell System Technical Journal, 38, pp. 933–968.
5. D. Huffman: *A method for the construction of minimum redundancy codes.* Proc. of the IRE, 40 1952, pp. 1098–1101.
6. S. Klein and D. Shapira: *Pattern matching in Huffman encoded texts.* Inf. Process. Manage., 41(4) 2005, pp. 829–841.
7. S. Klein and D. Shapira: *Compressed pattern matching in JPEG images.* Int. J. Found. Comput. Sci., 17(6) 2006, pp. 1297–1306.
8. S. 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.
9. S. Klein and Y. Wiseman: *Parallel Huffman decoding with applications to JPEG files.* The Computer Journal, 46(5) 2003, pp. 487–497.
10. D. Knuth: *Dynamic Huffman coding.* Journal of Algorithms, 6(2) 1985, pp. 163–180.
11. S. Kullback and R. Leibler: *On information and sufficiency.* Annals of Mathematical Statistics, 22(1) 1951, pp. 79–86.
12. G. Longo and G. Galasso: *An application of informational divergence to Huffman codes.* IEEE Trans. Information Theory, 28(1) 1982, pp. 36–42.
13. E. Schwartz and B. Kallick: *Generating a canonical prefix encoding.* Commun. ACM, 7(3) 1964, pp. 166–169.
14. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The arcade project, in parallel text processing.* pp. 369–388.
15. J. Vitter: *Design and analysis of dynamic Huffman codes.* Journal of the ACM (JACM), 34(4) 1987, pp. 825–845.
16. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. Information Theory, 23(3) 1977, pp. 337–343.
17. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Trans. Information Theory, 24(5) 1978, pp. 530–536.