# In-Place Calculation of Minimum-Redundancy Codes

Alistair Moffat[1] Jyrki Katajainen[2]

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

**Abstract.** The *optimal prefix-free code problem* is to determine, for a given array $p = [p_i \mid i \in \{1 \ldots n\}]$ of $n$ weights, an integer array $l = [l_i \mid i \in \{1 \ldots n\}]$ of $n$ codeword lengths such that $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ and $\sum_{i=1}^{n} p_i l_i$ is minimized. Huffman's famous greedy algorithm solves this problem in $O(n \log n)$ time, if $p$ is unsorted; and can be implemented to execute in $O(n)$ time, if the input array $p$ is sorted. Here we consider the space requirements of the greedy method. We show that if $p$ is sorted then it is possible to calculate the array $l$ in-place, with $l_i$ overwriting $p_i$, in $O(n)$ time and using $O(1)$ additional space. The new implementation leads directly to an $O(n \log n)$-time and $n + O(1)$ words of extra space implementation for the case when $p$ is not sorted. The proposed method is simple to implement and executes quickly.

**Keywords.** Prefix-free code, Huffman code, in-place algorithm, data compression.

## 1 Introduction

The algorithm introduced by Huffman [4, 7] for devising minimum-redundancy prefix-free codes is well known and continues to enjoy widespread use in data compression programs. Huffman's method is also a good illustration of the greedy paradigm of algorithm design and, at the implementation level, provides a useful motivation for the priority queue abstract data type. For these reasons Huffman's algorithm enjoys a prominence enjoyed by only a relatively small number of fundamental methods.

In this paper we examine the space-efficiency of this greedy algorithm for constructing optimal prefix-free codes. Textbooks describing the technique often provide pseudo-code rather than a complete implementation and draw figures showing forests of binary trees. These descriptions create the impression that the implementation of the greedy algorithm should be pointer-based and reliant upon a linear amount of auxiliary memory for node addresses and for internal tree nodes. This is, as we shall show, an erroneous impression. We describe an implementation of the greedy algorithm that, in addition to an input array

storing the weights of the symbols to be coded, requires just $O(1)$ words of extra space if the input array is sorted and $n + O(1)$ words of extra space if the input array is not sorted, where $n$ is the number of symbols for which the code is to be constructed. As for pointer-based implementations, the algorithms require $O(n)$ time for sorted input, and $O(n \log n)$ time for unsorted input. Moreover, implementation of the algorithms is straightforward, and they are suitable for practical use.

The main motivation for this study is our algorithmic curiosity. The best previous implementation of the greedy method for optimal prefix-free coding requires $n + O(1)$ words of extra memory and $O(n \log n)$ time for unsorted input arrays [10], so it was natural to ask whether these bounds could be improved if the input array is sorted. In particular, we were interested to know whether an in-place calculation was possible, since, for practical computation on large alphabets, the space constant is of overriding concern. For example, a typical textbook implementation of the greedy method requires around 20 megabytes of memory to calculate a code for a collection of one million symbols, whereas our implementation requires just 4 megabytes (one million rather than five million 4-byte words). Furthermore, recent research papers report (see, for example, [2]) that in-place algorithms can be faster in practice than their space-inefficient counterparts when run on a modern computer system with a hierarchical memory. Speed is one of the important characteristics of our implementation, too. We have calculated an optimal code for a set of over one million symbols in just a few seconds of CPU time.

## 2 Prefix Codes

Suppose that in some token stream there are $n$ distinct symbols and that the $i$th least frequent symbol appears $p_i$ times. That is, we suppose that $p = [p_i \mid i \in \{1 \ldots n\}]$ is a non-decreasing array of $n$ positive integer *weights*, $p_1 \leq p_2 \leq p_3 \cdots \leq p_n$. A *code* is an array $l = [l_i \mid i \in \{1 \ldots n\}]$ of $n$ integers, where the presumption is that the $i$th symbol is to be represented by an $l_i$-bit long binary *codeword* over the alphabet $\{0, 1\}$. A *prefix-free 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 $l$, it is straightforward to determine a set of $n$ codewords, one per distinct symbol, with the property that the codeword for symbol $i$ is $l_i$ bits long, and such that no codeword in the set is a proper prefix of any other.

An *optimal 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 token stream in question; a code is optimal if there is no other code that results in an output representation requiring fewer than $B$ bits. For any given array $p$ there can be more than one optimal code; for the assignment $p = [1, 1, 2, 2]$ both $l = [2, 2, 2, 2]$ and $l = [3, 3, 1, 2]$ (and one other) result in compressed representations that require $B = 12$ bits. Note, however, that there

is always at least one code for which $l_1 \geq l_2 \geq l_3 \cdots \geq l_n$, and it is such a code that we shall seek to calculate. Huffman's greedy algorithm [4] generates optimal prefix-free codes, and is sketched in Section 3; an alternative paradigm by which this problem may be solved has been articulated recently by Larmore and Przytycka [5].

Once an optimal prefix-free code has been determined and a set of codewords is known they can be used to generate an efficient representation of the token stream. If the original representation was not as economical then compression will result. 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 summarized in Witten, Moffat, and Bell [10] (see also [3]). The decoder is also space-efficient—for the case of sorted symbol weights and ordinal symbol identifiers in $\{1 \ldots n\}$ the decoder requires just $O(\max_{i=1}^{n} l_i)$ words of memory.

We also need some terminology for describing regular binary trees: a *node* $s$ is a *tree*, and if $t_1$ and $t_2$ are two trees then $t = s(t_1, t_2)$ is a tree, with node $s$ as its *root*. If $t = s(t_1, t_2)$ and nodes $s_1$ and $s_2$ are the roots of trees $t_1$ and $t_2$ respectively, then $s$ is the *parent* of nodes $s_1$ and $s_2$ in tree $t$. Similarly, nodes $s_1$ and $s_2$ are the *children* of node $s$. If node $s$ is a singleton and has no children, then it is a *leaf* node of the tree, otherwise it is an *internal* node. The *depth* of any node is one greater than the depth of its parent; the depth of a root is zero. A *forest* is a set of trees.

The model of computation we assume is a unit-cost random access machine, in which values as large as $U$ can be stored in a single word, where $U = \sum_{i=1}^{n} p_i$ is the sum of the input weights. That is, we suppose that addition and comparison operations on integer values in the range $1 \ldots U$ require $O(1)$ time each. At various stages of the algorithm we will store in these memory words partial sums of the input weights (integers in the range $1 \ldots U$), array indices (integers in the range $1 \ldots n$), and codeword lengths (integers in the range $1 \ldots n-1$).

In all of the algorithms that follow we will assume at no cost the $n$ words of storage for the input array $p$; this is the description of the problem and is "free" in the same way that in-place sorting algorithms such as Heapsort regard the input list as being "free". We also suppose that the algorithm may be destructive—that $p_i$ can be overwritten by $l_i$ and that the output array replaces the input array. What we seek to limit is the extra space required. As will be demonstrated in Section 4, $O(1)$ words of memory are sufficient to solve the optimal prefix-code problem as we have stated it here.

## 3 The Greedy Algorithm

Huffman's greedy algorithm [4] is widely known and descriptions appear in a wide range of algorithms textbooks (see, for example, [9]). In this greedy method a forest of trees is manipulated. At each step of the algorithm the two least weight trees are selected and melded, this continuing until a single tree remains. For

the purposes of ordering the sequence of operations, the *weight* of a tree is the sum of the weights of symbols associated with the leaves of that tree, with ties broken arbitrarily. Initially each symbol is installed in a singleton tree, so at the commencement of the algorithm the forest contains $n$ trees.

By the end of the melding process there is one tree remaining. This tree contains as subtrees all of the other trees constructed during the course of the algorithm; and the weight of the final tree is the sum of the set of initial weights, $\sum_{i=1}^{n} p_i$. The structure of the final tree defines an optimal code—symbol $i$ of weight $p_i$ should be allocated a codeword of $l_i$ bits, where $l_i$ is the depth in the tree of the leaf corresponding to the singleton tree $p_i$. To allow depths to be calculated, the structure of meldings during the melding loop is noted using *parent* pointers. A second loop then traces, for each symbol, the sequence of parent pointers through to the root of the final tree. The depth is the required codelength $l_i$.

A straightforward implementation of the greedy algorithm uses approximately $5n$ words—two words at each leaf node to store the initial weight and the parent pointer; $2n - 2$ words to store the weight and parent of the $n - 1$ internal nodes of the final tree; and $n$ words for a heap of at most $n$ items so that the priority queue operations can be performed efficiently. These operations require $O(\log n)$ time each, so $O(n \log n)$ time is sufficient for the main loop, since each iteration of the melding loop involves a constant number of priority queue operations. This is how the algorithm is described in most textbooks.

One small problem is that, as described, the second phase takes $O(\sum_{i=1}^{n} l_i)$ time, which might be as large as $\Theta(n^2)$ and could dominate the time required by the first phase. The solution to this problem is to label internal nodes with their depth the first time they are traversed, thereby short-circuiting subsequent traversals through that node. Since there are exactly $2n - 2$ edges and each edge is traversed once only, this variant of path-compression reduces the time for the depth-calculation phase to $O(n)$. This modification does not change the space complexity of the algorithm, as the weight fields can be used to record depth at both leaf and internal nodes.

Van Leeuwen [6] was apparently the first to note that if the input array $p$ is in sorted order then the running time can be improved to $O(n)$. The reduction is achieved by keeping the leaf nodes distinct from the internal nodes formed during the melding, and maintaining two separate priority queues. The queue implementation can then be a linked list, since the sequence of internal nodes is formed in sorted order, and the input list is already in sorted order. At each melding stage the two items with the smallest weights are within the first two of the unprocessed section of the input list and the first two of the list of internal nodes, so all of the priority queue operations can be effected in $O(1)$ time. Implementation of this idea requires $4n$ words of memory—$2n$ to store the weights and parent pointers of the leaves and $2n - 2$ to store the weights and parent pointers of the internal nodes.

## 4 In-Place Implementation

Let us now focus on the implementation of van Leeuwen's $O(n)$ variant of the greedy algorithm. During the melding phase, two lists are manipulated—a sorted list of leaves that have not yet been processed and a sorted list of internal nodes. The first observation we make about this operation is that the weight of any node need be maintained only until that node is processed. At any given stage of the melding process there are thus at most $n$ weights to be recorded; and by the end of this phase there is just one extant weight.

The second key observation is that it is not necessary to maintain parent pointers in both of these lists. If the depth of each internal node of the tree is known then the depth of each leaf can be inferred, since the codeword lengths can be assumed to be non-increasing. For example, a tree with internal node depths of $[3, 3, 2, 1, 0]$ must have leaves at depths $[4, 4, 4, 4, 2, 1]$. Furthermore, at the start of the melding phase there are no parent pointers in either list; and at the end there are $n - 2$ in the list of internal nodes, but none in the list of leaves. The combined total of weights (required for nodes yet to be processed) and parent pointers (required for internal nodes already processed) can never exceed $n$, so the parent pointers and weights can co-exist in the same array. If $r$ indicates the next tree node to be processed, $s$ indicates the next leaf node to be processed (singleton tree), and $t$ indicates the next vacant position to be used for a tree node, then the array can be partitioned and processed as shown in the following diagram:

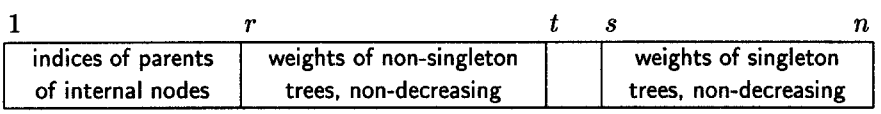



Figure 1 describes this process in more detail. Initially $A[i]$ is assumed to store $p_i$, but the values are modified in-place as the procedure executes. At the completion of the loop, word $A[n]$ is unused, word $A[n-1]$ stores the weight of

1. Set $s \leftarrow 1$ and $r \leftarrow 1$.
2. For $t \leftarrow 1$ to $n - 1$ do
   (a) If $(s > n)$ or $(r < t$ and $A[r] < A[s])$ then
   /* Select an internal tree node */
   Set $A[t] \leftarrow A[r]$, $A[r] \leftarrow t$, and $r \leftarrow r + 1$
   else
   /* Select a singleton leaf node */
   Set $A[t] \leftarrow A[s]$ and $s \leftarrow s + 1$.
   (b) Repeat Step 2a, but adding to $A[t]$ rather than assigning.

**Fig. 1.** In-place processing, phase one

the code tree, and words $A[1 \ldots n-2]$ store parent pointers. Care must be taken that nodes and leaves are only examined if they logically exist, so the test at Step 2a includes a validity guard. Note that both "and" and "or" are assumed to be evaluated conditionally. Note also the strict inequality in the last clause of the test at Step 2a. If ties are broken in favour of leaf nodes then the resulting code has the smallest possible value of $l_1 = \max_{i=1}^{n} l_i$ amongst all minimum redundancy codes [6].

Figure 2a shows an example array of $n = 6$ weights prior to the execution of the procedure of Figure 1. Figure 2b indicates the state of processing at the commencement of Step 2a when $t = 4$ and $A[3]$ has just been computed, at which time $s = 5$ indicating that $A[5]$ is the next leaf to be processed, and $r = 3$, marking $A[3]$ as the next tree node to be considered. The two sets of double lines in Figure 2b indicate the three active zones in the array. Finally, Figure 2c shows the contents of the array at the completion of this first phase.

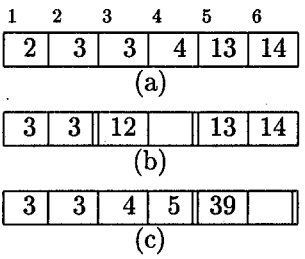
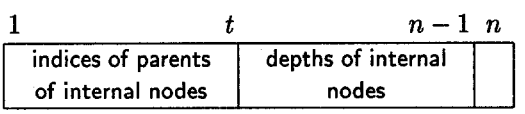



**Fig. 2.** Example of phase one on input array $[2, 3, 3, 4, 13, 14]$

In the second phase of the algorithm the array $A$ must be converted into a array of codelengths. This process is described in Figure 3, and requires two further scans of the array $A$. In the first scan $A$ is converted to an array of depths of internal nodes—Step 2 in Figure 3. The important observation here is that all of the array indices—that is, parent pointers—stored in $A[1 \ldots n-2]$ point to the right, so that $A[i] > i$. Hence, if $A[n-1]$ is assigned tree depth of 0, then a leftward scan in the array setting each depth to be one more than the depth of the parent node correctly converts parent pointers to node depths. By the completion of Step 2, $A[1 \ldots n-1]$ is a list of depths of the internal nodes of the tree. The arrangement in array $A$ during Step 2 in Figure 3 is:




Continuing the previous example, Figure 4a shows the result of applying this step to the array shown in Figure 2c.

1. Set $A[n-1] \leftarrow 0$.
2. For $t \leftarrow n-2$ downto 1 do
   Set $A[t] \leftarrow A[A[t]] + 1$.
3. Set $a \leftarrow 1$, $u \leftarrow 0$, $d \leftarrow 0$, $t \leftarrow n-1$, and $x \leftarrow n$.
4. While $a > 0$ do
   (a) While $t \geq 1$ and $A[t] = d$ do
   Set $u \leftarrow u+1$ and $t \leftarrow t-1$.
   (b) While $a > u$ do
   Set $A[x] \leftarrow d$, $x \leftarrow x-1$, and $a \leftarrow a-1$.
   (c) Set $a \leftarrow 2u$, $d \leftarrow d+1$, and $u \leftarrow 0$.

**Fig. 3.** In-place processing, phase two

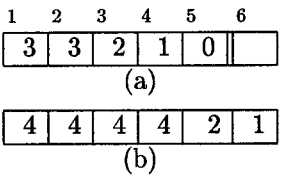
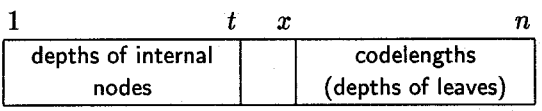



**Fig. 4.** Example of phase two on input array $[2,3,3,4,13,14]$

Finally, the $n-1$ internal node depths must be converted to $n$ leaf node depths. This is accomplished by a further right-to-left scan using pointers $t$, which consumes internal nodes, and $x$, which indicates the index at which the next external node depth should be stored. The arrangement during this phase (Step 4 of Figure 3) is:




The procedure used assumes that the internal node depths in $A[1 \ldots n-1]$ form a non-increasing sequence. That this must be so is demonstrated by the following argument. To disambiguate the two different values stored in array $A$, let *parent*$[i]$ denote the value of $A[i]$ prior to Step 2 of Figure 3 and let *depth*$[i]$ denote the value stored in $A[i]$ after the execution of Step 2. Suppose, in contradiction of the claim that the *depth* values are non-increasing, that *depth*$[i]$ $<$ *depth*$[j]$ for some $1 \leq i < j \leq n-1$. Further, assume that $j$ is the maximum value for which a corresponding $i$ can be found. Note that neither $i$ nor $j$ can be the root: $j$ cannot, since *depth*$[j] >$ *depth*$[i] \geq 0$; and $i$ cannot, since $i < n-1$ and the root is, by definition, in $A[n-1]$.

Consider the two values $i' = parent[i]$ and $j' = parent[j]$. If $i' = j'$ then $depth[i]$ and $depth[j]$ must be the same, since both are calculated as $depth[i'] + 1 = depth[j'] + 1$. Hence, $i' \neq j'$. Moreover, the strict first-in first-out nature of the queue in which internal nodes are stored means that when $i < j$ we have $parent[i] \leq parent[j]$. But, if $parent[i] \neq parent[j]$, then $i' < j'$. Moreover, $depth[i'] = depth[i] - 1$ and $depth[j'] = depth[j] - 1$, by the definition of *depth* used during the calculation at Step 2. But this contradicts the assumption that $j$ was the maximal value for which an $i$ could found, $i < j$ and $depth[i] > depth[j]$, since we have just demonstrated that $i' < j'$ and $depth[i'] > depth[j']$. Thus, no such $i$ and $j$ could have existed in the first instance and the claim is correct—the list of internal node depths is non-increasing.

To perform the conversion from internal node depths to leaf node depths, the number $u$ of internal nodes used at each depth $d$ is counted and subtracted from the total number of nodes (including leaves) available (variable $a$) at that depth of the tree at Step 4a. Any nodes that were available for use at this level but not encountered as internal nodes must be leaf nodes and can be assigned; this is done at Step 4b. Depth $d$ is then incremented and the next level of the tree is processed. The number of available nodes at any given depth is twice the number of internal nodes used at the previous depth; and initially there is one node of depth zero available. Figure 4b shows the state of the example array at the completion of Step 4. This final array is the desired set of codelengths.

To guarantee that Step 4 of Figure 3 is correct, we must be sure that $t < x$ at all times, as otherwise one or more unprocessed values might get overwritten. We show this by demonstrating that at the commencement of each loop iteration at Step 4 we have, as an invariant, that $t = x - a - u$. When $t = n - 1$, $x$, $a$, and $u$ have the values $n$, 1, and 0 respectively and so the claim is true the first time Step 4 is executed. Consider now the effect of Step 4a. Each iteration of the inner loop increases $u$ by one and decreases $t$ by one, maintaining the invariant.

When $t$ either reaches zero or a value at which $A[t] \neq d$ then $x$ is decreased by $a - u$ during the course of the second inner loop at Step 4b, following which Step 4c sets $a$ to twice the value of $u$ and $u$ to zero. If a prime indicates the value of a variable after this sequence of operations, then we have $t' = t$, $x' = x - a + u$, $u' = 0$, and $a' = 2u$. Hence, $t' = x' - a' - u'$ is true if $t = x - a + u - 2 \cdot u - 0$ holds. But the latter expression is true by assumption, so the claim of invariance is correct. Moreover, the variable $u$ is non-negative throughout; and $a$ is positive because of the guard at Step 4. Hence, $t < x$ holds until the loop terminates and the sequence of operations carried out by Step 4 is safe.

## 5 Other Considerations

An actual implementation of the complete algorithm differs only slightly from the pseudo-code shown in Figures 1 and 3 and is remarkably compact. For example, a test implementation in the language C is about 50 lines of code. Three straightforward scans over the input array are required, one in ascending order and two descending, meaning that locality of reference is high. The result is ex-

tremely fast execution. For example, codelengths for an array of 1,073,971 word frequencies (accumulated by processing three gigabytes of English text, see [10] for a description of this document collection) are calculated in just 1.4 seconds of CPU time on a Sun SparcStation 10/402.

If the input array is not sorted, we introduce an $n$-element auxiliary array $B$, initialized so that $B[i] = i$. We next sort $A$, taking care that $B[i]$ continues to record the location in $A$ of weight $p_i$. The in-place Huffman algorithm is then executed on array $A$, and finally the required codelengths are determined by setting $l_i = B[A[i]]$. In this case the running time is dominated by the cost of sorting, and $O(n \log n)$ time is required; the space cost is $n + O(1)$ words of auxiliary storage provided an in-place sorting algorithm such as Heapsort is used, or $n+O(\log n)$ words if a stack-bounded Quicksort (which is usually faster) is used.

If an explicit sort must be performed, sorting is the dominant step. For the same list of 1,073,971 word frequencies it takes around 3.2 seconds for the Bentley-McIlroy Quicksort [1] to sort an array of "frequency, index" pairs, so overall code construction time is 4.6 seconds.[1] By way of comparison, the heap-based construction method described in [10] (which assumes the input is not sorted) requires 23.2 seconds to generate the same codelengths. The difference between the two alternatives—heap-based calculation, and Quicksort then in-place code calculation—is accounted for by the locality of reference exhibited by both Quicksort and the algorithm presented in this paper, and because the Bentley-McIlroy Quicksort exploits duplicate values in the input list, of which, for this data, there are many. Even so, for random integer keys without duplications Quicksort requires just 8.0 seconds to order 1,000,000 two-word records. We thus conclude that the new algorithm is the most effective way to calculate optimal prefix-codes, irrespective of whether or not the data is sorted.

Also worth noting is that although we have assumed throughout that an instance of the optimal prefix-code problem is specified by an $n$-array of symbol weights, other methods for describing problem instances are possible and lead to different time and space requirements. One alternative input formulation suitable for situations in which there many symbols sharing the same weight is a list of pairs $[(p_i, q_i) \mid i \in \{1 \ldots r\}]$, where weight $p_i$ has repetition factor $q_i$, there are $r$ distinct symbol weights, and there are $n = \sum_{i=1}^{r} q_i$ symbols in total. If a similar list of "codelength, repetition count" pairs is the desired output, an optimal prefix-free code can be constructed in $O(r \log(n/r))$ time and space [8], which is $o(n)$ when $r$ is $o(n)$.

It is also interesting to examine the memory requirements of the actual encoding and decoding processes. If we assume—as we have—that tokens are integers in the range $1 \ldots n$ in increasing weight order, then both encoding and decoding can be carried out using two arrays each of $l_1$ words, where $l_1$ is the length of a longest codeword. These arrays are the only space requirement—in particular,

[1] Note, however, that the Bentley-McIlroy Quicksort is not stack-bounded, and in the worst case might require $O(n)$ words of auxiliary memory. Slightly increased times result if a stack-bounded variant is used.

there is no need to maintain an $n$-element array of codewords—and so if $l_1$ is $o(n)$ then the total encoding and decoding space requirement is sublinear. Witten, Moffat, and Bell [10] (see also Hirschberg and Lelewer [3]) describe a mechanism to achieve this. The time required by each encoding or decoding step is linear in the number of output bits, so the total time is $O(\sum_{i=1}^{n} p_i l_i)$. Additional memory is, of course, also required in both encoder and decoder if ordinal symbol numbers in increasing weight order must be mapped from or to actual compression tokens such as characters or words that are not naturally in weight order. The amount of memory required for this mapping and for storage of source tokens depends upon the compression model being used.

## Acknowledgements

We gratefully acknowledge the assistance of Andrew Turpin. We also thank one of the referees, who provided incisive comments that improved our presentation. This work was supported by the Australian Research Council.

## References

1. J.L. Bentley and M.D. McIlroy. Engineering a sorting function. *Software—Practice and Experience* **23** (1993) 1249–1265.
2. S. Carlsson, J. Katajainen, and J. Teuhola. In-place linear probing sort. Submitted. Preliminary version appeared in *Proceedings of the 9th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **577**, Springer-Verlag, Berlin/Heidelberg, Germany (1992) 581–587.
3. D. Hirschberg and D. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM* **33** (1990) 449–459.
4. D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Inst. Radio Engineers* **40** (1952) 1098–1101.
5. L.L. Larmore and T.M. Przytycka. Constructing Huffman trees in parallel. *SIAM Journal on Computing*. To appear.
6. J. van Leeuwen. On the construction of Huffman trees. In *Proceedings of the 3rd International Colloquium on Automata, Languages and Programming*, Edinburgh University Press, Edinburgh, Scotland (1976) 382–410.
7. D.A. Lelewer and D.S. Hirschberg. Data compression. *Computing Surveys* **19** (1987) 261–296.
8. A. Moffat, A. Turpin, and J. Katajainen. Space-efficient construction of optimal prefix codes. *Proceedings of the 5th IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California (1995) 192–201.
9. R. Sedgewick. *Algorithms in C.* 2nd Edition, Addison-Wesley, Reading, Massachusetts (1990).
10. I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, New York, New York (1994).