Practical Length-limited Coding for Large Alphabets*

ANDREW TURPIN AND ALISTAIR MOFFAT

Department of Computer Science, The University of Melbourne, Parkville 3052, Australia
Email: aht@cs.mu.oz.au

The use of minimum-cost coding for economical representation of a stream of symbols drawn from a defined source alphabet is widely known. However, for large-scale compression minimum-cost coding has the drawback that codewords generated may be longer than a machine word, limiting the usefulness of both software and hardware implementations on word-based architectures. The solution is to generate length-limited codes, and accept the consequent loss of compression effectiveness in order to preserve the simplicity and speed of the encoding and decoding software. Here we re-examine the package-merge algorithm for generating minimum-cost length-limited prefix-free codes and show that with a considered reorganization of the key steps it is possible for it to run quickly in significantly less memory than was required by previous implementations, while retaining asymptotic efficiency. As evidence of the practical usefulness of the improved method we describe experiments on an alphabet of over 1 million symbols, for which length-limited codes can be constructed in 11 Mb of memory and about 20 seconds of CPU time.

Received March 24 1995, revised August 3 1995

1. INTRODUCTION

Text compression is an important tool when large amounts of data are to be stored. Compression not only saves storage space, but also reduces disk traffic, sometimes to the extent that overall access times can be shorter with a compressed text than with an uncompressed text, even including the cost of the decompression (Zobel and Moffat, 1995).

It is generally accepted that a compression system consists of two activities: modelling and coding (Bell *et al.*, 1990). The model estimates, for each possible symbol that might appear next, a probability of occurrence. The coder is then responsible for representing the actual stream of symbols with respect to those estimated probabilities. There are many ways to model text and for an overview the reader is referred to Bell *et al.* (1990). Here we focus on the coding side of this partnership and ask what happens when the model contains a very large number—perhaps millions—of symbols. We do have in mind a particular model, discussed in Section 6, and our investigation has been motivated by very pragmatic requirements arising from the use of that model.

Probably the most famous of all coding techniques is Huffman's method (Huffman, 1952) for generating a minimum-cost code. Huffman's algorithm is described in textbooks covering both text compression and the more general area of algorithms and data structures. Moreover, the behaviour of the minimum-cost codes generated by Huffman's method is well understood and in most practical situations they given compression very close to the underlying model entropy (Manstetten, 1992), even though all of the codewords are integral length.

*This paper includes material presented in preliminary form at the 1995 Australasian Computer Science Conference.

There are two reasons why large alphabets pose problems for minimum-cost coding. The first is the memory space required by the algorithm. Typical descriptions (see e.g. Van Wyk, 1988, p. 238) make use of linked data structures with multiple fields per node. Since a code tree with n leaves (representing an alphabet of n symbols) contains $2n - 1$ nodes in total and each node requires as many as four words of memory, a straightforward implementation of Huffman's algorithm might require as much as $8n$ words of memory—32 Mb to generate a code for one million symbols.

By using an implicit tree structure rather than an explicit structure the space can be reduced. For example, Sedgewick (1990, p. 328) gives a construction employing arrays and an implicit heap structure that uses $5n$ words of memory. Further savings result if codeword lengths are calculated rather than actual codeword bit-descriptions and the memory requirement can be reduced to $2n$ words (Witten *et al.*, 1994). Although these improvements are in terms of constant factors only, the impact they have is important for large n, making large-scale coding practical. All of these algorithms require $O(n \log n)$ time to generate minimum-cost codes for an alphabet of n symbols and are asymptotically efficient. Note, however, that if the input list of symbol frequencies is already sorted then the running time of Huffman's algorithm can be improved to $O(n)$ (van Leeuwen, 1976) and n words of memory suffice to calculate the codeword lengths (Moffat and Katajainen, 1995).

The second problem with minimum-cost coding is more insidious; that of codeword overflow. If codewords can be stored in one machine word, input and output operations can be performed as a single 'mask and shift' sequence, giving rise to the high speed of practical coders (Moffat *et al.*, 1994). The speed of these coders is their principal advantage over arithmetic coding methods of the form described by Witten *et al.*, (1987) and modifications to cope with the possibility of multi-word codewords would severely erode this advantage.

For small alphabets of just a few hundred symbols codeword overflow is usually assumed to be a remote possibility. In reality, an alphabet of as few as 34 symbols can force a 33-bit codeword. However, for overflow to happen the least probable symbol must turn up fewer than one time in 10 million (Witten *et al.*, 1994), which seems a safe bet, since reasonably accurate statistics for a small alphabet can be achieved after just a few thousand or tens of thousands of symbols have been processed. On the other hand, when the alphabet is large, the possibility of codeword overflow becomes quite real—one imagines that when accumulating frequencies on an alphabet of 1 million symbols it is highly likely that several hundred million symbols in total are to be processed. In such a system it is thus desirable for a length-limit L to be imposed, an upper bound on the permitted length of the generated codewords. This entails some amount of compression loss, but throughput rates are preserved.

Of course, one could also move to more powerful hardware, such as a 64-bit machine. At least 4.5×10^{13} symbols must be processed before a 65-bit codeword can be generated and so the only real drawback of this alternative is expense. It is, however, pleasing to solve the codeword overflow problem in an elegant algorithmic manner rather than apply a brute-force solution and that is the approach we describe here.

A further advantage of length-limited coding is that the use of length-limit can ameliorate the undesirable consequences of underestimating the source probabilities (Gilbert, 1971). With a length-limit in force symbols predicted to be of very low probability cannot generate excessively long codewords and so if they turn out to be more frequent than expected, the compression degradation is, to a certain extent, controlled.

Many authors have considered the problem of generating length-limited codes or, equivalently, minimum-cost depth-limited binary search trees (Hu and Tan, 1972; Garey, 1974; Van Voorhis, 1974). These early algorithms were intractable in terms of either space or time, or both. Several other algorithms describe heuristics that build length-limited codes, but do not guarantee that the code is minimum cost (Murakami *et al.*, 1984; Fraenkel and Klein, 1993). It was not until 1990 that an efficient solution to the problem of finding minimum-cost length-limited codes was articulated, by Larmore and Hirschberg (1990). They described two algorithms. The first—known as the *package-merge* technique—takes $O(nL)$ time and $O(nL)$ space to generate codewords limited to L bits for an alphabet of n symbols. Package-merge requires an explicit data structure and, supposing three words of memory for each of the $2nL$ nodes required by the structure, requires more than 700 Mb of memory to generate a 32-bit limited code for an alphabet of 1 million symbols.

The second algorithm described by Larmore and Hirschberg (1990) is a refined version of the first and is called the recursive *package-merge* method. It uses a controlled amount of re-evaluation to reduce the space requirement to $O(n)$ within the same $O(nL)$ time limit. This algorithm is asymptotically efficient, but for our purposes the constant factor is all-important and the constant factor here is 16 words. To build a length-limited code for one million symbols requires 64 Mb of memory.

In this paper we consider methods for reducing these memory requirements. Starting with the $O(nL)$-space package-merge algorithm we develop a new implementation with reduced space consumption and the same $O(nL)$-time, $O(n)$-space asymptotic behaviour as the recursive package-merge method. The improvement is done in two stages. In the first stage we describe a compact representation of the lists of trees used by Larmore and Hirschberg (1990) and show how they can be manipulated to generate minimum-cost codes. In total, this method requires $2n + 2nL/w + O(L)$ words of memory to generate minimum-cost length-limited codes for an alphabet of n symbols, where w is the number of bits per word in the machine being used. In many applications $w = L$ (after all, this was one of the arguments used earlier to motivate the need for length-limited codes) and in these cases $4n$ words suffice, or 16 Mb for the hypothetical alphabet of 1 million symbols. Furthermore, the relationship between L and the word size w, explored in detail below, is such that the space requirement is less than $5n$ words for all viable combinations of w and L.

In the second stage we add a further twist—instead of calculating and storing that which is required to generate the final set of codes, we calculate that which is not and then infer the set of codes from the 'negative' information so obtained. It turns out that there is significantly less negative information to be stored than there is positive information, and the space required is correspondingly reduced. The final algorithm requires $2n + 2n(L - \log_2 n)/w + O(L)$ words. For 1 million symbols and $w = L = 32$, about 11 Mb of memory suffices. Note that the space usage after the second stage improvement can be no worse than the first stage algorithm and so the worst case space bound is again $O(n)$.

A statement of the length-limited coding problem appears in Section 2 and the package-merge solution of Larmore and Hirschberg is described in Section 3. Section 4 then describes the first improvement, the use of a compact structure to record the progress of the package-merge method. Section 5 re-examines the requirements of the package-merge method and shows how the memory requirement can be further reduced. The results of generating length-limited codewords on a distribution containing more than 1 million symbols are described in Section 6.

2. PREFIX CODES

Suppose that in some stream of symbols there are n

distinct symbols; and that the ith least frequent symbol appears p_i times. That is, we suppose that $p - [p_i | i \in \{1...n\}]$ is a list of n positive integers, with $p_1 \leq p_2 \leq ... \leq p_n$. The requirement that p be sorted is not onerous—in any particular situation if this is not already the case it can be achieved by a preprocessing step in $O(n \log n)$ time, which does not dominate any of the algorithms described here. Note, however, that presorting does add n words to the stated space requirements of all methods considered here. The extra space is used by an index array that records the original order of p.

A *code* is a list of n integers $l = [l_i | i \in \{1...n\}]$, where it is presumed that the ith symbol is to be represented by a binary codeword of length l_i over the set $\{0,1\}$.

A *prefix-free code* (sometimes known as a *prefix code*) is a code for which $\sum_{i=1}^{n} 2^{-l_i} \leq 1$. For example, assigning $l_i = \lceil \log_2 n \rceil$ is a prefix-free code, since $n \cdot 2^{-\lceil \log_2 n \rceil} \leq 1$. Given a prefix-free code $l = [l_i]$ it is straightforward to determine a set of n *codewords*, one per distinct symbol, with the property that the codeword for symbol i is exactly l_i bits long and such that no codeword in the set is a proper prefix of any other. A code for which $\sum_{i=1}^{n} 2^{-l_i} > 1$ is *ambiguous*. Where there is no possibility of confusion K is used to denote the quantity $\sum_{i=1}^{n} 2^{-l_i}$.

Once a prefix-free code has been determined and a set of codewords is known they can be used to generate a representation of the input stream and, perhaps, result in a storage reduction compared to the original representation. However we do not concern ourselves with the steps that actually assign final codewords or use them and will regard our task as being over when, for each symbol, a codeword length is assigned. One method for assigning codewords that leads to fast decoding is summarised in Witten *et al.* (1994).

A *minimum-cost code* (or *minimum-cost prefix-free code*) is a set of codeword lengths l_i such that not only is $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ satisfied, but also such that $B = \sum_{i=1}^{n} l_i p_i$ is minimized over all prefix-free codes. Quantity B is the number of output bits used by the code to represent the input stream; a code is minimum-cost if there is no other code that results in an output representation requiring fewer than B bits. Note that for any list $p = [p_i]$ there may be more than one minimum-cost code; for the assignment $p = [1,1,2,2]$ both $l = [3,3,2,1]$ and $l = [2,2,2,2]$ result in compressed representations that require $B = 12$ bits. A minimum-cost code maximises the value of K, i.e. $\sum_{i=1}^{n} 2^{-l_i} = 1$.

An *L-limited code* (or *L-bit length-limited prefix-free code*) is a set of codeword lengths l_i that not only satisfies $\sum_{i=1}^{n} 2^{-l_i} \leq 1$, but is also such that $l_i \leq L$ for all $1 \leq i \leq n$, where L is some predetermined integer number of bits. Finally, a *minimum-cost L-limited code* (or *minimum-cost L-bit length-limited prefix-free code*) is an L-limited code such that $B = \sum_{i=1}^{n} l_i p_i$ is minimal over all L-limited codes for constraint L.

The problem we consider is: given a list of symbol frequencies $p = [p_1, p_2, ..., p_n]$ and a bound L, derive a minimum-cost L-limited code $l = [l_1, l_2, ..., l_n]$. As always, we seek efficient algorithms, and are interested in bounding the running time and memory space consumed by any method as functions of n and L.

In all of the algorithms that follow we will count n words of storage for the input list p (i.e. p is assumed to be stored in an n-item array) and L words for the output list l. The latter bound is possible because for sorted input there is at least one minimum-cost code for which $L \geq l_1 \geq l_2 ... \geq l_n \geq 1$, and so it suffices to generate an L-item list C, with $C[j]$ recording the size of the set $\{l_i : l_i = j\}$. For clarity of description it is, however, convenient to suppose at first that l_i exists as a scalar variable, so that the method of calculating each value l_i is clear. Conversion between these alternative output formats is described in detail below. To be added to this basic cost is the memory space required by auxiliary data structures, and it is this space that we wish to minimize. In this framework an algorithm that requires $n + O(1)$ words of auxiliary structures will be recorded as requiring $2n + L + O(1)$ words of memory in total.

The model of computation we assume is a unit-cost random access machine, in which values as large U can be stored in a single word, where $U = \sum_{i=1}^{n} p_i$ is the sum of the input frequencies and is the largest value manipulated during the execution of most code-generation algorithms. That is, we suppose that addition and comparison operations on integer values in the range $1...U$ require $O(1)$ time each. Note that if w is the number of bits in a machine word, then for any code-generation program to assume unit-cost operations, it must be the case that $w \geq \lceil \log_2 U \rceil$. We will make use of this fact below.

3. THE PACKAGE-MERGE ALGORITHM

In this section we describe the (non-recursive) package-merge algorithm of Larmore and Hirschberg (1990).

In order to build a minimum-cost L-limited code we begin with each of the n symbols assigned $l_i = 0$. The resulting code is certainly efficient (one cannot hope to do better than $B = 0$) and meets the length limit, but is ambiguous, since $K = \sum_{i=1}^{n} 2^{-l_i} = n$. The latter quantity can only be reduced if one or more of the l_i's are increased. What is required is a mechanism for deciding which subset of the l_i's should be increased (and by what amount) to make the code unambiguous, without violating the length limit, and with minimal increase in B. Incrementing l_1 to one reduces the sum K by 0.5 and has, over all possible changes, the minimum impact upon the number of output bits B, since p_1 is the smallest frequency count. Incrementing l_2 to 1 is then guaranteed to achieve a further 0.5 reduction in K with minimal impact upon B. After these two steps $K = n - 1$ and B is $p_1 + p_2$.

The third step is not so obvious. There is now a choice—either the third least frequent symbol can be

```
1. Suppose that $p$ is an array of $n$ frequencies, and that $p[1] \le p[2] \le \cdots p[n]$. Let
   $L$ be the length limit.
2. If $L < \lceil \log_2 n \rceil$ return with failure, no prefix code is possible.
3. /* First list is a copy of the input array */
   Set $Q[1] \leftarrow p$.
4. /* Calculate $L-1$ more lists */
   For $i \leftarrow 1$ to $L-1$ do
       Set $Q[i+1] \leftarrow merge(p, package(Q[i]))$.
5. Set $l \leftarrow [0,0,\ldots,0]$
6. For $j \leftarrow 1$ to $2n-2$ do
   (a) Recursively expand $Q[L,j]$, tracing its composition through the pointers
       established by package.
   (b) For each symbol $k$ such that $p[k]$ appears as a leaf in the tree rooted at
       $Q[L,j]$
           Set $l_k \leftarrow l_k + 1$.
7. Return the list $[l_i]$

where package($Q[i]$) consists of
1. Set output_list $\leftarrow$ [].
2. For $j \leftarrow 1$ to $|Q[i]|$ div 2 do
   (a) Make a new node.
   (b) Link $Q[i,2j-1]$ and $Q[i,2j]$ as left and right subtrees respectively of the
       new node.
   (c) Set the weight of the new node to the sum of the weights of $Q[i,2j-1]$
       and $Q[i,2j]$.
   (d) Append the new node to output_list.
3. Return output_list.

and where merge is a standard list merge based upon node weights.
```

FIGURE 1. Package-merge algorithm.

assigned $l_3 = 1$ or to obtain the same 0.5 reduction in K, the first two symbols can be jointly incremented, setting $l_1 = l_2 = 2$. In the first case the increase in B is by p_3 and in the second, B goes up by $p_1 + p_2$. To follow the path of minimal increase we choose the smaller of these two and increment l_i's accordingly.

Continuing in this way for a total of $2n-2$ 'increments', each of which reduces K by 0.5, brings us to the point at which the code stops being ambiguous and becomes complete ($K = 1$). Provided that the set of symbols chosen in each increment generates a minimal increase in B and that no increment is allowed to boost any l_i value beyond L, then the final list $[l_i]$ is a minimum-cost L-limited code. The package-merge algorithm stipulates a mechanism for enumerating packages of symbols that allows easy recognition of which $2n-2$ increments should be performed to obtain $K = 1$, with minimal increase in B.

The full algorithm is sketched in Figure 1. In Figure 1, Q is a list of lists, so that $Q[i]$ is the ith list and $Q[i,j]$ is the jth element of that ith list. The algorithm generates, in a bottom-up manner, a list $Q[L]$ of *items*. A total of $L-1$ other lists must also be formed and these are stored as lists $Q[1]$ through to $Q[L-1]$. Each item in list i is a binary tree whose leaves describe which l_k's to increase to yield a $2^{-(L-i+1)}$ reduction in K. That is, each item describes some of the increments mentioned above. Each list is generated in ascending order of impact upon B and so the first $2n-2$ items of $Q[L]$, each of

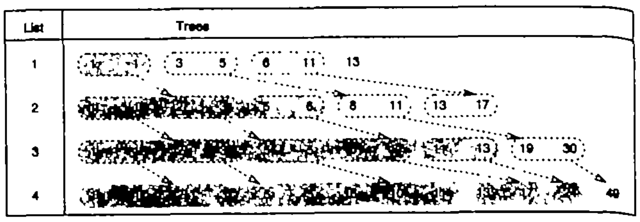

FIGURE 2. Package-merge on $p = [1,1,3,5,6,11,13]$ with $L = 4$.

which account for a 0.5 reduction in K, describe a minimum-cost L-limited code.

The generation of $Q[L]$ is based upon the original list of probabilities p and the previous list of items $Q[L-1]$, which represents increments each of which decrease K by 0.25. List $Q[L-1]$ is, in turn, built out of an earlier list and, provided that there are exactly L lists in total, no single l_i value can be incremented more than L times. The items in list $Q[1]$ are leaves with weight equal to their original frequency in p; each increment on this list takes one codeword to $l_i = L$ bits long, reduces K by 2^{-L} and adds p_i to B. Once $Q[L]$ has been generated it is simply a matter of choosing the first $2n-2$ of its items and increasing the l_i's corresponding to each leaf in the tree rooted at each item.

Figure 2 shows an example of the package-merge algorithm when applied to the list p =[1,1,3,5,6,11,13] with $L = 4$. A minimum-cost code is l =[5,5,4,3,2,2,2], with $B = 97$ bits. Restricting codeword lengths to four bits results in the first two symbols having reduced codeword lengths and one or more other symbols having a longer codeword in order to maintain $K = 1$. To calculate the code, the first 12 items from $Q[4]$ must be expanded, where $12 = 2n-2$ is the number of 0.5 increments necessary to reduce K from 7 to 1. These items are shown in grey. They involve, not surprisingly, all of the seven l_i values, plus the expansion of five packages, which correspond in turn to the first 10 items of $Q[3]$. Inspection of the list $Q[3]$ shows that to obtain these 10 items, seven leaves must be traversed, causing increments so that $l_i = 2$ for all i and three packages must be expanded. These three packages are drawn from the first six items in $Q[2]$; they in turn cause one package—two items— to be expanded out of $Q[1]$. Accumulating all of the individual increments on symbols (all of the grey regions), the final code is l =[4,4,3,3,3,2,2] (i.e. $C[4\ldots1] = [2,3,2,0]$), which results in $B = 98$. By taking the first 12 items from $Q[3]$ the same table can also be used to generate a 3-limited code; in this case the answer is $l = [3,3,3,3,3,3,2]$ (or $C[3\ldots1] = [6,1,0]$) and $B = 107$. There is no 2-limited code possible for $n = 7$ symbols.

Let us now examine the space required by the package-merge process. Implemented as described here, there are L lists generated, $Q[1]$ to $Q[L]$, and each contains a mixture of n symbols and at most $n-1$ packages. The latter bound is shown by the following inductive argument. The first list contains no packages and so a base for the induction is established. Assume that list $Q[i]$

contains n symbols and at most n-1 packages. List $Q[i+1]$ must contain $\lfloor (n+n-1)/2 \rfloor$ packages, which is no greater than $n-1$, completing the inductive step. That is, each list contains at most $2n-1$ items, and so there are fewer than $2nL$ items in the entire structure.

Each item requires a field for its weight, and left and right pointers (if it is a package) showing its composition. Leaves (i.e. symbols) can be indicated by the use of null pointers. Since the items are generated in increasing order of weight they can be stored in an array rather than linked together and so the list ordering can be managed implicitly rather than with an explicit pointer. Even so, assuming that pointers and weights occupy one word, a total $6nL$ words of memory are consumed. For an alphabet of 1 million symbols with $L = 32$ the structure occupies 730 Mb, a formidable requirement.

4. COMPACT PACKAGES

The first improvement to the package-merge algorithm involves representing each package with a single bit, rather than as a three-word item. Section 4.1 introduces this alternative representation and Section 4.2 describes a method for generating it efficiently. The relationship between the length-limit imposed on codewords, L, and the machine word size, w, is examined in Section 4.3, resulting in a tight asymptotic bound for the space usage of the new representation. The use of the list C to store a description of the minimum-cost L-limited code is described in Section 4.4.

4.1. Package representation

The key to reducing the space required by package-merge is the observation that it is not necessary to permanently record the composition of each package, only the fact that it *is* a package. Consider again the example shown in Figure 2. In the first 12 items of list $Q[4]$ there are seven symbols and five packages; those packages must, of necessity, correspond to the first 10 items of list $Q[3]$. Similarly, the three packages within the first 10 items of $Q[3]$ must correspond to the first six items in list $Q[2]$ and the single package expanded from $Q[2]$ must have as its source the first two items in $Q[1]$.

Suppose then that a bitmap M is added, with $M[i,j]$ set to 1 if the jth item in list $Q[i]$ is a package (internal tree node) and to 0 if it is a symbol (leaf). Bitmap M requires $2nL$ bits and if there are w bits per word on the machine being used, consumes $2nL/w$ words.

Furthermore, observe that the need to know the actual weight of each item is only temporary and once $Q[i+1]$ is constructed there is no need for any detail of $Q[i]$ to be preserved aside from the bitmap $M[i]$, which has the responsibility of showing the ordering of items in $Q[i]$. This observation suggests a swing buffer arrangement, in which $Q[i]$ is maintained in one array of $2n$ words until $Q[i+1]$ is constructed and then overwritten by $Q[i+2]$ during its construction. The next list, $Q[i+3]$, is then developed from $Q[i+2]$ and overwrites $Q[i+1]$, and so on. These two $2n$-word buffers, plus one n-word array to store p, are sufficient, since M records all of the other necessary information from which the L-limited code can be built. In total, $2nL/w + 5n + L$ words are required by this variant.

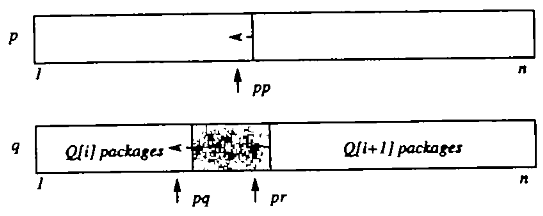

FIGURE 3. In-place calculation of $Q[i+1]$ from p and $Q[i]$.

4.2. In-place package development

In fact, with careful management it is possible for $Q[i]$ and $Q[i+1]$ to coexist in a single array q of just n words, further reducing the space requirement. The improved procedure is illustrated in Figures 3 and 4, in which $Q[i]$ is still logically the ith list of the package-merge process, but is not stored and manipulated directly. Instead, at any given stage both $Q[i]$ and $Q[i+1]$ are represented within the array q, which stores packages only. To be sure that the two uses of array q coexist rather than collide, the packaging and merging must now be performed from the largest item down. Reversing the direction of the merge means that $Q[i]$ is of odd length, the largest item—either a symbol or a package—must be discarded before any packages are formed (step 3 in Figure 4). Note also that the termination condition

```
1. Suppose that $p[1]\ldots p[n]$ is the original array of symbol frequencies, and that $p$
   and $q[1\ldots r]$ jointly store the $i$th list $Q[i]$, where $r$ is the number of packages in
   $Q[i]$ and $Q[i]$ has $n+r$ items in total.
2. Set $pp \leftarrow n$, $pq \leftarrow r$, $pr \leftarrow n$, and $j \leftarrow n+r$.
3. /* Discard the last item if list length is odd */
   If $(n+r) \bmod 2 = 1$ then
       If $q[pq] > p[pp]$ then
           Set $pq \leftarrow pq - 1$
       else
           Set $pp \leftarrow pp - 1$.
4. While $pp \neq 0$ do
   (a) /* Find first item for next package */
       if $p[pp] < q[pq]$ then
           Set $q[pr] \leftarrow q[pq]$, $M[i,j] \leftarrow 1$,
               $pq \leftarrow pq - 1$, and $j \leftarrow j - 1$
       else
           Set $q[pr] \leftarrow p[pp]$, $M[i,j] \leftarrow 0$,
               $pp \leftarrow pp - 1$, and $j \leftarrow j - 1$.
   (b) /* Find other item */
       Set $pr \leftarrow pr - 1$,
       Repeat step 4a.
   (c) /* Form package */
       Set $q[pr+1] \leftarrow q[pr] + q[pr+1]$.
5 /* Move left in q */
  Set $r' \leftarrow (n+r)/2$,
  Shift each item in $q[pr+1\ldots n]$ left $pr$ positions.
6 List $Q[i+1]$ is now stored jointly in $p$ and $q[1\ldots r']$, in the same form as was
  $Q[i]$ at the beginning of the process.
```

FIGURE 4. In-place calculation of $Q[i+1]$ from p and $Q[i]$.

shown at step 4 of Figure 4 is somewhat simplified and a more careful test is required in the actual implementation.

Merging from largest to smallest guarantees that q has enough room for $Q[i]$ and $Q[i+1]$ to coexist. All of the largest items in $Q[i]$ must be packages, and so in the initial stages of the algorithm detailed in Figure 4, pq moves left faster than pr; and since pq starts at least one position to the left of pr (the maximum value of r is $n-1$, by the argument used above to bound the space of the original implementation described), pq can never be overtaken by pr. The relative locations of the pointers pp, pq and pr are shown in Figure 3. The grey area depicts where the two values generated by step 4(a) reside. Once the merge is complete then the entries in q are shifted to the left so that they end up in positions $1...r'$.

While $Q[i+1]$ is being built the bitvector $M[i]$ is constructed. The merge can thus be performed in $2n + 2nL/w$ words—n for the list p, which is not altered; n for the array q, to hold all the packages; and $2nL/w$ for the bitmap M. A further $L+1$ words are required for the output list C of code length counts. For the hypothetical problem of $n = 1\,000\,000$ and $L = 32$ this amounts to 16 Mb, assuming $w = 32$.

4.3. Allowable values of L

The model of computation assumed in Section 2 allows integer values as large as $U = \sum_{i=1}^{n} p_i$ to be stored in a single machine word and manipulated using $O(1)$-time operations. Let us now consider the effect that this assumption has upon the space required for the bitmap M. It was already observed that if w is the number of bits per word, then $w \geq \lceil \log_2 U \rceil \geq \log_2 U$. It remains to calculate an upper bound for L.

If H is the length of the longest codeword in a minimum-cost (i.e. unrestricted length) code for p, then $L \leq H$. We can easily guarantee this constraint with the addition of a preprocessing step that calculates a minimum-cost code—using, perhaps, the method of Moffat and Katajainen (1995)—and accepts that code as an L-limited code if the longest codeword is already shorter than the required bound L.

The maximum length of a minimum-cost code has been extensively studied (see, for example, Katona and Nemetz (1976) and Buro (1993)), and is related to the Fibonacci sequence. Define $F(0) = F(1) = 1$ and $F(k) = F(k-1) + F(k-2)$ for $k \geq 2$. Then for a minimum-cost code to have H bits, it must be that $U = \sum_{i=1}^{n} p_i \geq F(H+2)$. Define $\phi = (1+\sqrt{5})/2$, the golden ratio. Graham *et al.* (1989) show that $F(k)$ may be evaluated as

$$\begin{aligned} F(k) &= \left\lfloor \frac{\phi^k}{\sqrt{5}} + \frac{1}{2} \right\rfloor \\ &\approx \frac{\phi^k}{\sqrt{5}} \\ &\geq \phi^{k-2}, \end{aligned}$$

with the final inequality holding because $\phi^2/\sqrt{5} > 1$.

```
1. Suppose that $M$ has been created, that $L$ is the length-limit, and that there are
   $n$ symbols.
2  Set $C[L \ldots 0] \leftarrow [0, \ldots, 0, n]$,
       $i \leftarrow L$, and $t \leftarrow 2n - 2$.
3. While $t \neq 0$ do
   (a) Set $t' \leftarrow 0$.
   (b) For $j \leftarrow 1$ to $t$ do
           If $M[i, j] = 1$ then
               Set $t' \leftarrow t' + 1$
           else
               Set $C[L - i] \leftarrow C[L - i] - 1$ and
                   $C[L - i + 1] \leftarrow C[L - i + 1] + 1$.
   (c) Set $t \leftarrow 2t'$ and $i \leftarrow i - 1$.
4. Return $C[L \ldots 1]$, it describes a minimum-cost $L$-limited code for $p$.
```

FIGURE 5. Determining C from M.

For a minimum-cost code to have a longest codeword of H bits, $U \geq \phi^H$, i.e. $L \leq H \leq \log_\phi U$.

Combining these inequalities on w and L we see that

$$\begin{aligned} \frac{2nL}{w} &\leq \frac{2n \log_\phi U}{\log_2 U} \\ &= 2n \log_\phi 2 \\ &\approx 2.88n \end{aligned}$$

That is, for all viable combinations of L and U, $2.88n$ words of memory suffice for matrix M and a total of $5n$ words of memory is adequate for all of the data structures used in the improved package-merge implementation.

4.4. Storing the solution

To generate the list C we process the bitmap M in the manner shown in Figure 5. Bits are inspected across each row of M, starting at row L; one bits indicate packages that need to be expanded from the previous level, the number of which is counted using t'; while zero bits indicate leaves that should have their code lengths increased by one bit. These are counted into $C[L-i+1]$ and removed from the count in $C[L-i]$. The process can stop as soon as t, the number of items to be expanded, reaches zero. The initial value of t for row L is $2n-2$, as discussed in Section 3.

5. TRIMMING THE EDGES

The discussion in Section 4 assumed that M was a rectangular bitmap of L rows and $2n$ columns. We can, however, calculate the exact number of bits required for each row in advance, thus reducing the total number of bits to be stored. For example, none of the items stored in $Q[1]$ are packages and so $M[1]$ need not be stored at all. Similarly $Q[2]$ contains $\lfloor n/2 \rfloor$ packages, and so is $n + \lfloor n/2 \rfloor$ items long, and $Q[3]$ contains $\lfloor (n + \lfloor n/2 \rfloor)/2 \rfloor$ packages and so on. By shortening each of the rows of M to eliminate the unused storage a total of $2n + n/2 + n/4 + \ldots = 3n$ bits can be saved.

At the other end of M even greater savings are possible. Consider $Q[L]$, the last list. We know in

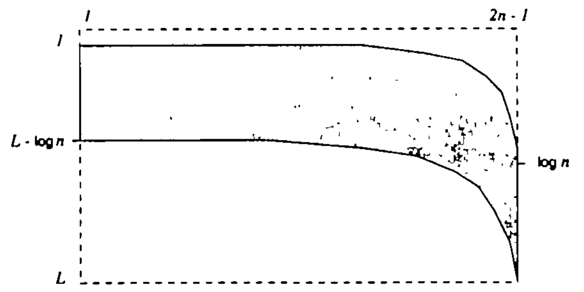

FIGURE 6. The necessary sections of M.

advance that $2n - 2$ items will be examined. Moreover, there must be exactly n leaves amongst those items, since if this were not the case there would be symbols for which $l_i = 0$ at the end of the process, which is clearly untenable. That is, of the $2n - 2$ items of $Q[L]$ that are examined, exactly $n - 2$ must be packages. For example, in Figure 2 for which $n = 7$, there were five packages amongst the 12 items expanded for $Q[L]$. Moreover, it is only the number of packages involved—and not their positions—that affects the final code. There is no need to store any elements of $M[L]$ and a further $2n$ bits can be saved.

Further savings are possible in $Q[L - 1]$. We know that $2n - 4$ items will be required from the front of $Q[L - 1]$ to build the required $n - 2$ packages. We also know that $Q[L - 1]$ is at most $2n - 1$ items long (the exact length of $Q[L - 1]$ can be deduced without actually constructing it) and contains exactly n leaves. Hence, if we know for each of the at most three (depending upon the exact length) final items on $Q[L - 1]$ whether or not it is a package, we can calculate how many of the first $2n - 4$ items are packages. That is, three bits or fewer suffice to represent $M[L - 1]$. Furthermore, since the merge takes place from largest to smallest, only a small subset of the computational effort is required. Note that here we have assumed the worst case scenario of list $Q[L - 1]$ being $2n - 1$ items long. In practice the exact length of list $Q[L - 1]$ may be less.

In general, suppose at least $2n - 2^{i+1}$ items (either packages or symbols) are to be expanded from the front list $Q[L - i]$. Since at most n of these can be leaf symbols, there must be at least $n - 2^{i+1}$ packages in $Q[L - i]$. Each package in $Q[L - i]$ corresponds to two items in the previous list $Q[L - i - 1]$ and so in list $Q[L - i - 1]$ at least $2n - 2 \cdot 2^{i+1} = 2n - 2^{i+2}$ items must be expanded. The basis for the induction was established when it was noted that $2n - 2$ items must be expanded in $Q[L]$ and so the claim is correct.

Since $Q[L - i]$ is at most $2n - 1$ items long, it is only necessary for M to record the composition of the final at most $2n - 1 - (2n - 2^{i+1}) = 2^{i+1} - 1$ items, since none of the earlier items can influence the final code. That is, three bits suffice for $M[L - 1]$, seven bits for $M[L - 2]$ and so on, until the full $2n - 1$ bits are required for row $M[L - \log_2 n]$. The net effect of these two approaches—savings at the end of each row of $M[i]$, but decreasing as i increases from one, and savings at the beginning of each row $M[i]$, increasing as i increases towards L—is that although there are potentially L bits in each column of M, only $L - \log_2 n$ of them are actually required. The matrix M is shown in Figure 6; only the shaded region need be stored. Since there are still notionally $2n - 1$ columns, the space occupied by M is thus reduced to $2n(L - \log_2 n)$ bits or $2n(L - \log_2 n)/w$ words. The running time is similarly reduced to $O(n + n(L - \log n)/w)$. When L is close to $\log_2 n$—and there is the most pressure on the length limit—the total amount of space required to devise a length-limited code is close to $2n$ words and the running time is almost linear.

6. EXPERIMENTAL RESULTS

Our investigation has been motivated by the work we have been undertaking with the *TREC* collection (Harman, 1992), a large corpus of text drawn from several sources including newspapers, government publications, and the US Patent Office. As originally processed, *TREC* consisted of two gigabytes of text. We have been compressing *TREC* using a zero-order word-based model and minimum-cost coding; there are about 350 million symbols in total and nearly 1 million distinct symbols. By luck, the minimum-cost code on this

TABLE 1. Results for the generation of codes on the *TREC* collection

		Length-limit (bits)		
	Method	22	27	32
Space (Mb)	recursive package-merge	65.5	65.5	65.5
	improved package-merge	8.7	10.0	11.3
	Fraenkel and Klein	8.2	8.2	8.2
	in-place minimum-cost	—	—	4.1
Time (s)	recursive package-merge	121.1	165.1	176.7
	improved package-merge	7.9	14.6	20.6
	Fraenkel and Klein	2.1	2.1	1.5
	in-place minimum-cost	—	—	1.3
Compression (bits/word)	Fraenkel and Klein	14.460	11.526	11.521
	package-merge	11.846	11.523	11.521
	minimum-cost	—	—	11.521

distribution yielded codewords that peaked at 30 bits and so did not present a problem on 32 bit machines (Moffat and Zobel, 1994). However, the collection has recently grown by another gigabyte, and it became apparent that the existing method of generating codewords could not be used indefinitely.

To gauge the effectiveness of the new package-merge implementation, we used it to generate 22-,27- and 32-bit length-limited codes for the full three gigabyte *TREC* word distribution, which contains $n = 1\,073\,971$ distinct symbols and $U = 480\,911\,085$ symbols in total. The most frequent symbol—the word 'the'—appears 23 795 386 times, with an overall probability of 4.94%.

The results of these experiments are shown in Table 1. We compared the new implementation against the $O(n)$-space recursive package-merge of Larmore and Hirschberg, the $O(n)$-space approximate length-limited coding method of by Fraenkel and Klein (1993) (including all of the improvements they describe) and against the in-place minimum-cost code generation method described by Moffat and Katajainen (1995).

The first section of Table 1 shows the data space consumed by actual implementations, including all allocated arrays and bitvectors, and including the n words of memory required to specify the input list p. The improved implementation requires substantially less memory than the $O(n)$-space method of Larmore and Hirschberg, and only a little more space than Fraenkel and Klein's approximate method. (The space required by Fraenkel and Klein's method can be halved if the refinements they mention are not implemented. The resultant code is, however, less efficient.)

The execution times shown in the second part of Table 1 show the corresponding amount of CPU time required when these programs were executed on a Sun SPARC 10 Model 402. It is assumed that the input to the programs is a sorted list of frequencies, and so the time taken to read and sort the frequencies is not included. If the input is in fact not already sorted, an extra array of n words (4.1 Mb) is required to record the permutation of the input generated by the sorting process and an allowance of 3.15 s should be added to the execution times. Although not as fast as the Fraenkel and Klein heuristic approach, the new implementation is substantially quicker than the recursive package-merge method. This latter algorithm performs a controlled but nevertheless non-trivial amount of repeated computation and this extra work accounts for the bulk of the difference.

It is also interesting to measure the compression inefficiency introduced by the use of length-limited codes. The final section of Table 1 shows the average codelength achieved by the various methods at the three length limits, measured as the average number of output bits per input symbol (in this case, a word in English text). Codes of fewer than $L = 21$ bits are not possible for the 3 Gb *TREC* collection, since the vocabulary contains $1\,073\,971 > 2^{20}$ distinct words. The unrestricted minimum-cost code is just 0.27% inefficient compared to the entropy of the distribution and so there would be almost no gain to be had by using arithmetic coding for this problem. Relatively severe length limits have surprisingly little effect on compression efficiency, provided that minimum-cost codes are used.

7. SUMMARY

We have examined the package-merge method of generating minimum-cost L-limited prefix codes for an alphabet of n symbols, where n might be large. Although the $O(n)$-space algorithm of Larmore and Hirschberg is asymptotically efficient, in practice it is both memory-extravagant and relatively slow.

The alternative implementation described here has the same asymptotic space and time bounds, but for practical application uses substantially less of both resources. It runs sufficiently quickly and within sufficiently limited memory spaces that it can be used as a realistic alternative to standard minimum-cost coding. For example, to actually compress the *TREC* collection takes several hours, and so the extra seconds required to generate a length-limited code are easily justified.

Further improvements in the package-merge method are also possible. In collaboration with J. Katajainen we have recently devised an implementation of the package-merge paradigm that operates in $n + O(L^2)$ words of memory and $O(nL)$ time (Katajainen *et al.*, 1995). A number of related problems have also received attention recently: Larmore and Przytycka (1994, 1995) have given new solutions for the *length-limited alphabetic coding* problem and the *parallel minimum-cost coding* problem; and Schieber (1995) has also given an asymptotically time-efficient algorithm for minimum-cost length-limited coding.

ACKNOWLEDGEMENTS

We gratefully acknowledge the assistance of Jyrki Katajainen (University of Copenhagen, Denmark) and Tomi Klein (Bar-Ilan University, Israel). This work was supported by the Australian Research Council.

REFERENCES

Bell, T. C., Cleary, J. C. and Witten, I. H. (1990) *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.
Buro, M. (1993) On the maximum length of Huffman codes. *Information Proc. Lett.*, **45**, 219–223.
Fraenkel, A. S. and Klein, S. T. (1993) Bounding the depth of search trees. *Comp. J.*, **36**, 668–678.
Garey, M. R. (1974) Optimal binary search trees with restricted maximal depth. *SIAM J. Comput.*, **3**, 101–110.
Gilbert, E. N. (1971) Codes based on inaccurate source probabilities. *IEEE Trans. Information Theory*, **IT-17**, 304–314.
Graham, R. L., Knuth, D. E. and Patashnik, O. (1989) *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA.
Harman, D. K. (1992) Overview of the first Text Retrieval Conference. In Harman, D. K. (ed.), *Proc. TREC Text*

Retrieval Conf. National Institute of Standards Special Publication 500-207, pp. 1-20, NIS, Washington, DC.

Hu, T. C. and Tan, K. C. (1972) Path length of binary search trees. *SIAM J. Appl. Math.*, **22**, 225-234.

Huffman, D. A. (1952) A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, **40**, 1098-1101.

Katajainen, J., Moffat, A. and Turpin, A. (1995) A fast and space-economical algorithm for length-limited coding. In *Proc. Int. Symp. on Algorithms and Computation*. Springer-Verlag, Berlin.

Katona, G. O. H. and Nemetz, T. O. H. (1976) Huffman codes and self-information. *IEEE Trans. Information Theory*, **IT-22**, 337-340.

Larmore, L. L. and Hirschberg, D. S. (1990) A fast algorithm for optimal length-limited Huffman codes. *J. ACM*, **37**, 464-473.

Larmore, L. L. and Przytycka, T. M. (1994) A fast algorithm for optimum height-limited alphabetic binary trees. *SIAM J. Comput.* **23**, 1283-1312.

Larmore, L. L. and Przytycka, T. M. (1995) Constructing Huffman trees in parallel. *SIAM J. Comput.*, in press.

Manstetten, D. (1992) Tight upper bounds on the redundancy of Huffman codes. *IEEE Trans. Information Theory*, **IT-38**, 144-151.

Moffat, A. and Katajainen, J. (1995) In-place calculation of minimum-redundancy codes. In *Proc. Workshop on Algorithms and Data Structures*. Springer-Verlag, Berlin.

Moffat, A. and Zobel, J. (1994) Compression and fast indexing for multi-gigabyte text databases. *Aust. Comp. J.* **26**, 1-9.

Moffat, A., Sharman, N., Witten, I. H. and Bell, T. C. (1994) An empirical evaluation of coding methods for multi-symbol alphabets. *Information Proc. Management*, **30**, 791-804.

Murakami, H., Matsumoto, S. and Yamamoto, H. (1984). Algorithm for construction of variable length code with maximum word length. *IEEE Trans. Commun.*, **COM-32**, 1157-1159.

Schieber, B. (1995) Computing a minimum-weight k-link path in graphs with the concave Monge property. In *Proc. 6th Ann. Symp. Discrete Algorithms*. SIAM, Philadelphia, PA, pp. 405-411.

Sedgewick, R. (1990) *Algorithms in C*, 2nd edn. Addison-Wesley, Reading, MA.

van Leeuwen, J. (1976) On the construction of Huffman trees. In *Proc. 3rd Int. Coll. on Automata, Languages, and Programming*, pp. 382-410.

Van Voorhis, D. C. (1974) Constructing codes with bounded codeword lengths. *IEEE Trans. Information Theory*, **IT-20**, 288-290.

Van Wyk, C. J. (1988) *Data Structures and C Programs*. Addison-Wesley, Reading, MA.

Witten, I. H., Neal, R. and Cleary, J. C. (1987) Arithmetic coding for data compression. *Commun. ACM*, **30**, 520-541.

Witten, I. H., Moffat, A. and Bell, T. C. (1994) *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York.

Zobel, J. and Moffat, A. (1995) Adding compression to a full-text retrieval system. In *Software—Practice and Experience*. Vol. 25, pp. 891-903.