A Memory-Efficient AdaptiveHuffman Coding Algorithm for Very Large Sets of Symbols

Stevcn Pigeon
Département d'informatique et de recherche opérationelle
Université de Montréal
pigeon@iro.umontreal.ca

Yoshua Bengio
Département d'informatique et de recherche opérationelle
Université de Montreal
bengioy@iro.umontreal.ca

Abstract

The problem of computing the minimum redundancy codes as we observe symbols one by one has received a lot of attention. However, existing algorithms implicitly assumes that either we have a small alphabet — quite typically 256 symbols — or that we have an arbitrary amount of memory at our disposal for the creation of the tree. In real life applications one may need to encode symbols coming from a much larger alphabet, for e.g. coding integers. We now have to deal not with hundreds of symbols but possibly with *millions* of symbols. While other algorithms use a number of nodes proportional to the number of observed symbols, we here propose one that uses a number of nodes proportional to the number of frequency classes, which is, quite interestingly, always smaller or equal to the size of the alphabet.

1. Proposed Algorithm: Algorithm M

Of all compression algorithms, it is certainly Huffman's algorithm [3] for generating minimum redundancy codes that is the most widely known. The general problem consists in assigning to each symbol s of a certain set S a binary code with an integer length that is the closest possible to $-\lg(\mathrm{p}(s))$, where $\mathrm{p}(s)$ is an estimate of the probability of symbol s, and lg x is a shorthand for $\log_2 x$. Huffman's algorithm generates a set of codes for which the average code length is bounded by the interval $[\mathrm{H}(S), \mathrm{H}(S) + p + 0.086)$, where p is the largest probability of all symbols s in S [8, p. 107]. While being rather efficient, this algorithm has several practical disadvantages.

The first is that it is a *static code*. Huffman's algorithm precomputes the code book C after having obtained an estimate for $\mathrm{p}(s)$ for all s in S, and then the compression proper uses the same code book C for all the data, which sometimes can lead to compression inefficiencies if the data in the sequence are not i.i.d., that is, differs substantially from $\mathrm{p}(s)$. One must not forget that we are using $\mathrm{p}(s)$ instead of a more appropriate $\mathrm{p}_t(s)$, which depends on t, the 'time' or the position in the sequentially processed data. If the code book was adaptive, one might capture some information leading to a better approximation of $\mathrm{p}_t(s)$ which in turn leads to possibly better compression.

The second major disadvantage is that *the algorithm must see all the data* before one can actually perform the compression. Since we are trying to estimate $\mathrm{p}(s)$ for a data set composed of symbols out of S, we must scan all the data in order to get a good estimate. Often, it is quite impractical to do so. One might not have wanted space to store all the data before starting compression or, as it is often the case in communication, 'all the data' might be an illusive concept. If we estimate $\mathrm{p}(s)$ out of a small sample of the data we expose ourselves to the possibility that the estimate of $\mathrm{p}(s)$ out of a small sample is a bad approximation of the real $\mathrm{p}(s)$.

The memory space used by Huffman's algorithm (whether the implementation is memory-efficient or not) is essentially $\mathrm{O}(n)$, where n is the number of symbols in S, which can lead to problems when n is large. In textbooks, one *never* finds an example of code books generated by Huffman's algorithm with more than 256 symbols. In this context, the 'worst case' code book consists of assigning a code to each of the 256 possible bytes. In the real world, however, one might rather need a code book for a set of several thousands, or even of several *million* symbols, like all the m bits long integers. In certain special cases, one can use an off-the-shelf code like the so-called Golomb's codes, or Rice's codes or others [8,9], but it is in fact rare that the distribution function exactly match a geometric distribution function. Golomb's codes are disastrous if used for a distribution that is not geometric! In general, the distribution is incompletely known at best so it might be hard to derive a standard parametric code for it. And, of course, one must generally reject the trivial solution of building up a look up table of several million entries in memory.

Lastly, for the compressed data to be understandable by the decoder, the decoder must know the code book C. Either one transmits the code book itself or the function p(s). In both cases the expense of doing so when the code book is large might be so high as to completely lose the gain made in compression. If S has 256 symbols, as it is often the case, then the cost of transmitting the code book remains modest compared to the cost of transmitting the compressed data which can be still quite large. While actually having a certain loss of compression due to the transmission of the code book, it might still be relatively insignificant and thus still give a satisfying result. But when one is dealing with a very large set of symbols, the code book must also be very large. Even if a very clever way of encoding the code book is used, it might remain so costly to transmit that the very idea of compressing this data using a Huffman-like algorithm must be abandoned.

In this paper, to address the problems of adaptivity, memory management and code book transmission problems, we propose a new algorithm for adaptive Huffman coding. The algorithm allows for very good adaptivity, efficient memory usage and efficient decoding algorithms. We present our algorithm which consists in three conceptual parts: set representation, set migration and tree rebalancing. We finally discuss various initialization schemes and compare our results against the Calgary Corpus and static Huffman coding [3] (the ordinary Huffman algorithm) and Vitter's algorithm Λ [7,12].

1.1. Algorithm M

We will now present an algorithm that is well suited for large sets of symbols. They naturally arise in a variety of contexts, such as in the 'deflate' compression algorithm [15,16] and in JPEG [17]. In the 'deflate' compression algorithm, lengths and positions of matches made by a basic LZ77 compressor over a window 32K symbols long are encoded using a variable length coding scheme thus achieving better compression. In the JPEG still image compression standard, the image is first decomposed by a discrete cosine transform and the coefficients are then quantized. Once quantization is done, a scheme, not unlike the one in Fig. 1. [13], is used to represent the possible values of the coefficients. There are however fewer values possible in JPEG, but still a few thousands. There are many other situations where a large number of symbols is needed.

The algorithm we now propose uses a tree with leaves that represent *sets* of symbols rather than *individual* symbols and uses only two operations to remain as close as possible to the optimal: *set migration* and *rebalancing*. The basic idea is to put in the same leaf all the symbols that have the exact same probability. Set migration happens when a symbol moves from one set to an other set. This is when a symbol, seen m times (and thus belonging to the set of all symbols seen m times) is seen one more time and is moved to the set of symbols seen m+1 times. If the set of symbols seen m+1 times does not exist, we create it as the sibling of the set of symbols seen m times[1]. If the set of symbols seen m times is now empty, we destroy it. Rebalancing, when needed, will be performed in at most O(-lg p(s)) operations. Let us now present in detail this new algorithm.

Let us say that we have the symbols representing the set of the integers from 0 to B. The minimal initial configuration of the tree is a single leaf (which is also the root) containing the set $\{0,...,B\}$, to which a zero frequency is assigned. It can be zero since we don't work directly with the probability, but with the number of times a symbol was seen. At that point, the codes are simply the binary representation of the numbers 0 to B. When the first symbol is seen, say '2', it has a count of 1. Since we don't have a set of symbols seen exactly 1 time, we create it. That gives us a tree that has a root node and two leaves. The codes now have a one bit prefix, followed by either no bits, if it is the (only) symbol in the new set, or $\left\lceil \lg\left(\left|\{0,1,3,...,B\}\right|\right)\right\rceil$ bits (for any other). As other symbols are seen for the first time the set of all symbols seen once grows and the set of all possible but yet unobserved symbols shrinks. When we encounter a symbol a second time, we create the set of symbols seen exactly 2 times, and so on. This situation is shown in fig. 2. Note that the numbers written on nodes are their weights (a parent's weight is the sum of its two children's weight, and the leaves' weight are simply the number of time the symbols in its associated set have been seen so far times the number of symbols in its set). If a region of the tree is too much out of balance, we rebalance it, starting at the position where the new set was inserted.

[1] If at the moment of the insertion of node z, the node x has already a sibling, we create a new parent node t in place of node x, such that its children are the node x and the node z.

Codes	Representable Values
1	0
10 + 1 bit	-1,+1
1100 + 2 bits	-3,-2,+2,+3
1101 + 3 bits	-7,-6,-5,-4,4,5,6,7
11100 + 5 bits	-15,-14,...,-9,-8,8,9,...,13,14,15
11101 + 6 bits	-47,...,-16,16,...,47
etc.	etc.

Fig. 1. An example of code for a large set and a hypothetical distribution.

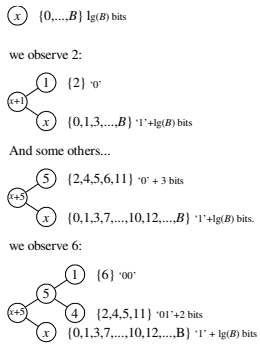

Fig. 2. Creation of new sets. Assigned codes are to the right of the sets. The value of *x* depends of the type of prior one wants to use for the probability distribution (for e.g., *x* = 1 is a Laplacian prior).

An important question is, of course, when do we know it's time to rebalance the tree? If we look at fig. 3., we see that after the insertion of the new set {6}, an internal node has count 5 while it sibling has only a count of 1. We would like this internal node to be nearer to the root of at least one step since it's intuitively clear that this node (and all its children) is far more frequent than its sibling. The rule to 'shift up' a node is:

If (the node's weigth > its sibling's weight +1) **and**
(the node's weight > its uncle's weight) **then** shift up that node.

The shifting up itself is, except for the condition, exactly as in a standard, off-the-shelf, AVL tree. Fig. 4. shows how this operation works. If for a particular node the rule is verified, a shifting up occurs. There might be more than one shift up. We repeat until the rule ceases to apply or until we reach the root. And, of course, we update weights up to the root. The algorithm, that we call M, as in 'migration' is listed in pseudocode at algorithm 1. The shift up procedure is given as algorithm 2.

1.2. Asymptotic performance of Algorithm M

The code generated by algorithm M has an entropy within [H(*S*), H(*S*)+2) when given a sufficiently long sequence of symbols. Let us explain how one derives this result. First, all codes have two parts: a prefix and a suffix. The prefix identifies in which subset $K \subseteq S$ the symbol *s* is. The suffix identifies the symbol within the subset. The length of the code is, for each symbol *s* in a subset $K \subseteq S$, given by $l(s) = l(K) + l(s|K) = -\lg \mathrm{p}(K) - \lg \mathrm{p}(s|K) = -\lg \mathrm{p}(s)$. We will assign an integer number of bits to both prefix and suffix. In the algorithm, the ideal code it is approximated by $\tilde{l}(s) = \tilde{l}(K) + \tilde{l}(s|K) = \mathrm{Prefix}(K) + \lg|K|$ where Prefix(*K*) is the length of the prefix for the set *K*. This is a very good approximation of *l*(*s*) when all the symbols in the subset *K* have roughly the same probability *and* Prefix(*K*) is near the optimal prefix length.

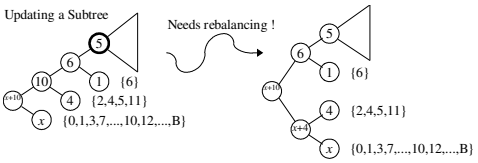

Fig. 3. Rebalancing the tree.

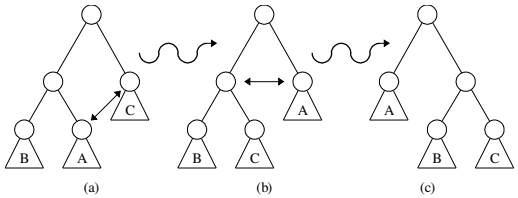

Fig. 4. Shifting up subtree A. (a) Exchanging subtree with uncle subtree, (b) rotation and (c) final state.

```
Procedure Update_M(a : symbol)
 {
  q,p :pointers to leaves ;

  p = find(a) ;  // Leaf/set containing a
  q = find(p's frequency + 1) ; // where to
migrate ?

  if (q != ⊥ ) // somewhere to migrate? (⊥ is NIL)
   {
    remove a from p's set ;
    p's weight = p's weight – p's frequency
    Add a to q's set ;
    q's weight = q's weight + p's frequency

    ShiftUp(q);
    If (p = ∅)
         remove p from the tree ;
    else ShiftUp(p's sibling) ;

   }
  else
   {
    create a new node t ;
    t's left child is p ;
    t's right child is a new node n ;
    n's set = {a} ;
    n's weight = p's frequency + 1 ;
    n's frequency = p's frequency + 1 ;
    replace the old p in the tree by t ;

    remove a from p's set ;
    p's weight = p's weight – p's frequency ;

    If (p = ∅)
         remove p from the tree ;
    else ShiftUp(p's sibling) ;

    ShiftUp(t) ;
   }
 }
```

Algorithm 1. Algorithm M.

```
Procedure ShiftUp(t : pointer to leaf)
 {
  while (t is not the root)
   {
    t's weight = t's right child's weight +
                 t's left  child's weight ;
    if ( (t's weight > t's sibling's weigth+1) &&
         (t's weight > t's uncle's weight))
     then {
           q = parent of parent of t ;
           exchange t with t's uncle ;
           exchange q's right and left child ;
           Update t's ancient parent's weight ;
          }
    t = t's parent ;
   }
 }
```

Algorithm 2. ShiftUp algorithm.

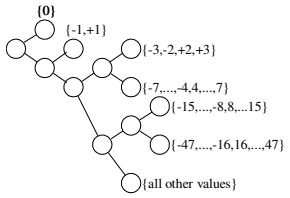

Fig. 5. A possible different initial configuration for the tree.

First, we consider the suffixes. All symbols in a set K are equiprobable (given set K) by definition[2]. One observes that in that case -lg p($s \mid K$) is exactly lg $\mid K \mid$. When we assign a natural code to every symbol s in K this code will be of length of at most $\lceil \lg \mid K \mid \rceil$, which is never more than one bit too long compared to the optimal code. A worst case for a suffix is when the number of symbol in this set is 2^n+1, for some n, and in that case we will waste almost 1 bit. The best case will be when the size of the set is exactly 2^m, for some m, for which we will use exactly m bits (and not waste any).

The prefixes are Shannon-Fano codes for the sets K_i — remember that the leaves of the tree are sets and not individual symbols and that all symbols in a set have the same probability. Since the shift-up algorithm tries to produce equally weighted subtrees it is essentially a classical Shannon-Fano algorithm from the *bottom up* instead of *top-down*. The Shannon-Fano algorithm isn't optimal (in the same way Huffman's algorithm is) but it produces codes within [H(S), H(S) + 1) of the entropy of the set $S = \{K_1,K_2,...,K_m\}$ [8].

Combining the two results leads to the bounds on entropy of [H(S), H(S) + 2). That makes Algorithm M very interesting especially when we consider that we don't create as many nodes as algorithm Vitter's algorithm Λ.

[2] However, one may want to put in the set K_i all symbols of *approximately* the same probability. In that case, the bounds change because the 'rounding' of the codes will be different. The code will not degrade if the optimal code of the least probable symbol of a set is less than one bit shorter than the optimal code of the most probable. That is, one must satisfy the constraint

$$-\lg \min\{p(s_i|K)\} + \lg \max\{p(s_i|K)\} \le 1 .$$

In section 2, Results, we compare the number of nodes created.

1.3. Computational complexity of Algorithm M

Ideally, we would like to have O(1) operations on a set — adding and removing elements in constant time — but it will be quite good if it is in O(lg k) for a set of k elements. While Anti-AVL trees are used in algorithm M for the codes themselves, AVL trees might not be a good choice for the internal representation of the sets. AVL trees do eat a lot of memory — in addition of being a 'big' O(lg n) in time complexity — and the total number of nodes in the trees (prefix and sets) would then be the same as in any other Huffman-like tree algorithms, that is, $2|S|-1$. One alternative representation is a skip list of intervals and integers. We use interval nodes when there are enough successive integers (like {1,2,3,4,5,6}) and single integers when it is not (like {3,5,7,11,13}) — at the expense of a bit to distinguish between the two. The representation we used in our experiments had a quite efficient memory usage. For e.g., when 3 was added to the set {1,2,4,5} (two intervals, {1,...,2} and {4,...,5}) the resulting set was represented by only one interval: {1,...,5} which needs only two `int` rather than five. Savings are greater when the width of the interval grows. Needless to say that great care must be exercised when one chooses the representation of the sets. The use of a skip list or and AVL tree as a catalog of intervals give an O(lg n) average case for finding an interval in a set. The actual operation of adding or removing of an item within a set should be in O(1), thus giving a total complexity of O(lg k) in worst case for set manipulation, where here k is the cardinality of the set.

The average update complexity is also in logarithmic time. More precisely, we can show that it is $O(\lg p(s) + f(s) + g(s) + h(s,K_1,K_2))$, where $p(s)$ is the probability of the symbol s, and $f(s)$ is the cost of finding in which set s is (that is, K_1), $g(s)$ the cost of finding in which set s will land (K_2) and finaly $h(s,K_1,K_2)$ is the cost of moving s from set K_1 to set K_2. The $f(\cdot)$ and $g(\cdot)$ functions are in O(lg n_s), where n_s is the number of frequency classes (sets). We already described h(a,A,B), which is a combination of a removal from a set and an insertion in another set, which are both O(lg $|K|$), therefore h is also of logarithmic complexity. Now, for the lg $p(s)$ part, we know that the shift up procedure updates the destination set (which is at depth of lg $p(s)'$, where $p(s)'$ is the new probabiliy of s) and its source set's sibling. Since $p(s) \cong p(s)'$, both shift up are O(lg $p(s)$).

1.4. Initialisation

One way to initialize the algorithm is with a single node whose set contains all possible symbols (of frequency 1[3], of weight B). But one may also choose to start with several subsets, each having different priors. If we return to the example in fig. 1, we could build with *a priori* knowledge, as in fig. 5, an initial solution already close to what we want, with correct initial sets and weights. That would give a better compression right from the start, since we would need much less adaptation. That does not imply, however, that we always have to transmit the code book or the tree prior to decompression. In various schemes the initial configuration of the tree may be standardized and would therefore be implicit. In our hypothetical example of fig. 1, we can decide that the tree is always the same when encoding or decoding starts.

1.5. Fast Decoding

Yet another advantage of algorithm M is that it may provide for a faster decoding than the other algorithms. When we decode the compressed data with the other algorithms, we read the bits one by one as we go down the tree until a leaf is reached. In algorithm M, however, we don't have to read *all* the bits one by one since once we reach a leaf, we know how many bits there are still to be read and that allows us to make a single operation to get a bit string which is readily converted into an integer, which in turn is converted into the correct symbol. On most general processors, the same time is needed to read a single bit as to extract a bit string, since it is generally the same operations but with different mask values. So the speedup is proportional to the number of bits read simultaneously. This technique could lead to very efficient decoders when the number of frequency classes is exponentially smaller than the alphabet size.

[3] That is, we assume that the distribution is uniform and a Laplacian prior.

2. Experimental Results

We finally present our results and compare algorithm M against Huffman's algorithm. Table 1. summarizes the results on the Calgary corpus. The 'Huffman' column of Table 1 is the number of bits per symbol obtained when we use a two pass static Huffman code (that is, that we read the entire file to gather information on the p(s) and then generate the code book). In one case, we omitted the cost of transmitting the dictionary, as it is often done. In the other column, namely Huffman*, we took into account the cost of sending the code book, which is always assumed to be 1K, since with Huffman and M we assumed *a priori* that each file in this set had 256 different symbols. For algorithm Λ, we find a column named 'algorithm Λ*'. This is Paul Howard's version of algorithm Λ (see [7,12] and Acknowledgments). In this program the actual encoding is helped by an arithmetic coder. In the columns of algorithm M, we can see also the number of bits per symbols obtained and we see that they are in general very close to the optimal Huffman codes, and in most cases *smaller* than Huffman*. In the 'nodes' column we see how many nodes were in the tree when compression was completed. The 'Migs%' column counts, in percent, how many updates were solved using *only* set migration. The 'ShiftUps' column counts the number of time a node was shifted up.

For all the experiments, the initial configuration shown in fig. 6. was used since most of the Calgary Corpus' files are text files of some kind, to the exception of 'Geo', 'Obj1', 'Obj2' and 'Pic' that are binary files. Here again we stress the importance the initial configuration has for our algorithm. If we examine Table 1, we see that most of the updates don't need either set migration nor rebalancing. These updates consist in adding and removing nodes from the tree. This situation arises when simple migration fails, that is, when the destination set does not exist or the source set has only one element (therefore needs to be destroyed). We also see that shifting up in rebalancing is a relatively rare event, which good even if it is not a costly operation.

With Huffman's algorithm and 256 symbols, one always gets 511 nodes (since it is a full binary tree, we have $2|S|-1$ nodes, counting leaves, for a set S) while this number varies with algorithm M depending on how many sets are created during compression/decompression. One can see that in the average, significantly less nodes are created. Instead of 511 nodes, an average of 192 is created by the files of the Calgary Corpus. This is about only 37.6% of the number of nodes created by the other algorithms. When we use 16 bits per symbol (which were obtained by the concatenation of two adjacent bytes), we find that algorithm M creates an average of 360.6 nodes instead of an average of 3856.6 with the other algorithms. This is about 9.3% : ten times less. Of course, there exist degenerate cases where the same number of nodes will be created, but never more.

In Table 2, we present the results, again, but with 16 bits symbols. We supposed that each file of the Calgary Corpus was composed of 16 bits symbols — words instead of bytes. We compare Vitter's algorithm Λ, static Huffman (both with and without the cost of transmission of the dictionary) and algorithm M.

One may notice that Vitter's algorithm Λ is almost always the best when we consider 8 bits per symbols (it wins over the two other algorithms 13 times out of 18) but that the situation is reversed in favor of algorithm M when we consider 16 bits per symbols. In the latter situation, algorithm M wins 13 times out of 18. The average number of bits outputed by each algorithm is also a good indicator. When the symbols have 8 bits, we observe that Huffman* generates codes that have an average of 5.12 bits/symbols, Algorithm Λ 4.75 bits/symbols and algorithm M 5.07 bits/symbols, which leaves algorithm Λ as a clear winner. However, again, the situation is reversed when we consider 16 bits symbols. Huffman* has an average of 9.17 bits/symbols, algorithm Λ 8.97 bits/symbols and algorithm M generates codes of average length of 8.67 bits/symbols. Here again, algorithm M wins over the two others.

3. Improvements: Algorithm M$^+$

While algorithm M (and others) uses all past history to estimate p(s), one might want to use only part of history, a *window* over the source because of non-stationarity in the data sequence. Let us now present a modified algorithm M, called algorithm M$^+$ that handles windows over the source. The window is handled by demoting a symbol as it leaves the window. As a symbol leaves the window, we decide that if had until then a frequency m, it is now has a frequency m-1. The difference between the promotion, or *positive update algorithm* and the demotion, or

negative update algorithm is minimal. Both operate the migration of a symbol from a set to another. The difference is that we move a symbol from a set of frequency m to a set of frequency m-1 rather than from m to m+1. If the set of frequency m-1 exists, the rest of the update is as in algorithm M: we migrate the symbol from its original set to the set of frequency m-1 and we shift up the involved sets (the source's sibling and the destination). If the destination does not exist, we create it and migrate the symbol to this new set. Algorithm 3 shows the negative update procedure.

The complete algorithm that uses both positive and negative updates is called M+. We will need a windowed buffer, of a priori fixed length[4], over the source. For compression, the algorithm proceeds as follows: as a symbol `a` enters the window, we do a positive update (or `Update_M(a)`) and emit the code for `a`. When a symbol `b` leaves the window, we do a negative update (`Negative_Update_M(b)`). On the decompression side, we will do exactly the same. As we decode a symbol `a`, we do a positive update and insert it into the window. This way, the decoder's window is synchronized with the encoder's window. When a symbol `b` leaves the window, we do a negative update (`Negative_Update_M(b)`). Algorithm 4 shows the main loop of the compression program, while algorithm 5 shows the decompression program.

3.1. Results and Complexity Analysis of M+

We present here the new results. The various window sizes tried in the experiments were 8, 16, 32, 64, 128, 256, 512, and 1024. The results are presented in table 3. For some files, a window size smaller than file size gave a better result, while for some other, only a window as large as the file itself gave the best results. Intuitively, one can guess that files that are compressed better with a small window do have non-stationarities and those which do not compress better either are stationary or do have non-stationarities but that they are not captured: they go unnoticed or unexploited because either way the algorithm can't adapt fast enough to take them into account.

Algorithm M+ is about twice as slow as algorithm M. While it is still O(lg n_s) per symbol, we have the supplementary operation of demotion, which is O($h(s,K_1,K_2)$ + lg n_s), where $h(s,K_1,K_2)$ is the complexity of moving a symbol from set K_1 to set K_2, and n_s is the number of sets. Since migration processes are symmetric (promotion and demotion are identical : one migration and two shift up) algorithm M+ is exactly twice as costly as algorithm M. Since algorithm M+ is O($h(s,K_1,K_2)$ + lg n_s), we will remind the reader that she will want the complexity of $h(s,K_1,K_2)$ should be as low as possible, possibly in O(max(lg $|K_1|$, lg $|K_2|$)), which is reasonably fast[5].

4. Conclusion

The prototype program for Algorithm M was written in C++ and runs on a 120 MHz 486 PC under Windows NT and a number of other machines, like a Solaris SPARC 20, an UltraSPARC and a SGI Challenge. On the PC version, the program can encode about 70,000 symbols per second when leaves are individual symbols and about 15,000 symbols/second when they are very sparse sets (worst case: very different symbols in each frequency class).

We see that the enhanced adaptivity of algorithm M+ can give better results than simple convergence under the hypothesis of stationarity. The complexity of Algorithm M+ is the same as Algorithm M's, that is, O($h(s,K_1,K_2)$ + lg n_s) = O(max($h(s,K_1,K_2)$, lg n_s)) while the so-called hidden constant only grows by a factor of 2. In our experiments, M+ can encode or decode about 10000 to 50000 symbols per second, depending on many factors such as compiler, processor and operating system. The reader may keep in mind that this prototype program wasn't written with any hacker-style optimizations. We kept the source as clean as possible for the reader. We feel that with profiling and fine-tuning the program could run at least twice as fast, but that objective goes far off a simple feasibility test.

In conclusion, Algorithms M and M+ are a good way to perform adaptive Huffman coding, especially when there is a very large number of different symbols we don't have that much memory. We also showed that for Algorithm M, the compression performance is very close to the optimal static Huffman code and that the bounds

[4] We do not address the problem of finding the optimal window length in algorithm M+.

[5] The current implementation of our sets is in worst case O(|s|), but is most of the time in O(lg |s|).

[H(S), H(S)+2) guarantee us that the code remains close enough to optimality. Furthermore, these algorithms are free to use since released to the public domain [1,2].

						Algo M		
		Bits/symb						
File	Length	Huffman*	Huffman	Algo Λ*	algo M	Nodes	Migs (%)	ShiftUps
Bib	111 261	5.30	5.23	5.24	5.33	161	3.92	2374
Book1	768 771	4.57	4.56		4.61	153	0.62	2494
Book2	610 856	4.83	4.82		4.91	191	0.94	3570
Geo	102 400	5.75	5.67		5.82	369	25.6	5068
News	377 109	5.25	5.23	5.23	5.31	197	2.00	4651
Obj1	21 504	6.35	5.97		6.19	225	35.88	1091
Obj2	246 814	6.32	6.29		6.40	449	12.04	11802
Paper1	53 161	5.17	5.02	5.03	5.12	171	5.86	1408
Paper2	82 199	4.73	4.63	4.65	4.73	155	3.12	1152
Paper3	46 526	4.87	4.69	4.71	4.86	147	5.22	978
Paper4	13 286	5.35	4.73	4.80	4.98	109	11.97	523
Paper5	11 954	5.65	4.97	5.05	5.20	131	16.48	665
Paper6	38 105	5.25	5.04	5.06	5.15	161	8.15	1357
Pic	513 216	1.68	1.66	1.66	1.68	169	0.76	1957
ProgC	39 611	5.44	5.23	5.25	5.38	177	11.31	1984
ProgL	71 646	4.91	4.80	4.81	4.92	147	4.32	1442
ProgP	49 379	5.07	4.90	4.91	5.00	159	7.13	1902
Trans	93 695	5.66	5.57	5.43	5.69	189	6.06	2926
	average	5.12	4.95	4.75	5.07			

Table 1. Performance of Algorithm M. The Huffman* column represents the average code length if the cost of transmission of the code book is included, while the Huffman column only take into account the codes themselves. Algorithm Λ* is Vitter's algorithm plus arithmetic coding. Algorithm M does not transmit the code book, the bits/s are really the averages of code length for Algorithm M. The number of nodes with the static Huffman (Huffman, Huffman*) is always 511 — 256 symbols were assumed for each files. Grey entries correspond to unavailable data.

			Huffman				Algo M	
				Bits/symb				
File	Length	Symbols	Nodes	Huffman*	Huffman	algo Λ*	algo M	Nodes
Bib	111 261	1323	2645	8.96	8.58	10.48	8.98	423
Book1	768 771	1633	3265	8.21	8.14		8.35	873
Book2	610 856	2739	5477	8.70	8.56		8.81	835
Geo	102 400	2042	4083	9.86	9.22		9.74	281
News	377 109	3686	7371	9.61	9.30	9.62	9.66	713
Obj1	21 504	3064	6127	13.72	9.17		9.65	97
Obj2	246 814	6170	12339	9.72	8.93		9.40	483
Paper1	53 161	1353	2705	9.45	8.64	9.47	9.13	281
Paper2	82 199	1121	2241	8.57	8.13	8.57	8.48	369
Paper3	46 526	1011	2021	8.93	8.23	8.94	8.68	281
Paper4	13 286	705	1409	9.83	8.13	9.84	8.81	131
Paper5	11 954	812	1623	10.60	8.43	10.60	9.13	113
Paper6	38 105	1218	2435	9.63	8.61	9.66	9.14	231
Pic	513 216	2321	4641	2.53	2.39	3.74	2.47	273
ProgC	39 611	1443	2885	9.97	8.80	9.97	9.37	221
ProgL	71 646	1032	2063	8.46	8.00	8.48	8.37	315
ProgP	49 379	1254	2507	8.86	8.05	8.89	8.56	223
Trans	93 695	1791	3581	9.52	8.91	9.34	9.39	347
			average	9.17	8.23	8.97	8.67	

Table 2. Comparison of algorithms with a larger number of distinct symbols. Grey entries correspond to unavailable data.

```
Procedure Negative_Update_M(a : symbol)
 {
  q,p :pointers to leaves ;

  p = find(a) ;  // Leaf/set containing a
  q = find(p's frequency - 1) ; // were to migrate ?

  if (q != ⊥ ) // somewhere to migrate ?
   {
    remove a from p's set ;
    p's weight = p's weight - p's frequency
    Add a to q's set ;
    q's weight = q's weight + p's frequency

    ShiftUp(q);
    If (p = Ø)
         remove p from the tree ;
    else ShiftUp(p's sibling) ;
   }
  else
   {
    create a new node t ;
    t's left child is p ;
    t's right child is a new node n ;
    n's set = {a} ;
    n's weight = p's frequency - 1 ;
    n's frequency = p's frequency - 1 ;
    replace the old p in the tree by t ;

    remove a from p's set ; // t's left child
    p's weight = p's weight - p's frequency ;

    If (p = Ø)
         remove p from the tree ;
    else ShiftUp(p's sibling) ;

    ShiftUp(t) ;
   }
 }
```

Algorithm 3. Negative update algorithm.

```
while (not eof(f))
 {
  read c from f;
  put c in window/buffer ;

  w = code for c ;

  Emit w into f2 ; // output file

  Update_M(c) ;

  get d from the other end of window/buffer ;
  Negative_Update_M(d) ;
 }
```

Algorithm 4. Encoding loop.

```
for i=1 to w
 {
  c= Decode from f;
  Update_M(c) ;

  put c in window/buffer ;

  output c into destination file ;

  get d from the other end of window/buffer ;
  Negative_update_M(d) ;
}
```

Algorithm 5. Decoding loop.

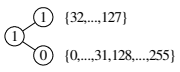

Fig. 6. Initial configurations for the tests.

		M^+		window size							
File	Size	8	16	32	64	128	256	512	1024	M	Huff*
BIB	111261	5.87	5.50	5.49	5.42	5.37	5.35	5.31	5.29	5.33	5.30
BOOK1	768771	4.86	4.73	4.66	4.66	4.64	4.63	4.63	4.62	4.61	4.57
BOOK2	610856	5.01	4.90	4.91	4.90	4.90	4.88	4.87	4.93	4.91	4.83
GEO	102400	7.55	7.60	7.52	7.07	6.83	6.71	6.24	6.14	5.82	5.75
NEWS	377109	5.38	5.33	5.40	5.31	5.30	5.31	5.33	5.30	5.31	5.25
OBJ1	21504	7.92	7.39	7.02	6.92	6.70	6.58	6.51	6.46	6.19	6.35
OBJ2	246814	7.79	7.09	6.71	6.52	6.47	6.48	6.48	6.43	6.40	6.32
PAPER1	53161	5.61	5.41	5.29	5.19	5.16	5.13	5.15	5.21	5.12	5.17
PAPER2	82199	5.11	4.86	4.77	4.76	4.71	4.73	4.83	4.74	4.73	4.73
PAPER3	46526	5.18	4.93	4.87	4.81	4.81	4.82	4.79	4.79	4.86	4.87
PAPER4	13286	5.54	5.18	5.15	5.03	4.93	5.00	4.97	4.96	4.98	5.35
PAPER5	11954	5.93	5.47	5.35	5.26	5.24	5.25	5.20	5.25	5.20	5.65
PAPER6	38105	5.81	5.38	5.42	5.25	5.30	5.22	5.26	5.19	5.15	5.25
PIC	513216	1.78	1.72	1.71	1.70	1.70	1.68	1.68	1.68	1.68	1.68
PROGC	39611	5.87	5.69	5.63	5.45	5.41	5.47	5.37	5.40	5.38	5.44
PROGL	71646	5.14	5.12	4.95	4.92	4.95	4.98	4.99	4.89	4.92	4.91
PROGP	49379	5.61	5.28	5.12	5.13	5.03	5.02	5.03	5.02	5.00	5.07
TRANS	93695	6.29	5.86	5.75	5.75	5.69	5.71	5.70	5.64	5.69	5.66
average											

Table 3. Results of Algorithm M+ against Huffman.

Acknowledgements

Special thanks to J.S. Vitter (`jsv@cs.duke.edu`) who graciously gave me a working implementation of his algorithm in Pascal. We would also like to thank Paul G. Howard of AT&T Research (`pgh@research.att.com`) for his C implementation of Algorithm Λ, that we called Λ*, since in his version Vitter's algorithm is followed by arithmetic coding (J.S. Vitter was Paul G. Howard's Ph.D. advisor).

References

[1] Steven Pigeon, Yoshua Bengio — A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols — Université de Montréal, Rapport technique #1081

[2] Steven Pigeon, Yoshua Bengio — A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols Revisited — Université de Montréal, Rapport technique #1095

[3] D.A. Huffman — A method for the construction of minimum redundancy codes — in Proceedings of the I.R.E., v40 (1951) p 1098-1101

[4] R.G. Gallager — Variation on a theme by Huffman — IEEE. Trans. on Information Theory, IT-24, v6 (nov 1978) p 668-674

[5] N. Faller — An Adaptive System for Data Compression — Records of the 7th Asilomar conference on Circuits, Systems & Computers, 1973, p. 393-397

[6] D.E. Knuth — Dynamic Huffman Coding — Journal of Algorithms, v6, 1983 p. 163-180

[7] J.S. Vitter — Design and analysis of Dynamic Huffman Codes — Journal of the ACM, v34 #4 (oct. 1987) p. 823-843

[8] T.C. Bell, J.G. Cleary, I.H. Witten — Text Compression — Prentice Hall 1990 (QA76.9 T48 B45)

[9] G. Seroussi, M.J. Weinberger — On Adaptive Strategies for an Extended Family of Golomb-type Codes — in DCC 97, Snowbird, IEEE Computer Society Press.

[10] A. Moffat, J. Katajainen — In place Calculation of Minimum Redundancy Codes — in Proceeding of the Workshop on Algorithms and Data Structures, Kingston University, 1995, LNCS 995 Springer Verlag.

[11] Alistair Moffat, Andrew Turpin — On the Implementation of Minimum-Redundancy Prefix Codes — (extended abstract) in DCC 96, p. 171-179

[12] J.S. Vitter — Dynamic Huffman Coding — Manuscript available on the author's web page, with the source in PASCAL

[13] Steven Pigeon — A Fast Image Compression Method Based on the Fast Hartley Transform — AT&T Research, Speech and Image Processing Lab 6 Technical Report (number ???)

[14] Douglas W. Jones — Application of Splay Trees to Data Compression — CACM, v31 #8 (August 1988) p. 998-1007

[15] Peter Deutsch, Jean-Loup Gailly — ZLIB Compressed Data Format Specification version 3.3 — RFC memo #1950

[16] Peter Deutsch — DEFLATE Compressed Data Format Specification version 1.3 — RFC memo #1951

[17] Khalid Sayhood — Introduction to Data Compression — Morgan Kaufmann 1996. TK5102.92.S39 1996).