Distribution-Sensitive Construction of Minimum-Redundancy Prefix Codes

Ahmed Belal and Amr Elmasry

Department of Computer Engineering and Informatics,
Beirut Arab University, Lebanon*
{abelal, elmasry}@alexeng.edu.eg

Abstract. A new method for constructing minimum-redundancy prefix codes is described. This method does not build a Huffman tree; instead it uses a property of optimal codes to find the codeword length of each weight. The running time of the algorithm is shown to be $O(nk)$, where n is the number of weights and k is the number of different codeword lengths. When the given sequence of weights is already sorted, it is shown that the codes can be constructed using $O(\log^{2k-1} n)$ comparisons, which is sub-linear if the value of k is small.

1 Introduction

Minimum-redundancy coding plays an important role in data compression applications [9]. Minimum-redundancy prefix codes give the best possible compression of a finite text when we use one static code for each symbol of the alphabet. This encoding is extensively used in various fields of computer science, such as picture compression, data transmission, etc. Therefore, the methods used for calculating sets of minimum-redundancy prefix codes that correspond to sets of input symbol weights are of great interest [1,4,6].

The minimum-redundancy prefix code problem is to determine, for a given list $W = [w_1, \ldots, w_n]$ of n positive symbol weights, a list $L = [l_1, \ldots, l_n]$ of n corresponding integer codeword lengths such that $\sum_{i=1}^{n} 2^{-l_i} = 1$ (Kraft equality), and $\sum_{i=1}^{n} w_i l_i$ is minimized. Once we have the codeword lengths corresponding to a given list of weights, constructing a corresponding prefix code can be easily done in linear time using standard techniques.

Finding a minimum-redundancy code for $W = [w_1, \ldots, w_n]$ is equivalent to finding a binary tree with minimum-weight external path length $\sum_{i=1}^{n} w(x_i) l(x_i)$ among all binary trees with leaves $x_1, \ldots, x_n$, where $w(x_i) = w_i$ and $l(x_i) = l_i$ is the level of x_i in the corresponding tree. Hence, if we define a leaf as a weighted node, the minimum-redundancy prefix code problem can be defined as the problem of constructing an optimal binary tree for a given list of leaves.

Based on a greedy approach, Huffman algorithm [3] constructs specific optimal trees, which are referred to as Huffman trees. Huffman algorithm starts with a list $\mathcal{H}$ containing n leaves. In the general step, the algorithm selects the two

* The two authors are on sabbatical from Alexandria University of Egypt.

B. Durand and W. Thomas (Eds.): STACS 2006, LNCS 3884, pp. 92–103, 2006.

nodes with the smallest weights in the current list of nodes $\mathcal{H}$ and removes them from the list. Next, the removed nodes become children of a new internal node, which is inserted in $\mathcal{H}$. To this internal node is assigned a weight that is equal to the sum of the weights of its children. The general step repeats until there is only one node in $\mathcal{H}$, the root of the Huffman tree. The internal nodes of a Huffman tree are thereby assigned values throughout the algorithm. The value of an internal node is the sum of the weights of the leaves of its subtree. Huffman algorithm requires $O(n \log n)$ time and linear space. Van Leeuwen [8] showed that the time complexity of Huffman algorithm can be reduced to $O(n)$ if the input list is already sorted.

Throughout the paper, we exchange the use of the terms leaves and weights. When mentioning a node of a tree, we mean that it is either a leaf or an internal node. The levels of the tree that are further from the root are considered higher; the root has level 0. We use the symbol k as the number of different codeword lengths, i.e. k is the number of levels that have leaves in the corresponding tree.

A distribution-sensitive algorithm is an algorithm whose running time relies on how the distribution of the input affects the output [5,7]. In this paper, we give a distribution-sensitive algorithm for constructing minimum-redundancy prefix codes. Our algorithm runs in $O(nk)$, achieving a better bound than the $O(n \log n)$ bound of the other known algorithms when $k = o(\log n)$.

The paper is organized as follows. In the next section, we give a property of optimal trees corresponding to prefix codes, on which our construction algorithm relies. In Section 3, we give the basic algorithm and prove its correctness. We show in Section 4 how to implement the basic algorithm to ensure the distribution-sensitive behavior; the bound on the running time we achieve in this section is exponential with respect to k. In Section 5, we improve our algorithm, using a technique that is similar in flavor to dynamic programming, to achieve the $O(nk)$ bound. We conclude the paper in Section 6.

2 The Exclusion Property

Consider a binary tree T^* that corresponds to a list of n weights $[w_1, \ldots, w_n]$ and has the following properties:

1. The n leaves of T^* correspond to the given n weights.
2. The value of a node equals the sum of the weights of the leaves of its subtree.
3. For every level of T^*, let $\tau_1, \tau_2, \ldots$ be the nodes of that level in non-decreasing order with respect to their values, then τ_{2p-1} and τ_{2p} are siblings for all $p \geq 1$.

We define the *exclusion property* for T^* as follows: T^* has the exclusion property if and only if the values of the nodes at level j are not smaller than the values of the nodes at level $j + 1$.

Lemma 1. *Given a prefix code whose corresponding tree T^* has the aforementioned properties, the given prefix code is optimal and T^* is a Huffman tree if and only if T^* has the exclusion property.*

Proof. First, assume that T^* does not have the exclusion property. It follows that there exists two nodes x and y at levels j_1 and j_2 such that $j_1 < j_2$ and $value(x) < value(y)$. Swapping the subtree of x with the subtree of y results in another tree with a smaller external path length and a different list of levels, implying that the given prefix code is not optimal.

Next, assume that T^* has the exclusion property. Let $[x_1, \ldots, x_n]$ be the list of leaves of T^*, with $w(x_i) \leq w(x_{i+1})$. We prove by induction on the number of leaves n that T^* is an optimal binary tree that corresponds to an optimal prefix code. The base case follows trivially when $n = 2$. As a result of the exclusion property, the two leaves x_1, x_2 must be at the highest level of T^*. Also, Property 3 of T^* implies that these two leaves are siblings. Alternatively, there is an optimal binary tree with leaves $[x_1, \ldots, x_n]$, where the two leaves x_1, x_2 are siblings; a fact that is used to prove the correctness of Huffman's algorithm [3]. Remove x_1, x_2 from T^*, replace their parent with a leaf whose weight equals $x_1 + x_2$, and let T' be the resulting tree. Since T' has the exclusion property, it follows using induction that T' is an optimal tree with respect to its leaves $[x_1 + x_2, x_3, \ldots, x_n]$. Hence, T^* is an optimal tree and corresponds to an optimal prefix code. □

In general, building T^* requires $\Omega(n \log n)$. It is crucial to mention that we do not have to explicitly construct T^*. Instead, we only need to find the values of some of, and not all, the internal nodes at every level.

3 The Main Construction Method

Given a list of weights, we build the tree T^* bottom up. Starting with the highest level, a weight is assigned to a level as long as its value is less than the sum of the two nodes with the smallest values at that level. The Kraft equality is enforced by making sure that the number of nodes at every level is even, and that the number of nodes at the lowest level containing leaves is a power of two.

3.1 Example

For the sake of illustration, consider a list with thirty weights: ten weights have the value 2, ten have the value 3, five the value 5, and five the value 9. To construct the optimal codes, we start by finding the smallest two weights in the list; these will have the values $2, 2$. We now identify all the weights in the list with value less than 4, the sum of these two smallest weights. All these weights will be momentarily placed at the same level. This means that the highest level l will contain ten weights of value 2 and ten of value 3. The number of nodes at this level is even, so we move to the next level $l - 1$. We identify the smallest two nodes at level $l - 1$, amongst the two smallest internal nodes resulting from combining nodes of level l, and the two smallest weights among those remaining in the list; these will be the two internal nodes $4, 4$ whose sum is 8. All the remaining weights with value less than 8 are placed at level $l - 1$. This level now contains an odd number of nodes: ten internal nodes and five weights of value 5. To make this number even, we move the node with the largest weight to the, still empty, next lower level $l-2$. The node to be moved, in this case, is an internal node with value 6. Moving an internal node one level up implies moving the weights in its subtree one level up. In such case, the subtree consisting of the two weights of value 3 is moved one level up. At the end of this stage, the highest level l contains ten weights of value 2 and eight weights of value 3; level $l-1$ contains two weights of value 3 and five weights of value 5. For level $l-2$, the smallest two internal nodes have values $6, 8$ and the smallest weight in the list has value 9. This means that all the five remaining weights in the list will go to level $l-2$. Since we are done with all the weights, we only need to enforce the condition that the number of nodes at level $l-3$ is a power of two. Level $l-2$ now contains eight internal nodes and five weights, for a total of thirteen nodes. All we need to do is to move the three nodes with the largest values, from level $l-2$, one level up. The largest three nodes at level $l-2$ are the three internal nodes of values $12, 12$ and 10. So we move eight weights of value 3 and two weights of value 5 one level up. As a result, the number of nodes at level $l-3$ will be 8. The final distribution of weights will be: the ten weights of value 2 are in the highest level l; level $l-1$ contains the ten weights of value 3 and three weights of value 5; and level $l-2$ contains the remaining weights, two of value 5 and five of value 9. The corresponding code lengths are $6, 5$ and 4 respectively.

3.2 The Basic Algorithm

The idea of the algorithm should be clear. We construct the optimal code tree by maintaining the exclusion property for all the levels. Since this property is always satisfied by the internal nodes, the weights are placed at the levels in such a way that the exclusion property is satisfied. Adjusting the number of nodes at each level will not affect this property since we are always moving the largest nodes one level up to a still empty lower level. A formal description follows.

1. The smallest two weights are found, removed from W, and placed at the highest level l. Their sum S is computed. The list W is scanned and all weights less than S are removed and placed in level l. If the number of leaves at level l is odd, the leaf with the largest weight among these leaves is moved to level $l-1$.
2. In the general iteration, after moving weights from W to level j, determine the weights from W that will go to level $j-1$ as follows. Find the smallest two internal nodes at level $j-1$, and the smallest two leaves from W. Find the smallest two nodes amongst these four nodes, and let their sum be S. Scan W for all weights less than S, and move them to level $j-1$. If the number of nodes at level $j-1$ is odd, move the subtree of the node with the largest value among these nodes to level $j-2$.
3. When W is exhausted, let m be the number of nodes at the shallowest level that has leaves. Move the $2^{\lceil \log_2 m \rceil} - m$ subtrees of the nodes with the largest values, from such level, one level up.

3.3 Proof of Correctness

To guarantee its optimality following Lemma 1, we need to show that both the Kraft equality and the exclusion property hold for the constructed tree.

By construction, the number of nodes at every level of the tree is even. At Step 3 of the algorithm, if m is a power of 2, no subtrees are moved to the next level and Kraft equality holds. If m is not a power of two, we move $2^{\lceil \log_2 m \rceil} - m$ nodes to the next level, leaving $2m - 2^{\lceil \log_2 m \rceil}$ nodes at this level other than those of the subtrees that have just been moved one level up. Now, the number of nodes at the next lower level is $m - 2^{\lceil \log_2 m \rceil - 1}$ internal nodes resulting from the higher level, plus the $2^{\lceil \log_2 m \rceil} - m$ nodes that we have just moved. This sums up to $2^{\lceil \log_2 m \rceil - 1}$ nodes, that is a power of 2, and Kraft equality holds.

Throughout the algorithm, we maintain the exclusion property by making sure that the sum of the two nodes with the smallest values is larger than all the values of the nodes at this level. When we move a subtree one level up, the root of this subtree is the node with the largest value at its level. Hence, all the nodes of this subtree at a certain level will have the largest values among the nodes of this level. Moving these nodes one level up will not destroy the exclusion property. We conclude that the resulting tree has the exclusion property.

4 Distribution-Sensitive Construction

Up to this point, we have not shown how to evaluate the internal nodes needed by our basic algorithm, and how to search within the list W to decide which weights are at which levels. The basic intuition behind the novelty of our approach is that it does not require evaluating all the internal nodes of the tree corresponding to the prefix code, and would thus surpass the $\Theta(n \log n)$ bound for several cases, a fact that will be asserted in the analysis. We show next how to implement the basic algorithm in a distribution-sensitive behavior.

4.1 Example

The basic idea is clarified through an example having $1.5n + 2$ weights. Assume that the resulting optimal tree will turn out to have n leaves at the highest level, $n/2$ at the following level, and two leaves at level 2; the $1.5n$ leaves, at the highest two levels, combine to produce two internal nodes at level 2.

In such case, we show how to produce the codeword lengths in linear time. For our basic algorithm, we need to evaluate the smallest node x of the two internal nodes at level 2, which amounts to identifying the smallest $n/2$ nodes amongst the nodes at the second highest level. In order to be able to achieve this in linear time, we need to do it without having to evaluate all $n/2$ internal nodes resulting from the pair-wise combinations of the highest level n weights. We show that this can be done through a simple pruning procedure. The nodes at the second highest level consist of two sets; one set has $n/2$ leaves whose weights are known and thus their median M can be found in linear time [2], and another set containing $n/2$ internal nodes which are not known but whose median M'

can still be computed in linear time, by simply finding the two middle weights of the highest level n leaves and adding them. Assuming without loss of generality that $M > M'$, then the bigger half of the $n/2$ weights at the second highest level can be safely discarded as not contributing to x, and the smaller half of the highest level n weights are guaranteed to contribute to x. The above step is repeated recursively on a problem half the size. This results in a procedure satisfying the recurrence $T(n) = T(n/2) + O(n)$, and hence $T(n) = O(n)$.

If the list of weights is already sorted, the number of comparisons required to find any of the medians M or M' is constant. This results in a procedure satisfying the recurrence $T_s(n) = T_s(n/2) + O(1)$, and hence $T_s(n) = O(\log n)$.

4.2 The Detailed Algorithm

Let $l_1 > l_2 > \dots l_{k'}$ be the levels that have already been assigned weights at some step of our algorithm (other levels only have internal nodes), n_j be the count of the weights assigned to level l_j, and $\mu_j = \sum_{i=1}^{j} n_i$. At this point, we are looking forward to find the next level $l_{k'+1} < l_{k'}$ that will be assigned weights by our algorithm. Knowing that the weights that have already been assigned to higher levels are the only weights that may contribute to the internal nodes of any level $l \geq l_{k'+1}$, we need to evaluate some internal nodes at these levels.

Finding the splitting node. Consider the internal node x at a level l, $l_{k'} > l \geq l_{k'+1}$, where the count of the weights contributing to the internal nodes of level l, whose values are smaller (larger) than that of x, is at most $\mu_{k'}/2$. We call x the splitting node of l.

The following recursive procedure is used to evaluate x. We find the leaf with the median weight M among the list of the $n_{k'}$ weights already assigned to level $l_{k'}$ (partition the $n_{k'}$ list into two sublists around M), and recursively evaluate the splitting node M' at level $l_{k'}$ using the list of the $\mu_{k'-1}$ weights of the higher levels (partition the $\mu_{k'-1}$ list into two sublists around M'). Comparing M to M', we either conclude that one of the four sublists - the two sublists of the $n_{k'}$ list and the two sublists of the $\mu_{k'-1}$ list - will not contribute to x, or that one of these four sublists contributes to x. If one of the sublists of the $n_{k'}$ list is discarded, find a new median M for the other sublist and compare it with M'. If one of the sublists of the $\mu_{k'-1}$ list is discarded, recursively find the new splitting node M' corresponding to the other sublist and compare it to M. Once one of the two lists becomes empty, we would have identified the weights that contribute to x and hence evaluated x. As a byproduct, we also know which weights contribute to the internal nodes at level l whose values are smaller (larger) than that of x.

Let $T(\mu_{k'}, k')$ be the time required by the above procedure. The total amount of work, in all the recursive calls, required to find the medians among the $n_{k'}$ weights assigned to level k' is $O(n_{k'})$. The time for the i-th recursive call to find a splitting node at level k' is $T(\mu_{k'-1}/2^{i-1}, k'-1)$. The next relations follow:

$$T(\mu_1, 1) = O(n_1),$$
$$T(\mu_{k'}, k') \leq \sum_{i \geq 1} T(\mu_{k'-1}/2^{i-1}, k'-1) + O(n_{k'}).$$

Substitute with $T(a,b) = c \cdot 2^b a$, for $a < \mu_{k'}$, $b < k'$, and some big enough constant c. Then, $T(\mu_{k'}, k') \leq c \cdot 2^{k'-1} \sum_{i \geq 1} \mu_{k'-1}/2^{i-1} + O(n_{k'}) < c \cdot 2^{k'} \mu_{k'-1} + c \cdot n_{k'}$. Since $\mu_{k'} = \mu_{k'-1} + n_{k'}$, it follows that

$$T(\mu_{k'}, k') = O(2^{k'} \mu_{k'}).$$

Consider the case when the list of weights W is already sorted. Let $T_s(\mu_{k'}, k')$ be the number of comparisons required by the above procedure. The number of comparisons, in all recursive calls, required to find the medians among the $n_{k'}$ weights assigned to level k', is at most $\log_2 (n_{k'} + 1)$. The next relations follow:

$$\begin{aligned} T_s(\mu_2, 2) &\leq 2 \log_2 \mu_2, \\ T_s(\mu_{k'}, k') &\leq \sum_{i \geq 1} T_s(\mu_{k'-1}/2^{i-1}, k'-1) + \log_2 (n_{k'} + 1). \end{aligned}$$

Since the number of internal nodes at level k' is at most $\mu_{k'-1}/2$, the number of recursive calls at this level is at most $\log_2 \mu_{k'-1}$. It follows that $T_s(\mu_{k'}, k') \leq \log_2 \mu_{k'-1} \cdot T_s(\mu_{k'-1}, k'-1) + \log_2 (n_{k'} + 1) < \log_2 \mu_{k'} \cdot T_s(\mu_{k'}, k'-1) + \log_2 \mu_{k'}$. Substitute with $T_s(a,b) \leq \log_2^{b-1} a + \sum_{i=1}^{b-1} \log_2^i a$, for $a < \mu_{k'}$, $b < k'$. Then, $T_s(\mu_{k'}, k') < \log_2 \mu_{k'} \cdot \log_2^{k'-2} \mu_{k'} + \log_2 \mu_{k'} \cdot \sum_{i=1}^{k'-2} \log_2^i \mu_{k'} + \log_2 \mu_{k'} = \log_2^{k'-1} \mu_{k'} + \sum_{i=1}^{k'-1} \log_2^i \mu_{k'}$. It follows that

$$T_s(\mu_{k'}, k') = O(\log^{k'-1} \mu_{k'}).$$

Finding the t-th smallest (largest) node. Consider the node x at level $l_{k'}$, which has the t-th smallest (largest) value among the nodes at level $l_{k'}$. The following recursive procedure is used to evaluate x.

As for the case of finding the splitting node, we find the leaf with the median weight M among the list of the $n_{k'}$ weights already assigned to level $l_{k'}$, and evaluate the splitting node M' at level $l_{k'}$ (applying the above recursive procedure) using the list of the $\mu_{k'-1}$ leaves of the higher levels. As with the above procedure, comparing M to M', we conclude that either one of the four sublists - the two sublists of $n_{k'}$ leaves and the two sublists of $\mu_{k'-1}$ leaves - will not contribute to x, or that one of these four sublists contributes to x. Applying the aforementioned pruning procedure, we identify the weights that contribute to x and hence evaluate x. As a byproduct, we also know which weights contribute to the nodes at level $l_{k'}$ whose values are smaller (larger) than that of x.

Let $T'(\mu_{k'}, k')$ be the time required by the above procedure. Then,

$$T'(\mu_{k'}, k') \leq \sum_{i \geq 1} T(\mu_{k'-1}/2^{i-1}, k'-1) + O(n_{k'}) = O(2^{k'} \mu_{k'}).$$

Let $T'_s(\mu_{k'}, k')$ be the number of comparisons required by the above procedure, when the list of weights W is already sorted. Then,

$$T'_s(\mu_{k'}, k') \leq \sum_{i \geq 1} T_s(\mu_{k'-1}/2^{i-1}, k'-1) + O(\log n_{k'}) = O(\log^{k'-1} \mu_{k'}).$$

Finding $l_{k'+1}$, the next level that will be assigned weights. Consider level $l_{k'} - 1$, which is the next level lower than level $l_{k'}$. We start by finding the minimum weight w among the weights remaining in W at this point of the algorithm, and use this weight to search within the internal nodes at level $l_{k'} - 1$ in a manner similar to binary search. The basic idea is to find the maximum number of the internal nodes at level $l_{k'} - 1$ with the smallest values, such that the sum of their values is less than w. We find the splitting node x at level $l_{k'} - 1$, and evaluate the sum of the weights contributing to the internal nodes, at that level, whose values are smaller than that of x. Comparing this sum with w, we decide which sublists of the $\mu_{k'}$ leaves to proceed to find its splitting node. At the end of this searching procedure, we would have identified the weights contributing to the r smallest internal nodes at level $l_{k'} - 1$, such that the sum of their values is less than w and r is maximum. We conclude by setting $l_{k'+1}$ to be equal to $l_{k'} - \lceil \log_2 (r+1) \rceil$.

To prove the correctness of this procedure, consider any level l, such that $r > 1$ and $l_{k'} - \lceil \log_2 (r+1) \rceil < l < l_{k'}$. The values of the two smallest internal nodes at level l are contributed to by at most $2^{l_{k'} - l} \leq 2^{\lceil \log_2 (r+1) \rceil - 1} \leq t$ internal nodes from level $l_{k'} - 1$. Hence, the sum of these two values is less than w. For the exclusion property to hold, no weights are assigned to any of these levels. On the contrary, the values of the two smallest internal nodes at level $l_{k'} - \lceil \log_2 (r+1) \rceil$ are contributed to by more than r internal nodes from level $l_{k'} - 1$, and hence the sum of these two values is more than w. For the exclusion property to hold, at least the weight w is assigned to this level.

The time required by this procedure is the $O(n - \mu_{k'})$ time to find the weight w among the weights remaining in W, plus the time for the calls to find the splitting nodes. Let $T''(\mu_{k'}, k')$ be the time required by this procedure. Then,

$$T''(\mu_{k'}, k') \leq \sum_{i \geq 1} T(\mu_{k'}/2^{i-1}, k') + O(n - \mu_{k'}) = O(2^{k'} \mu_{k'} + n).$$

Let $T''_s(\mu_{k'}, k')$ be the number of comparisons required by the above procedure, when the list of weights W is already sorted. Then,

$$T''_s(\mu_{k'}, k') \leq \sum_{i \geq 1} T_s(\mu_{k'}/2^{i-1}, k') + O(1) = O(\log^{k'} \mu_{k'}).$$

Maintaining Kraft equality. After deciding the value of $l_{k'+1}$, we need to maintain Kraft equality in order to produce a binary tree corresponding to the optimal prefix code. This is accomplished by moving the subtrees of the t nodes with the largest values from level $l_{k'}$ one level up. Let m be the number of nodes currently at level $l_{k'}$, then the number of the nodes to be moved up t is $2^{l_{k'} - l_{k'+1}} \lceil m / 2^{l_{k'} - l_{k'+1}} \rceil - m$. Note that when $l_{k'+1} = l_{k'} - 1$ (as in the case of our basic algorithm), then t equals one if m is odd and zero otherwise.

To establish the correctness of this procedure, we need to show that both the Kraft equality and the exclusion property hold. For a realizable construction, the number of nodes at level $l_{k'}$ has to be even, and if $l_{k'+1} \neq l_{k'} - 1$, the number of nodes at level $l_{k'}-1$ has to divide $2^{l_{k'}-l_{k'+1}-1}$. If m divides $2^{l_{k'}-l_{k'+1}}$, no subtrees are moved to level $l_{k'}-1$ and Kraft equality holds. If m does not divide $2^{l_{k'}-l_{k'+1}}$, then $2^{l_{k'}-l_{k'+1}}\lceil m/2^{l_{k'}-l_{k'+1}}\rceil - m$ nodes are moved to level $l_{k'}-1$, leaving $2m-2^{l_{k'}-l_{k'+1}}\lceil m/2^{l_{k'}-l_{k'+1}}\rceil$ nodes at level $l_{k'}$ other than those of the subtrees that have just been moved one level up. Now, the number of nodes at level $l_{k'}-1$ is $m-2^{l_{k'}-l_{k'+1}-1}\lceil m/2^{l_{k'}-l_{k'+1}}\rceil$ internal nodes resulting from the nodes of level $l_{k'}$, plus the $2^{l_{k'}-l_{k'+1}}\lceil m/2^{l_{k'}-l_{k'+1}}\rceil - m$ nodes that we have just moved. This sums up to $2^{l_{k'}-l_{k'+1}-1}\lceil m/2^{l_{k'}-l_{k'+1}}\rceil$ nodes, which divides $2^{l_{k'}-l_{k'+1}-1}$, and Kraft equality holds. The exclusion property holds following the same argument mentioned in the proof of the correctness of the basic algorithm.

The time required by this procedure is basically the time needed to find the weights contributing to the t nodes with the largest values at level $l_{k'}$, which is $O(2^{k'}\mu_{k'})$. If W is sorted, the required number of comparisons is $O(\log^{k'-1}\mu_{k'})$.

Summary of the algorithm

1. The smallest two weights are found, moved from W to the highest level l_1, and their sum S is computed. The rest of W is searched for weights less than S, which are moved to level l_1.
2. In the general iteration of the algorithm, after assigning weights to k' levels, perform the following steps:
   (a) Find $l_{k'+1}$, the next level that will be assigned weights.
   (b) Maintain the Kraft equality at level $l_{k'}$ by moving the t subtrees with the largest values from this level one level up.
   (c) Find the values of the smallest two internal nodes at level $l_{k'+1}$, and the smallest two weights from those remaining in W. Find the two nodes with the smallest values among these four, and let their sum be S.
   (d) Search the rest of W, and move the weights less than S to level $l_{k'+1}$.
3. When W is exhausted, maintain Kraft equality at the last level that has been assigned weights.

4.3 Complexity Analysis

Using the bounds deduced for the described steps of the algorithm, we conclude that the time required by the general iteration is $O(2^{k'}\mu_{k'}+n)$. If W is sorted, the required number of comparisons is $O(\log^{k'}\mu_{k'})$.

To complete the analysis, we need to show the effect of maintaining the Kraft equality on the complexity of the algorithm. Consider the scenario when, as a result of moving subtrees one level up, all the weights at a level move up to the next level that already had other weights. As a result, the number of levels that contain leaves decreases. It is possible that within a single iteration the number of such levels decreases by one half. If this happens for several iterations, the amount of work done by the algorithm would have been significantly large compared to the actual number of distinct codeword lengths k. Fortunately, this scenario will not happen quite often. In the next lemma, we bound the number of iterations performed by the algorithm by $2k$. We also show that, at any iteration, the number of levels that contain leaves is at most twice the number of distinct optimal codeword lengths for the weights that have been assigned so far.

Lemma 2. *Consider the set of weights, all having the j-th largest optimal codeword length. During the execution of the algorithm, such set of weights will be assigned to at most two consecutive levels, among those levels that contain leaves. Hence, these two levels will be the at most $2j-1$ and $2j$ highest such levels.*

Proof. Consider a set of weights that will turn out to have the same codeword length. During the execution of the algorithm, assume that some of these weights are assigned to three levels. Let $l_i > l_{i+1} > l_{i+2}$ be such levels. It follows that $l_i - 1 > l_{i+2}$. Since we are maintaining the exclusion property throughout the algorithm, there will exist some internal nodes at level $l_i - 1$ whose values are strictly smaller than the values of the weights at level l_{i+2} (some may have the same value as the smallest weight at level l_{i+2}). The only way for all these weights to catch each other at the same level of the tree would be as a result of moving subtrees up (starting from level l_{i+2} upwards) to maintain the Kraft equality. Suppose that, at some point of the algorithm, the weights that are currently at level l_i are moved up to catch the weights of level l_{i+2}. It follows that the internal nodes that are currently at level $l_i - 1$ will accordingly move to the next lower level of the moved weights. As a result, the exclusion property will not hold; a fact that contradicts the behavior of our algorithm. It follows that such set of weights will never catch each other at the same level of the tree; a contradiction.

We prove the second part of the lemma by induction. The base case follows easily for $j = 1$. Assume that the argument is true for $j - 1$. By induction, the levels of the weights that have the $(j-1)$-th largest optimal codeword length will be the at most $2j-3$ and $2j-2$ highest such levels. From the exclusion property, it follows that the weights that have the j-th largest optimal codeword length must be at the next lower levels. Using the first part of the lemma, the number of such levels is at most two. It follows that these weights are assigned to the at most $2j-1$ and $2j$ highest levels. □

Using Lemma 2, the time required by our algorithm to assign the set of weights whose optimal codeword length is the j-th largest, among all distinct lengths, is $O(2^{2j}n) = O(4^j n)$. Summing for all such lengths, the total time required by our algorithm is $\sum_{j=1}^{k} O(4^j n) = O(4^k n)$.

Consider the case when the list of weights W is already sorted. The only step left to mention, for achieving the claimed bounds, is how to find the weights of W smaller than the sum of the values of the smallest two nodes at level l_j. Once we get this sum, we apply an exponential search that is followed by a binary search on the weights of W for an $O(\log n_j)$ comparisons. Using Lemma 2, the number of comparisons performed by our algorithm to assign the weights whose codeword length is the j-th largest, among all distinct lengths, is $O(\log^{2j-1} n)$. Summing for all such lengths, the number of comparisons performed by our algorithm is $\sum_{j=1}^{k} O(\log^{2j-1} n) = O(\log^{2k-1} n)$. The next theorem follows.

Theorem 1. *If the list of weights is sorted, constructing minimum-redundancy prefix codes can be done using $O(\log^{2k-1} n)$ comparisons.*

Corollary 1. *For $k < c \cdot \log n / \log\log n$, and any constant $c < 0.5$, the above algorithm requires $o(n)$ comparisons.*

5 The Improved Algorithm

The drawback of the algorithm we described in the previous section is that it uses many recursive median-finding calls. The basic idea we use here is to incrementally process the weights throughout the algorithm by partitioning them into unsorted blocks, such that the weights of one block are smaller or equal to the smallest weight of the succeeding block. The time required during the recursive calls becomes smaller when handling these shorter blocks. The details follow.

The invariant we maintain is that during the execution of the general iteration of the algorithm, after assigning weights to k' levels, the weights that have already been assigned to a level $l_j \geq l_{k'}$ are partitioned into blocks each of size at most $n_j/2^{k'-j}$ weights, such that the weights of one block are smaller or equal to the smallest weight of the succeeding block. To accomplish this invariant, once we assign weights to a level, the median of the weights of each block among those already assigned to all the higher levels is found, and each of these blocks is partitioned into two blocks around this median weight. Using Lemma 2, the number of iterations performed by the algorithm is at most $2k$. The amount of work required for this partitioning is $O(n)$ for each of these iterations, for a total of $O(nk)$ time for this partitioning phase.

The basic step for all our procedures is to find the median weight among the weights already assigned to a level l_j. This step can now be done faster. To find such median weight, we can identify the block that has such median in constant time, then we find the required weight in $O(n_j/2^{k'-j})$ time, which is the size of the block at this level. The recursive relations for all our procedures performed at each of the k general iterations of the algorithm can be written as

$$G^{k'}(\mu_1, 1) = O(n_1/2^{k'-1}),$$
$$G^{k'}(\mu_{k'}, k') \leq \sum_{i \geq 1} G^{k'}(\mu_{k'-1}/2^{i-1}, k'-1) + O(n_{k'}).$$

Substitute with $G^{k'}(a, b) = c \cdot a/2^{k'-b}$, for $a < \mu_{k'}$, $b < k'$, and some big enough constant c. Then, $G^{k'}(\mu_{k'}, k') \leq c/2 \cdot \sum_{i \geq 1} \mu_{k'-1}/2^{i-1} + O(n_{k'}) < c \cdot \mu_{k'-1} + c \cdot n_{k'}$. Since $\mu_{k'} = \mu_{k'-1} + n_{k'}$, it follows that

$$G^{k'}(\mu_{k'}, k') = O(\mu_{k'}) = O(n).$$

Since the number of iterations performed by the algorithm is at most $2k$, by Lemma 2. Summing up for these iterations, the running time for performing the recursive calls is $O(nk)$ as well. The next main theorem follows.

Theorem 2. *Constructing minimum-redundancy prefix codes is done in $O(nk)$.*

6 Conclusion

We gave a distribution-sensitive algorithm for constructing minimum-redundancy prefix codes, whose running time is a function of both n and k. For small values of k, this algorithm asymptotically improves over other known algorithms that require $O(n \log n)$; it is quite interesting to know that the construction of optimal codes can be done in linear time when k turns out to be a constant. For small values of k, if the sequence of weights is already sorted, the number of comparisons performed by our algorithm is asymptotically better than other known algorithms that require $O(n)$ comparisons; it is also interesting to know that the number of comparisons required for the construction of optimal codes is poly-logarithmic when k turns out to be a constant.

Two open issues remain; first is the possibility of improving the algorithm to achieve an $O(n \log k)$ bound, and second is to make the algorithm faster in practice by avoiding so many recursive calls to a median-finding algorithm.

References

1. M. Buro. *On the maximum length of Huffman codes.* Information Processing Letters 45 (1993), 219-223.
2. T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to algorithms,* 2nd Edition. The MIT press (2001).
3. D. Huffman. *A method for the construction of minimum-redundancy codes.* Proc. IRE 40 (1952), 1098-1101.
4. R. Milidiu, A. Pessoa, and E. Laber. *Three space-economical algorithms for calculating minimum-redundancy prefix codes.* IEEE Transactions on Information Theory 47(6) (2001), 2185-2198.
5. J.I. Munro and P. Spira. *Sorting and searching in multisets.* SIAM Journal on Computing 5(1) (1976), 1-8.
6. A. Moffat and A. Turbin. *Efficient construction of minimum-redundancy codes for large alphabets.* IEEE Transactions on Information Theory 44 (1998), 1650-1657.
7. S. Sen and N. Gupta. *Distribution-sensitive algorithms.* Nordic Journal of Computing 6(2) (1999), 194-211.
8. J. Van Leeuwen. *On the construction of Huffman trees.* 3rd International Colloquium for Automata, Languages and Programming (1976), 382-410.
9. J. Zobel and A. Moffat. *Adding compression to a full-text retrieval system.* Software: Practice and Experience 25(8) (1995), 891-903.