A Fast and Space-Economical Algorithm for Length-Limited Coding

Jyrki Katajainen[1] Alistair Moffat[2] Andrew Turpin[2]

[1] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk
[2] Department of Computer Science, The University of Melbourne, Parkville 3052, Australia
{alistair,aht}@cs.mu.oz.au

Abstract. The *minimum-redundancy prefix code problem* is to determine a list of integer codeword lengths $l = [l_i \mid i \in \{1 \ldots n\}]$, given a list of n symbol weights $p = [p_i \mid i \in \{1 \ldots n\}]$, such that $\sum_{i=1}^{n} 2^{-l_i} \leq 1$, and $\sum_{i=1}^{n} l_i p_i$ is minimised. An extension is the *minimum-redundancy length-limited prefix code problem*, in which the further constraint $l_i \leq L$ is imposed, for all $i \in \{1 \ldots n\}$ and some integer $L \geq \lceil \log_2 n \rceil$. The package-merge algorithm of Larmore and Hirschberg generates length-limited codes in $O(nL)$ time using $O(n)$ words of auxiliary space. Here we show how the size of the work space can be reduced to $O(L^2)$. This represents a useful improvement, since for practical purposes L is $\Theta(\log n)$.

1 Introduction

Use of Huffman's algorithm [2] for the generation of minimum-redundancy prefix codes for a weighted set of symbols is well known. For practical use an important restriction is to limit the codewords to be at most L bits long, since implementations of data compression methods are usually designed around fixed-width registers. For example, most current computers use words of 32 bits.

Larmore and Hirschberg [3] described the first efficient algorithm for the generation of minimum-redundancy length-limited prefix codes. Their *package-merge* method requires $O(nL)$ time and $O(n)$ space, where n is the number of symbols in the alphabet and L is the length limit. This improves the method of Van Voorhis [9] (see also Hu and Tan [1]), which consumes $O(Ln^2)$ time and space. Asymptotically faster algorithms for solving the problem have been developed recently, see Schieber [7] and the references therein.

In practical applications the speed of the package-merge algorithm is not a problem and the space requirements turn out to be of greater importance. In this paper we describe an improved implementation of the package-merge paradigm—the *boundary package-merge* algorithm—that constructs a minimum-redundancy length-limited prefix code in $O(L^2)$ auxiliary space, while retaining the $O(nL)$ time bound. For most purposes the space required is negligible, since L is typically $\Theta(\log n)$. That is, we have developed an "almost in-place" algorithm for calculating length-limited codes.

2 Prefix Codes

Consider a list $p = [p_i \mid p \in \{1 \ldots n\}]$ of n positive symbol *weights*. For example, p might be the observed frequencies of an alphabet of symbols, as defined by some data compression method. A *code* is an integer list $l = [l_i \mid i \in \{1 \ldots n\}]$, where l_i is the length of the codeword to be assigned to the ith symbol of the alphabet described by p. A *prefix code* (more precisely, *prefix-free code*, but for brevity we use the contraction) is a code for which $K = \sum_{i=1}^{n} 2^{-l_i} \leq 1$. Given a prefix code l, it is straightforward to create a set of n binary codewords such that no codeword is a proper prefix of any other and such that the ith codeword is l_i bits long. In terms of codeword calculation, we can thus consider the task to be essentially done when a code l has been devised. An *L-limited prefix code*, or L-bit *length-limited* prefix code, for some integer $L \geq \lceil \log_2 n \rceil$, is a prefix code in which $l_i \leq L$, for all $i \in \{1 \ldots n\}$.

A prefix code is *minimum-redundancy* if $B = \sum_{i=1}^{n} l_i p_i$ is minimal over all prefix codes. An L-limited prefix code is minimum-redundancy if $B = \sum_{i=1}^{n} l_i p_i$ is minimal amongst all L-limited prefix codes. If the symbol weights are frequencies quantity B is the number of output bits required by the code l.

Here we examine the construction of minimum-redundancy L-limited prefix codes. The model of computation we suppose is a unit-cost random access machine, in which integers as large as $U = \sum_{i=1}^{n} p_i$ can be stored in a single word of storage and manipulated (addition, comparison) in $O(1)$ time. It is also assumed that p is presented in non-decreasing order, $p_1 \leq p_2 \leq p_3 \cdots \leq p_n$. We measure the space requirements of various algorithms by counting the number of extra words of storage required by that algorithm, above and beyond the space required by input list p, which is assumed to be free. In this framework the recursive package-merge algorithm described by Larmore and Hirschberg [3], requires $O(n)$ space, and the improved boundary package-merge described in Section 4 requires $O(L^2)$ space. This difference represents a significant improvement, since for practical use L is a small constant such as 32.

The package-merge algorithm makes use of a *tree* data structure. An *item* s is a tree, and if s is an item and t_1 and t_2 are two trees then $t = s(t_1, t_2)$ is also a tree, with item s the *root* of tree t. If $t = s(t_1, t_2)$ and items s_1 and s_2 are the roots of trees t_1 and t_2 then s is the *parent* of items s_1 and s_2 in tree t. Similarly, items s_1 and s_2 are the *children* of item s. If item s is a singleton and has no children, then it is a *leaf* item of the tree, otherwise it is an *internal* item. To be consistent with the description of Larmore and Hirschberg, we will also refer to internal items as *packages*. The *depth* of any item is one greater than the depth of its parent; the depth of a root is zero.

We also require an analogous one-dimensional structure called a *chain*. A singleton *node* x is a chain; and if y is a chain and x is a node then $x(y)$ is a chain. In this latter case x is the *head* of the chain and y is its *tail*. Note that chains may coalesce; if y is a chain and x_1 and x_2 are two nodes then $x_1(y)$ and $x_2(y)$ are both chains.

3 Package-Merge

In the package-merge algorithm of Larmore and Hirschberg [3] L lists of trees are developed, with each tree in each list having an associated *weight*. The first list is simply a list of n leaves, with the ith tree in the list having weight p_i. The second list is then developed by merging, in increasing weight order, a copy of the first list with a list of packages produced from the first list. Packages are produced from a list of trees by forming new trees $s_i(t_{2i-1}, t_{2i})$, for each $i \in \{1, 2 \ldots \lfloor m/2 \rfloor\}$, where t_h is the hth tree in the list, and m is the cardinality of the list. The weight of a package, stored in item s_i, is the sum of the weights of its children. In general, list j is developed by forming a list of packages from list $j-1$ and merging this list with a copy of the first list developed. For example, the lists developed by the package-merge method when applied to the input weights $p = [1, 1, 5, 7, 10, 14]$ with $L = 4$ are shown in Figure 1.

Define the *active leaves* to be the leaves of the first $2n - 2$ trees of the Lth list. Extracting the code from the final set of L lists is achieved by processing these active leaves—each active leaf corresponds to exactly one of the original symbols, and l_i should be set to the number of active leaves corresponding to p_i. In the above example $n = 6$, so the first 10 trees in the fourth list contain the active leaves that yield the code, as shown by the shaded region in Figure 1. The first two trees in the bottom list of Figure 1 are active leaves corresponding to p_1 and p_2, thus $l = [1, 1, 0, \ldots, 0]$ after these two trees are processed. The third tree has two leaves, again corresponding to p_1 and p_2, so $l = [2, 2, 0, \ldots, 0]$ after this tree has been expanded. After all 10 trees are processed $l = [4, 4, 3, 2, 2, 2]$, which is a minimum-redundancy 4-limited code. Note that a 3-limited code can also be calculated by processing the active leaves reachable from the first ten items in the third list, yielding $l = [3, 3, 3, 3, 2, 2]$.

It is instructive to examine how $K = \sum_{i=1}^{n} 2^{-l_i}$ changes as the code is extracted from the active leaves. Initially, $l = [0, 0, \ldots, 0]$ and $K = n$. If a tree chosen in list L is a leaf with weight p_i, then l_i increases from 0 to 1, and K reduces by 2^{-1}. If the chosen tree is a package then it will have leaves at various levels. A straightforward inductive argument shows that the reduction in K

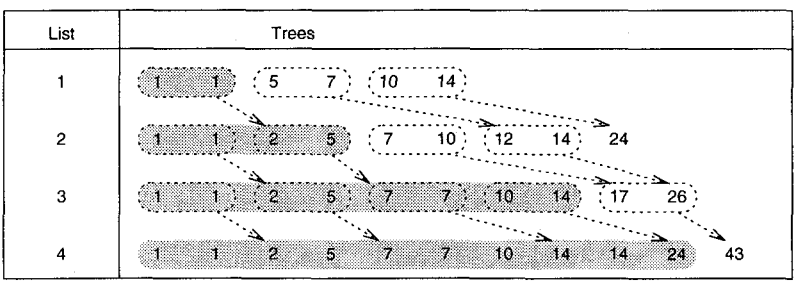

Fig. 1. Package-merge on the input weights $[1, 1, 5, 7, 10, 14]$ with $L = 4$

arising from the leaves of any tree rooted in the jth list is $2^{-(L-j+1)}$, so every package in the Lth list reduces K by 2^{-1}. That is, choosing $2n-2$ trees from the Lth list reduces K from n to 1, guaranteeing that the code generated is a prefix code. Furthermore, the greedy manner in which the trees are constructed and chosen guarantees both that $l_i \leq L$ and that $B = \sum_{i=1}^{n} l_i p_i$ is minimised.

As described here, the package-merge method requires both $O(nL)$ time and $O(nL)$ space. Larmore and Hirschberg reduced the space requirement to $O(n)$ by implementing the method recursively and performing a controlled amount of reevaluation rather than storing intermediate results [3]. Our development, described in the next section, also starts with the space-inefficient package-merge and arrives at a third version that requires $O(L^2)$ auxiliary space.

Finally, note that it is possible to store the solution as a list $a = [a_j \mid j \in \{1 \ldots L\}]$, where a_j is the number of active leaves in the jth list, thereby avoiding the need to store the list l. In the example $a = [2,3,6,6]$ when $L = 4$ and $a = [4,6,6]$ when $L = 3$. If conversion is required, a straightforward $O(n)$-time loop suffices to convert a into the corresponding list l. Boundary package-merge, described in the next section, produces a as its output.

4 Boundary Package-Merge

The package-merge algorithm develops trees in an exhaustive manner; that is, list $j-1$ is completely created before construction of list j is commenced, and all of lists 1 through to $L-1$ are fully instantiated before even the initial tree in list L is determined. The first key observation we make is that trees can also be built as a *demand-driven* or *lazy* process, beginning with the roots of the $2n-2$ trees in the Lth list and adding the necessary children to complete them [6]. The roots must be created in increasing weight order, so for every package that appears in list L the weight of two items in list $L-1$ must have been calculated. This in turn leads to the creation of items in list $L-2$, and so on. Since there are no packages in the first list these cascading demands can always eventually be satisfied.

The basic operation in this demand-driven process is the appending of a tree to some list j, where $1 \leq j \leq L$. Each time this operation is performed it is necessary to determine whether the next item is a leaf or a package. The next leaf has weight p_{c+1}, where all leaves of weight p_k for $1 \leq k \leq c$ have already been included in list j; and the next package is of weight equal to the sum of the next two unused trees in list $j-1$. As in the original package-merge method the candidate with the smaller weight is selected. Hence, when a tree is required in list j two trees must be instantiated in list $j-1$ before the choice between package and leaf can be made. These two trees will be referred to as *lookahead* trees, as they are created to allow knowledge of the weight of the next package in list j, but the package they form may not become part of an active tree until some time in the future. When the two lookahead trees in list $j-1$ are eventually used to form a package in list j two more lookahead trees must be created in list $j-1$ so that the weight of the next package in list j can be determined.

Input	A set of L lists of trees, a list of n symbol weights, and the index j of the list in which an item is required. Suppose that list j currently contains c singleton trees.
Step 1	If $j = 1$ then create a singleton tree d of weight p_{c+1} and append it to list 1. Return.
Step 2	Let t_1 and t_2 be the two lookahead trees in list $j-1$ and s be the sum of the weights of their roots. If s is larger than p_{c+1} then do Step 3a, otherwise do Step 3b.
Step 3a	Create a new leaf d with weight p_{c+1}. Append d to list j.
Step 3b	Create a new tree $d = s(t_1, t_2)$. Append d to list j. Invoke LAZYPM twice with parameter $j-1$ to generate new lookahead trees in list $j-1$.
Output	The same set of L lists, with one extra tree in list j and perhaps extra trees in lists 1 to $j-1$.

Fig. 2. Algorithm LAZYPM

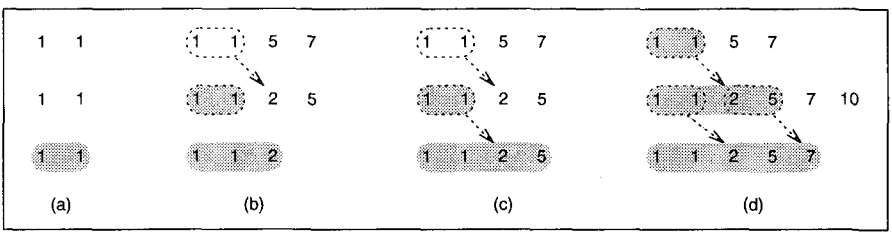

Fig. 3. Demand-driven list development on the weights $[1, 1, 5, 7, 10, 14]$ with $L = 3$ after: (a) two trees have been created; (b) three trees have been created; (c) four trees have been created; and (d) five trees have been created

Algorithm LAZYPM in Figure 2 describes this process, accepting a list number as a parameter, and then generating the next tree in that list using the demand-driven method. When applied $2n-2$ times with parameter L to a set of initialised lists, LAZYPM produces all active trees. Initially, each list contains as lookahead trees two leaves with weight p_1 and p_2—these are the first two items in all of the L lists, since all of the weights are assumed to be positive and the first package in each list has weight $p_1 + p_2$.

Figure 3 shows the lists from the earlier example when evaluated using algorithm LAZYPM with $L = 3$. Figure 3a shows the state of the lists after two trees in L have been processed, with Figures 3b, 3c, and 3d showing the lists after creation of the third, fourth and fifth trees in list L respectively.

The advantage of the demand-driven approach to list development is that all

of the leaves in all of the constructed trees rooted in list L are definitely known to be active, as exactly $2n-2$ trees are created in list L. This means that the leaves of each level L tree can be processed and included into a running solution as soon as that tree is formed, and then the space occupied by that tree freed for reuse. For example, in Figure 3d the first five trees rooted in list L (shown in the shaded region) are known to be active. These trees can be discarded and a set to $[2,3,3]$, where a is the active leaf list described earlier. Unfortunately, the peak space requirement is still $\Theta(nL)$. To see this, consider the case when $p_i = 1$, for all i. For large n and L, two lookahead trees in list $L-1$ correspond to four outstanding trees in list $L-2$; moreover, two more lookahead trees are required. These six trees correspond to 12 trees (plus two more lookahead trees) in list $L-3$, and so on. Hence, for this example $\Theta(nL)$ space can still be required.

By the construction of the lists, however, we know that if the rightmost active leaf in list j has weight p_c, then $a_j = c$, and this is the second key observation that makes the new method possible. This observation suggests that building complete trees is wasteful, as a can be extracted knowing only the rightmost active item at each depth in each tree. Boundary package-merge exploits this observation by manipulating lists of one-dimensional *chains* rather than two-dimensional trees. Each chain corresponds to an equivalent tree of the same weight in the package-merge lists. A node in a chain is a *package node* if its weight is derived from the sum of two weights from the previous list, or a *leaf node* otherwise. A node in list j differs from a tree, however, in that it stores a single pointer into the list $j-1$, and it also stores a count of the number of leaf nodes to its left in list j including itself. Let the pair $\langle w, c\rangle$ define a node, where w is the weight of the node and c is the leaf node count; and let each of the L lists be a list of chains of nodes.

Each chain forms a boundary between the nodes that will become active if the head of the chain becomes active—the nodes that were children of this tree in the two-dimensional package-merge—and those nodes that belong to other trees. Hence, if the head node in any chain becomes active it is known that all other nodes in chain are active. Furthermore, for each node in the chain that becomes active, say $\langle w_j, c_j\rangle$ in list j, it is known that the first c_j leaf nodes in list j have become active, and that any others remain of indeterminate status as lookahead nodes. The chain beginning at the $(2n-2)$nd node in list L gives the result a by setting $a_j = c_j$ for each node $\langle w_j, c_j\rangle$ in the chain. For example, the final boundary chain for the lists in Figure 1 is $\langle 24,6\rangle(\langle 14,6\rangle(\langle 5,3\rangle(\langle 1,2\rangle)))$, giving $a = [2,3,6,6]$.

The revised method is described in Figure 4. All L lists are initialised to contain the first two lookahead chains $\langle p_1, 1\rangle$ and $\langle p_2, 2\rangle$. The need for $2n-2$ chains in list L again drives the process.

At each step there are two candidate chains to be considered for entry into list j. The first candidate is a singleton chain $\langle p_{c+1}, c+1\rangle$, where all leaf nodes of weight p_k for $k \leq c$ have already been added to list j. The second candidate is a package with weight equal to the sum of the weights of the head nodes in the two lookahead chains in list $j-1$, and whose count is equal to the count of

Input	A set of L lists of chains, a list of n symbol weights, and the index j of the list in which an item is required. Suppose that $\langle w, c\rangle(y)$ is the last chain in list j, with weight w, count c, and tail y. That is, there are c singletons in list j.
Step 1	If $j = 1$ then create a singleton chain $\langle p_{c+1}, c+1\rangle$ and append it to list 1. Return.
Step 2	Let s be the sum of the weights of the heads of the two lookahead chains in list $j-1$. If s is larger than p_{c+1} then do Step 3a, otherwise do Step 3b.
Step 3a	Create a new chain $d = \langle p_{c+1}, c+1\rangle(y)$. Append d to list j.
Step 3b	Let z be the second of the two lookahead chains in list $j-1$. Create a new chain $d = \langle s, c\rangle(z)$. Append d to list j. Invoke BOUNDARYPM twice with parameter $j-1$.
Output	The same set of L lists, with one extra chain in list j and perhaps extra chains in lists 1 to $j-1$.

Fig. 4. Algorithm BOUNDARYPM

the head of the chain currently at the end of list j.

Figure 5 shows how Steps 3a and 3b of BOUNDARYPM affect the list structure. Figure 5a shows the input to the algorithm, with the last chain in the Lth list labelled $\langle w, c\rangle(y)$ and the second lookahead chain in list $L-1$ labelled z. Chain $\langle w, c\rangle(y)$ defines the chain of rightmost active nodes (the current solution boundary), so all nodes in and to the left of that chain are definitely in the active set. The next symbol weight that has not been used in list L is p_{c+1}. If $p_{c+1} < s$, where s is the sum of the weights of the nodes at the heads of the two lookahead chains in list $L-1$, then a chain with head $\langle p_{c+1}, c+1\rangle$ and tail y is added to

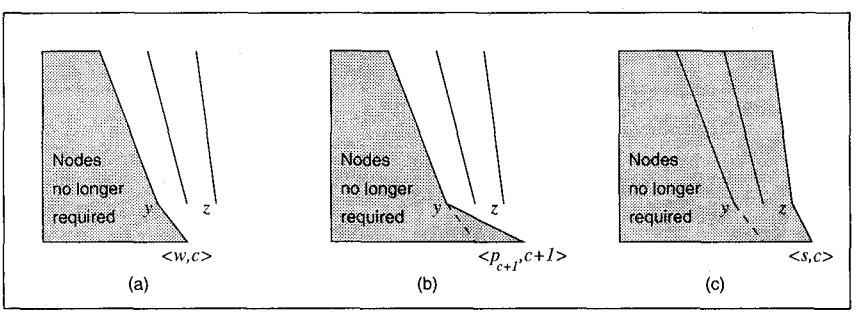

Fig. 5. Addition of a node to list L: (a) initial situation; (b) if Step 3a is used; and (c) if Step 3b is used

list L and the boundary shifts to the position shown in Figure 5b. On the other hand (Figure 5c), if $p_{c+1} \geq s$, then the chain added to list L has tail z and head $\langle s, c \rangle$. In this case no new leaves are added in list L, so the count is unchanged from the node $\langle w, c \rangle$; but all nodes in the chain are confirmed as active, so the tail of the chain is inherited from z, the second lookahead chain in list $L-1$. As more nodes are created in list L the boundaries move further to the right; the final boundary after $2n-2$ nodes have been created in list L defines the desired L-limited prefix code.

5 Analysis

We now examine the computational resources consumed during the execution of algorithm BOUNDARYPM, first considering the space required.

At any given point of time during the execution of the algorithm there are exactly two lookahead chains extant in each of the first $L-1$ lists, and two chains are required in the Lth list—one current chain, and one new chain being constructed. Each extant chain contains just one node in each prior list. Moreover, chains might coalesce as they link through the lists. The total number M of nodes actually required at any given point in time is thus bounded above by $2L + 2(L-1) + \cdots + 2 = 2\sum_{i=1}^{L} i = L(L+1)$.

Suppose that a pool of free nodes of size $2M$ is allocated at the start of the algorithm. Nodes are allocated at Steps 1 and 3 of Figure 4, but are nowhere deallocated. Indeed, it is not at all obvious how explicit deallocation can take place in any efficient manner. Fortunately, deallocation of nodes no longer required can be done economically by periodically performing a *garbage collection* step. Whenever the pool of available nodes is empty, the set of current chains is traversed and all nodes on them marked as "in-use". Then, the complete node pool is inspected sequentially and nodes not currently "in-use" collected onto a free list. Both of these steps require $O(M)$ time, and if the pool contains $2M$ nodes, the combined mark/collect process is guaranteed to release at least M nodes onto the free list. Thus, the amortized cost of each node allocation is $O(1)$, and the total memory required is $2M = O(L^2)$ nodes plus $O(L)$ space for the various list indexing arrays. Therefore, we have

Theorem 1. *The boundary package-merge algorithm described above requires $O(L^2)$ auxiliary space to construct a minimum-redundancy L-limited prefix code for an alphabet of n symbols.*

Let us now consider the running time of BOUNDARYPM. The two lemmas below follow directly from the description of the algorithm:

Lemma 2. *There are at most $2nL$ list items created by boundary package-merge.*

Proof: By induction, showing that each list contains fewer than $2n$ items. The first list contains n items, establishing a basis. Assume that list j has $i < 2n$ items. Then list $j+1$ has $n + \lfloor i/2 \rfloor < 2n$ items. □

Lemma 3. *Each execution of algorithm* BOUNDARYPM *takes* $O(1)$ *time.*

Proof: If a pointer is maintained to the end of each list, nodes can be appended to the list in constant time. This array of pointers consumes $O(L)$ space, and does not impact the space bound of Lemma 1. Comparison, addition and assignment are all $O(1)$ operations by assumption, so Steps 1 and 2 take $O(1)$ time. Provided the pool of nodes is at least twice the size of the peak requirement, the cost of creating a new node is $O(1)$ amortized time, as discussed above. Steps 3a and 3b also require chain manipulations, but these involve a single pointer assignment each, so are also $O(1)$. Finally, note that the recursive calls at Step 3b are not counted, since they result in separate executions of BOUNDARYPM. □

Lemmas 2 and 3 yield

Theorem 4. *The boundary package-merge algorithm described above requires* $O(nL)$ *time to construct a minimum-redundancy* L*-limited prefix code for an alphabet of* n *symbols.*

6 Discussion

The boundary package-merge algorithm has been implemented and tested on the frequency distribution of the approximately 1,000,000 distinct words appearing in a corpus containing three gigabytes of English text. A minimum-redundancy code for this distribution averages 11.521 bits per word and has a maximum codelength of 29 bits; the boundary package merge method calculated an $L = 22$ length-limited code in approximately two minutes on a mid-range workstation using just a few kilobytes of auxiliary memory. In contrast, our implementation of Larmore and Hirschberg's algorithm requires over 60 megabytes of auxiliary memory [6] for that input data. Surprisingly, this heavily restricted code still attains 11.846 bits per symbol.

A number of other methods have been devised recently for calculating length-limited codes. Based upon a problem reduction due to Larmore and Przytycka [4], Schieber [7] has given an $O(n2^{O(\sqrt{\log L \log\log n})})$-time and $O(n)$-space algorithm for this problem. At the time of writing we know of no implementation of this method. Nevertheless, it provides another starting point from which the space-efficiency of the length-limited coding problem can be explored, and may yield algorithms of practical significance.

Moffat *et al.* [6] considered an alternative formulation in which each problem instance is described by a list $[(w_i, f_i) \mid i \in \{1 \ldots r\}]$, where the pair (w_i, f_i) represents f_i repetitions of weight w_i, and $\sum_{i=1}^{r} f_i = n$. In this case $O(r \log(n/r))$ time and space suffices to generate a minimum-redundancy code, and $O(Lr \log(n/r))$ time and space suffices for the generation of a length-limited code.

Acknowledgement

This work was supported by the Australian Research Council.

References

1. T.C. Hu and K.C. Tan. Path length of binary search trees. *SIAM Journal of Applied Mathematics*, 22(2):225–234, March 1972.
2. D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Engineers*, 40(9):1098–1101, September 1952.
3. L.L. Larmore and D.S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473, July 1990.
4. L.L. Larmore and T.M. Przytycka. Constructing Huffman trees in parallel. *SIAM Journal on Computing*. To appear.
5. A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In *Proc. Workshop on Algorithms and Data Structures*, Kingston University, Canada, August 1995. Springer-Verlag. To appear.
6. A. Moffat, A. Turpin, and J. Katajainen. Space-efficient construction of optimal prefix codes. In *Proc. IEEE Data Compression Conference*, pages 192–201, Snowbird, Utah, March 1995. IEEE Computer Society Press, Los Alamitos, California.
7. B. Schieber. Computing a minimum-weight k-link path in graphs with the concave Monge property. In *Proc. 6th Ann. Symp. Discrete Algorithms*, pages 405–411, San Francisco, California, January 1995. SIAM, Philadelphia, Pennsylvania.
8. J. van Leeuwen. On the construction of Huffman trees. In *Proc. 3rd International Colloquium on Automata, Languages, and Programming*, pages 382–410, Edinburgh University, Scotland, July 1976.
9. D.C. Van Voorhis. Constructing codes with bounded codeword lengths. *IEEE Transactions on Information Theory*, IT-20(2):288–290, March 1974.