

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

Steven Pigeon

Yoshua Bengio

{pigeon,bengioy}@iro.umontreal.ca

Département d'Informatique et de Recherche opérationnelle (DIRO)

Université de Montréal

6/19/97

Note

The algorithm described in this paper as “algorithm M” is hereby released to the public domain by its authors. Any one can use this algorithm free of royalties provided that any product, commercial or not, includes a notice mentioning the use of this algorithm, the name of the algorithm and the name of the authors.

Abstract

The problem of computing the minimum redundancy codes as we observe symbols one by one has received a lot of attention. However, existing algorithm 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 symbol, 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 number of observed symbols.

1. Introduction and motivation.

Of all compression algorithms, it is certainly Huffman's algorithm [1] for generating minimum redundancy codes that is the most popular. 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 $-\log(p(s))$, where $p(s)$ is an estimate of the probability of symbol s , as would dictate information theory. For the set S , the optimal average code length is given by

$H(S) = -\sum_{s \in S} p(s) \log(p(s))$. Huffman's algorithm, which we won't cover in great detail here, generates a set of codes C (also called code book) such that each s in S has its code c in C . These codes are *prefix-free* codes and *instantaneously decodable*, that is, no code is the prefix of some other code and one can know when the last bit of the code is read and immediately recognize the code.

Huffman's algorithm generates a set of codes for which the average code length is bounded by the interval $[H(S), H(S) + p + 0.086]$, where p is the largest probability of all symbols s in S [6, 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 C after having obtained an estimate for $p(s)$ 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 $p(s)$. One must not forget that we are using $p(s)$ instead of a more appropriate $p_t(s)$, 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 $p_t(s)$ 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 $p(s)$ 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 $p(s)$ out of a small sample of the data we expose ourselves to the possibility that the estimate of $p(s)$ out of a small sample is a bad approximation of the real $p(s)$.

The memory space used by Huffman’s algorithm (whether the implementation is memory-efficient or not) is essentially $O(n)$, 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 [6,7], 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, in fact, 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 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 technical report, 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 first review Huffman’s algorithm and the related adaptive algorithms from Faller, Gallager, Knuth and Vitter. We then present our algorithm which consists in three conceptual parts: set representation, set migration and tree rebalancing. We finally discuss about various initialization schemes and compare our results against the Calgary Corpus and static Huffman coding (the ordinary Huffman algorithm).

2. Huffman's Algorithm for Minimum-Redundancy Codes.

Huffman's algorithm is already described in detail in many textbooks like the excellent [6]. Huffman's procedure, quite simple, to construct the code book for a set S and a probability function $p(s)$ is shown in Algorithm 1 and fig. 1. The objective is to construct a tree such that the path from a leaf that contains the symbols s to the root has a length as close as possible to $-\log(p(s))$. To each leaf is associated a weight, which is basically the number of times the symbol s has been seen in the data (or $p(s)$, both can be used interchangeably). Huffman algorithm's strategy is to build a tree with weights that are as balanced as possible.

What is unusual in Huffman's algorithm is that instead of building this tree from the root down to the leaf (which would then be Shannon-Fano's algorithm! [6]) it is built from the *leaves up to the root*. The first step is to build a list which contains only leaves. Each leaf has an associated symbol s and a weight, $p(s)$. We pick, out of that list, the two element with the smallest weights. We create a new node such that its children are the two picks, and that its weight is the sum of its children's weight. We put back the new node in the list. We repeat this procedure until only one node is left, which will become the root of the tree. Codes are then assigned easily. Each time we go left in the tree, we assign a '1', and when we go right, we assign a '0', like in fig. 1. The code for a leaf is the path one must follow to reach it from the root. There are other algorithms that first compute the length of the codes before actually assigning bits to the codes. Such algorithms [8,9] are quite fast and can be done *in situ* without the expense of the extra memory needed to maintain the tree structure. One can show that the average code length is within $[H(S), H(S)+p+0.086)$ of the optimal average code length, where p is the probability of the most probable symbol in S .

The Shannon-Fano algorithm proceeds, as we said, at the opposite of Huffman's. In the Shannon-Fano algorithm, we first construct a list L sorted in reverse order, from the most to the least probable symbol. All

symbols, at that time, have an empty code. We then cut the list in half, but in such a way that the top half has the same total weight as the bottom half (as best as we can, there's no warranty that one can always cut the total weight of the list *exactly* in two halves!). To all the codes of the symbols of the top half of the list, we append '1', and to all the codes in the lower half we append '0'. We repeat this process recursively on the top half of the list and the bottom half, until we get a list with only one element, to which code nothing is appended. The Shannon-Fano algorithm is however not quite as good as Huffman's: its average code length is provably within $[H(S), H(S) + 1)$ of the optimal entropy $H(S)$.

```

For each symbol s create a leaf of weight p(s)
Put all leaves in the list L.

While there is at least 2 elements in L
{
  Pick A and B out of L such that their
  weights are the smallests ;

  Create a new node p such that
  {
    p's weight = A's weight + B's weight
    Make A and B p's children ;
  }
  Put p in the list L ;
}

```

Algorithm 1. Huffman's Algorithm

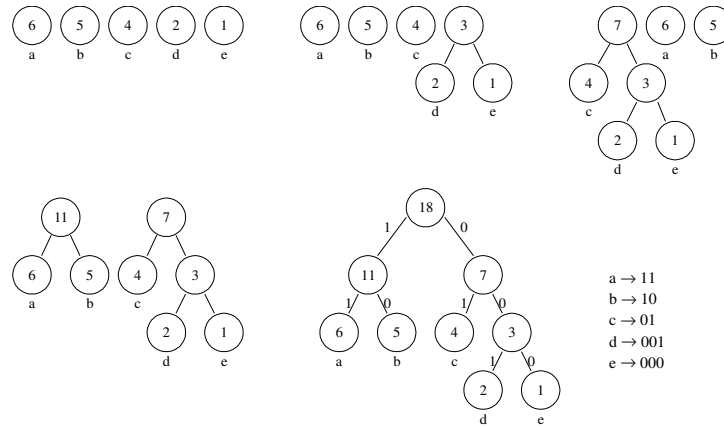


Fig. 1. Example of construction of a Huffman tree.

3. Adaptive Huffman algorithms: Algorithms FGK and Λ

Faller, Gallager and Knuth independently proposed an algorithm for creating adaptive Huffman codes [2,3,4] that we will designate as the 'FGK algorithm'. The FGK algorithm needs some secondary structures to manage the internal nodes of the tree. Not unlike AVL trees, the FGK algorithm seeks not to balance the tree by the depths of its nodes but by their weights. In that way, 'heavy' leaves (that is, frequent leaves) are near the root and

infrequent leaves very far. The algorithm is also making sure that leaves are as close to $-\log(p(s))$ nodes away from the root as possible. However, this algorithm can be quite bad in some cases as Vitter [5] shows. The algorithm can encode the data with as much as $(2H+1)t$ bits, where H is not $H(S)$ but the average code length obtained by the static Huffman algorithm, and t is the total number of symbols (the length of the data sequence). Its update procedure is shown in Algorithm 2. Nodes are numbered breadth-first from left to right, starting at the root. The FGK algorithm also uses a special ‘zero-node’ which is used to introduce the symbols that have not been seen so far in the sequence. An update is always made in $O(-\log p(s))$ operations.

Vitter, in [5], also proposes an algorithm for adaptive Huffman codes. This algorithm, algorithm A, to follow Vitter’s own denomination, offers a better bound on compression performance than the FGK algorithm. Vitter shows that his algorithm is guaranteed to encode a data sequence within $[H(S), H(S)+1+p+0.086]$ of the entropy for a sequence sufficiently long. In this technical report, we only present an overview of the algorithm, but the reader may consult [5] and [10] for details and a Pascal language implementation. It also updates in $O(-\log p(s))$ in any case, but the bound on compression is clearly better and thus justifies the complexification of the algorithm (which is not too great, either).

These two algorithms, however, do not provide any solutions to the case where the number of distinct symbols is really large. In each case, the number of nodes is still $2|S|+1$, which is prohibitively high (especially when one considers all the pointers, weights, and other side information needed to construct and maintain the tree) in the case that is of interest to us, that is, sets with potentially millions of symbols.

```

procedure Update_FGK(a)
{
  q = leave associated to a
  If ( q is a zero-node ) && ( k < n )
  {
    Replace q by a zero-node parent with
    two zero-node leaves (numbered in this
    order: right, left, parent) ;
    q = right child just created;
  }
  If (q is the sibling of a zero-node)
  {
    Exchange q with the greatest numbered leaf of
    the same weight ;
    Add 1 to q's weight ;
    q = parent of q;
  }
  While (q is not the root)
  {
    Exchange q with the greatest numbered leaf of
    the same weight ;
    If (q is the greatest numbered node of the same weight)
    Add 1 to q's weight;
    q = parent of q;
  }
}

```

Algorithm 2. FGK Algorithm.

```

procedure Update_Λ(a);
{
  LeafToInc = 0;
  q = leaf associated to a;
  If ( q is a zero-node ) and ( k < n )
  {
    Replace q by a new internal zero-node such that
    its two children are also zero-node and that
    the right child is a ;
    q = the new zero-node ;
    LeafToInc = right child of q ;
  }
  else
  {
    Exchange q with the node with the greatest number and
    of same depth ;
    If (q is the sibling of a zero-node)
    {
      LeafToInc = q ;
      q = parent of q ;
    }
  }
  While (q is not the root)
  {
    q = the node of the same depth of q but with
    the largest number ;

    ShiftAndInc ( q );
    If LeafToInc != 0
    {
      // here: check for the two special cases
      // q is or is not a zero-node
      // Same as the first 'If' in the procedure
      ShiftAndInc(LeafToInc) ;
    }
  }
}

```

Algorithm 3. (a) . Algorithm Λ.

```

procedure ShiftAndInc(p)
{
  wt = p's weight;
  b = smallest numbered node with a weight
  greater than p's weight.
  If ( ( p is a leaf ) &&
  ( b is in a block of weight wt ) )
  || ( ( p is an internal node ) &&
  ( b is a block of leaves of weight wt+1 ) )
  {
    Move p such that it is the first node or leave
    in the block b ;
    Add 1 to p's weight ;
    If ( p is a leaf )
      p = new parent of p ;
    else p = ancient parent of p ;
  }
}

```

Algorithm 3. (b) . Algorithm ShiftAndInc.

4. Proposed Algorithm: Algorithm M.

We will now present an algorithm that is well suited for large sets of symbols. They naturally arises in a variety of contexts, such as in the ‘deflate’ compression algorithm and in JPEG. 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. 2. [11], is used to represent the possible values of the coefficients. There are however fewer values possible in JPEG, but still a few thousands. There is plenty of other situations where a large number of symbols is needed.

The algorithm we now propose uses 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 another 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 times¹. If the set of symbols seen m times is now empty, we destroy it. Since rebalancing is only needed when a new set (leaf) is created or deleted, we often avoid it as more and more sets are created, it becomes more and more likely that the right set already exists. Rebalancing, when needed, will be performed in at most $O(-\log p(s))$ operations. Let us now present in detail this new algorithm.

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 node x , such that its children are the node x and the node z .

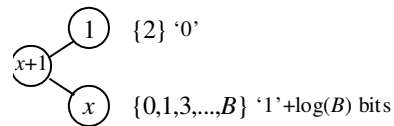
Let us say that we have the symbols represent the set of the integers from 0 to B . The minimal initial configuration of the tree is a single leaf (which is also the root) containing the set $\{0, \dots, B\}$, to which a zero frequency is assigned. It can be zero since we don't work directly with the 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, 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 $\lceil \log(|\{0, 1, 3, \dots, B\}|) \rceil$ 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. 3. 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.

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.

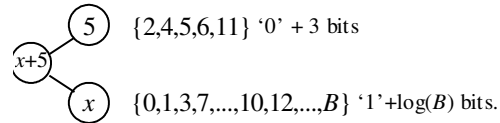
Fig. 2. An example of code for a large set and a hypothetical distribution.

(x) $\{0, \dots, B\}$ $\log(B)$ bits

we observe 2:



And some others...



we observe 6:

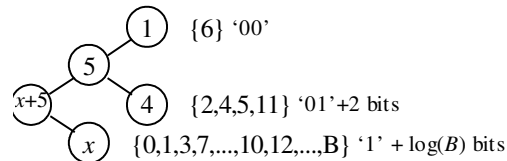


Fig. 3. Creation of new sets. Assigned codes are to the right of the sets. The value of x depends of the type of prior one wants to use for the probability distribution (for e.g., $x = 1$ is a Laplacian prior).

The big question is, of course, when do we know it's time to rebalance the tree? If we look at fig. 4., we see that after the insertion of the new set $\{7\}$, an internal node has count 5 while its 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 weight $>$ its sibling's weight +1) **and**
 (the node's weight $>$ its uncle's weight) **then** shift up that node.

The shifting up itself is exactly as in a standard, off-the-shelf, AVL tree. Fig. 5. shows how the shifting up of a tree 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 4.

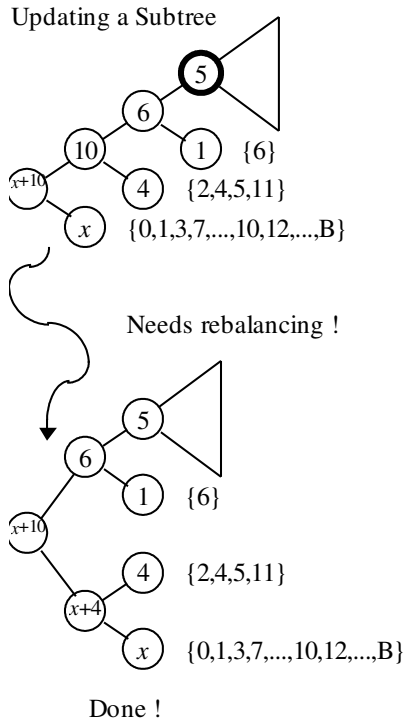


Fig. 4. Rebalancing the tree.

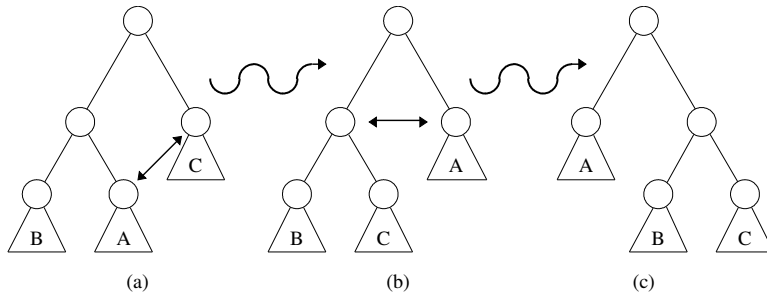


Fig. 5. Shifting up subtree A. (a) Exchanging subtree with uncle subtree, (b) rotation and (c) final state.

```

Procedure Update_M(a)
{
  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 = ∅)
      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 4. Algorithm M.

```

Procedure ShiftUp(t)
{
  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 weight+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 5. ShiftUp algorithm.

4.1. Asymptotic performance of Algorithm M.

The code generated by the algorithm M has an entropy within $[H(S), H(S)+2)$ given a sufficiently long sequence of symbols. Let us explain how one might derive 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. If we assign an integer number of bits to both prefix and suffix, the length of the code is, for each symbol s in a subset $K \subseteq S$, given by

$$l(s) = l(K) + l(s|K) = -\log p(K) - \log p(s|K) = -\log p(s)$$

that we approximate in the algorithm by

$$\tilde{l}(s) = \tilde{l}(K) + \tilde{l}(s|K) = \text{Prefix}(K) + \log|K|$$

where $\text{Prefix}(K)$ 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* $\text{Prefix}(K)$ is near the optimal prefix length.

First, we consider the suffixes. All symbols in a set K are equiprobable by definition². One observes that in that case $-\log p(s|K)$ is exactly $\log|K|$. When we assign a natural code to every symbol s in K this code will be of length $\lceil \log|K| \rceil$, 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 about 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.

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 isn't optimal (as Huffman's algorithm is) but it produces codes within $[H(S), H(S)+1)$ of the entropy of the set $S=\{K_1, K_2, \dots, K_m\}$ [6].

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

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

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 the two other adaptive algorithms. In section 5, Results, we compare the number of nodes created.

4.2. Initialisation

One way to initialize the algorithm is 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 take back the example in fig. 2, we could build with *a priori* knowledge, as in fig. 6, 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. 2, we can decide that the tree is always the same when encoding or decoding starts.

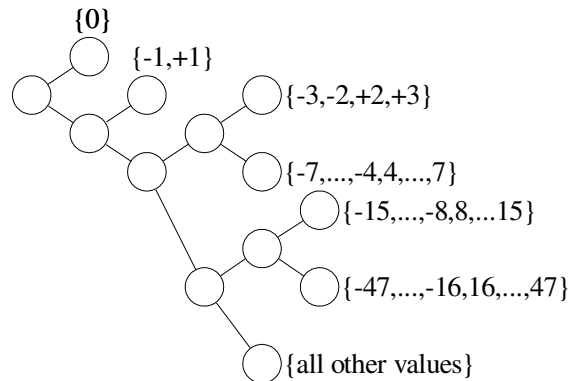


Fig. 6. A possible different initial configuration for the tree.

4.3. 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 or to extract a bit string, since it is generally the same

³ That is, we assume that the distribution is uniform — a Laplacian prior.

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.

4.4. Efficiency Considerations and Drawbacks

The major drawback of algorithm M is the representation used for the sets. When the sets are very sparse and irregular (and for which no really convenient representation exists, as it is the case with, for example, a bunch of prime numbers) they can use a lot of memory if one does not use a good representation. One must use a sophisticated representation for the sets if we want to manage memory efficiently *and* to do fast operations on the sets.

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 $O(\log n)$ for a set of n elements. While Anti-AVL trees are used in algorithm M for the codes themselves, AVL trees might be a good choice for the internal representation of the sets. However, AVL trees do eat a lot of memory — in addition of being a ‘big’ $O(\log n)$ — and the total number of nodes in the trees would be the same as in any other Huffman-like tree algorithms, 2|S|-1. One may use an AVL tree with intervals as leaves or even a bit-mapped boolean array if the number of different symbols is not too great (for 64k symbols, 8k would be needed for such a device).

One alternative representation is a sparse 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}). The representation we used 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.

5. 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 by symbol obtained when we use a two passes 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 had 256 different symbols. For algorithm Λ , we find a column named 'algorithm Λ^* '. This is Paul Howard's version of algorithm Λ (see [5] 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 by 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. 7. 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 a 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, many 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 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 when the same number of nodes will be created, but never more.

File	Length	Bits/symb				Algo M		
		Huffman*	Huffman	Algo A*	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
		μ: 5.12	μ: 4.95	μ: 4.75	μ: 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 code length of 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.

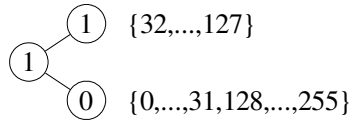


Fig. 7. Initial configurations for the tests.

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 long symbols. 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 outputted 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.

The prototype program was written in C++ and runs on a 120 Mhz 486 PC under Windows NT and a number of other machines, like a Solaris Spark 20, UltraSpark and 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.

In conclusion, Algorithm M is a good way to perform adaptive Huffman coding, especially when a very large number of different symbols is needed and we don't have that much memory. We also showed that the compression performance is very close to the optimal static Huffman code and that the bounds $[H(S), H(S)+2)$ guaranty us that the code remains close enough of optimality. Furthermore, this algorithm is free to use since released to the public domain.

File	Length	Symbols	Nodes	Huffman		Algo M		Nodes
				Huffman*	Huffman	algo Λ^*	algo M	
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
				μ : 9.17	μ : 8.23	μ : 8.97	μ : 8.67	

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

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] D.A. Huffman — A method for the construction of minimum redundancy codes — in. *Proceedings of the I.R.E.*, v40 (1951) p.1098-1101
- [2] R.G. Gallager — Variation on a theme by Huffman — *IEEE Trans. on Information Theory*, IT-24, v6 (nov 1978) p. 668-674
- [3] N. Faller — An adaptive system for data compression — records of the 7th Asilomar conference on Circuits, Systems & Computers, 1973, p. 393-397
- [4] D.E. Knuth — Dynamic Huffman Coding — *Journal of Algorithms*, v6 1983 p. 163-180
- [5] J.S. Vitter — Design and analysis of Dynamic Huffman Codes — *Journal of the ACM*, v34 # 4 (octobre 1987) p. 823-843
- [6] T.C. Bell, J.G. Cleary, I.H. Witten — Text compression — Prentice-Hall, 1990 (QA76.9 T48B45).
- [7] G. Seroussi, M.J. Weinberger — On Adaptive Strategies for an Extended Family of Golomb-type Codes — in DCC 1997, Snowbird, IEEE Computer Society Press.
- [8] Alistair Moffat and J. Katajainen — In place Calculation of minimum redundancy codes — in *Proceedings of the Workshop on Algorithms and Data Structures*, Kingston University, 1995, LNCS 995 Springer Verlag.
- [9] Alistair Moffat et Andrew Turpin, — On the implementation of minimum-redundancy prefix codes — (extended abstract), in DCC 1996, p. 171-179.
- [10] J.S. Vitter — Dynamic Huffman Coding — Manuscript from the Internet with a Pascal source.
- [11] 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 ???)
- [12] Douglas W. Jones — Application of Splay Trees to Data Compression — *CACM*, V31 #8 (August 1988) p. 998-1007