Verification of Minimum-Redundancy Prefix Codes

Ahmed Belal and Amr Elmasry

***Abstract*—We show that verifying a given prefix code for optimality requires $\Omega(n \log n)$ time, indicating that the verification problem is not asymptotically easier than the construction problem. Alternatively, we give linear-time verification algorithms for several special cases that are either typical in practice or theoretically interesting.**

***Index Terms*—Asymptotic complexity, Huffman codes, lower bounds, optimal codes, verification algorithms.**

I. Introduction

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 $\sum_{i=1}^{n} 2^{-l_i} \leq 1$, and $\sum_{i=1}^{n} w_i l_i$ is minimized. Minimum-redundancy coding plays an important role in data compression applications [1], [2]. Therefore, the methods used for calculating sets of minimum-redundancy prefix codes that correspond to sets of input symbol weights are of great interest.

The problem of 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. This equivalence is due to the fact that every prefix code can be represented as a binary 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. Optimal trees are not unique; for example, every minimum-redundancy code can be easily rearranged into a canonical code with the same compression efficiency.

Based on a greedy approach, the well-known Huffman algorithm [3] constructs specific optimal trees, which are referred to as Huffman trees. Huffman algorithm starts with a list $\mathcal{S}$ containing n leaves. In the general step, the algorithm selects the two nodes with the smallest weights in the current list of nodes $\mathcal{S}$ and removes them from the list. Next, the removed nodes become children of a new internal node, which is inserted in $\mathcal{S}$. 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 $\mathcal{S}$, the root of the Huffman tree. The internal nodes of a Huffman tree are thereby assigned weights throughout the algorithm. The weight 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 [4] showed that the time complexity of Huffman algorithm can be reduced to $O(n)$ if the input list is already sorted.

Given a list $L = [l_1, \ldots, l_n]$ of n integer codeword lengths corresponding to the list of positive symbol weights $W = [w_1, \ldots, w_n]$, we investigate the problem of verifying whether this gives a minimum-redundancy prefix code or not. Inspired by the fact that the verification of the constructions of some well-known problems is asymptotically easier than producing the solution, we are interested in checking whether this is true for the minimum-redundancy prefix code problem. For example, the best known bound for the construction of a minimum spanning tree (MST) for a given graph is super-linear [5], while the verification of such a tree can be done in linear time [6]. The fact that the linear-time MST verification algorithm is used in the only-known linear-time randomized algorithm for the construction problem was another motivation [7]. In addition, there is no known polynomial-time algorithm to solve any of the NP-hard problems, yet known polynomial-time verification algorithms exist. Hence, the problem of verifying the optimality of minimum-redundancy prefix codes is of both theoretical and practical importance.

In this paper, we show that the verification of minimum-redundancy prefix codes requires $\Omega(n \log n)$ comparisons in the algebraic decision-tree model, implying that it is asymptotically as hard as constructing such codes. However, we give linear-time verification algorithms for several special cases.

Buro [8] proved that no codeword of a minimum-redundancy prefix code is longer than $\log_\Phi \frac{\Phi+1}{P_1\Phi+P_2}$, where P_1 and P_2 ($P_1 \leq P_2$) are the probabilities of the two least probable symbols and Φ is the golden ratio $\frac{1+\sqrt{5}}{2}$. Consider a message with r symbol appearances, such that n of them are distinct. Since the probability of one symbol cannot be smaller than $1/r$ in a source message of length r, it is immediate that the codeword lengths are bounded from above by $\log_\Phi r$. Note that in many practical cases, when we are working with large alphabets, $\log_\Phi r$ is considerably smaller than n. For example, Moffat and Turbin [9] used a word-based model to compress the 3-Gbyte TREC document collection, finding 480 911 085 word appearances and 1 073 971 distinct words. In this case, the length of the longest codeword cannot be greater than 41. In Section III, Cases 2 and 3, we give linear-time verification algorithms when the leaves are on a small number of levels, a situation that is typical in practice.

The paper is organized as follows. In Section II, we give a property of prefix codes that we call the *exclusion property*. We show that verifying this property is equivalent to verifying the optimality of the given code. In Section III, a lower bound situation is constructed. We show that verifying the exclusion prop-

Manuscript received September 23, 2004; revised June 19, 2005.

The authors are with the Department of Computer Engineering and Systems, Alexandria University, Alexandria, Egypt (e-mail: abelal, elmasry@alexeng.edu.eg).

Communicated by A. E. Ashikhmin, Associate Editor for Coding Theory.
Digital Object Identifier 10.1109/TIT.2006.871578

erty in such situation requires $\Omega(n \log n)$ in the algebraic decision tree model, implying such a lower bound for verifying the optimality of prefix codes. Finally, in Section IV, we give linear-time algorithms to verify the exclusion property for some special cases.

II. The Exclusion Property

Given a list of positive values $W = [w_1, \ldots, w_n]$ and a list of positive integers $L = [l_1, \ldots, l_n]$, respectively, representing the weights and levels of the leaves of a binary tree corresponding to a prefix code, we prove in Lemma 2 that verifying the optimality of this code is equivalent to verifying a property, that we call the exclusion property, of a tree which we construct from W and L. We use Lemma 1 (which is also in [10]) in the proof of Lemma 2. For completeness, we give next the proof of Lemma 1.

Lemma 1: Consider a list of leaves $[x_1, \ldots, x_n]$, with $w(x_i) \leq w(x_{i+1})$. Given a positive integer k such that $w(x_1) + w(x_2) \geq w(x_{2k})$, there is an optimal binary tree with leaves $x_1, \ldots, x_n$, where x_{2i-1} and x_{2i} are siblings, for $i = 1, \ldots, k$.

Proof: The proof is by induction on k. The base case follows from the fact that there is an optimal binary tree where the two leaves with the smallest weights are siblings; a fact that is used to prove the correctness of Huffman's algorithm [11]. By the induction hypothesis, there is an optimal binary tree where x_{2i-1} and x_{2i} are siblings, for $i = 1, \ldots, j$ and $j < k$. Let T be such a tree with weighted external path length C. Let x'_i be the common parent of x_{2i-1} and x_{2i} in T, for $i = 1, \ldots, j$, implying that $w(x'_i) = w(x_{2i-1}) + w(x_{2i})$. Remove the nodes $x_1, \ldots, x_{2j}$ from T and let T' be the resulting tree with weighted external path length C'. It follows that C' is the minimum-weight external path length among the trees with leaves $x'_1, \ldots, x'_j, x_{2j+1}, \ldots, x_n$, otherwise C would not have been the minimum-weight external path length among the trees with leaves $x_1, \ldots, x_n$. We also have $C = C' + \sum_{i=1}^{2j} w(x_i)$. Since

$$w(x_{2j+1}) \leq w(x_{2j+2}) \leq w(x'_1) \leq \ldots \leq w(x_n)$$

there exists an optimal tree among the trees with leaves $x'_1, \ldots, x'_j, x_{2j+1}, \ldots, x_n$ where x_{2j+1} and x_{2j+2} are siblings. Let T'' be such a tree. It follows that the weighted external path length of T'' is C'. Finally, insert the nodes x_{2i-1} and x_{2i} in T'' by linking them to the node x'_i, for $i = 1, \ldots, j$. Let T^* be such a tree. Since the weighted external path length of T^* equals $C' + \sum_{i=1}^{2j} w(x_i) = C$, it follows that T^* is an optimal binary tree with leaves $x_1, \ldots, x_n$ having x_{2i-1} and x_{2i} as siblings, for $i = 1, \ldots, j+1$. □

Consider the following algorithm that uses W and L to build a tree, which we call $T^?$. The algorithm works by mimicking the behavior of Huffman's algorithm as follows. It starts by getting the sequence of the leaves with the highest level sorted by their weights, and combines every two consecutive leaves in such a sequence adding their weight in an internal node. In the general step, the algorithm uses a sorted sequence of the leaves of level i together with the internal nodes resulting from level $i+1$, and combines every two consecutive nodes in such a sequence adding their weights in an internal node that will be used by the next step. If at any step the number of nodes to be combined is odd, the algorithm will not produce $T^?$. This case means that the given code is not equivalent to a binary tree and hence not optimal. Note that this case can be first checked in linear time, and hence we assume that the tree $T^?$ is constructible.

We define the *exclusion property* for $T^?$ as follows: $T^?$ has the exclusion property if and only if the weights of the nodes (leaves and internal nodes) at level i are not smaller than the weights of the nodes at level $i+1$.

Lemma 2: Given a prefix code whose corresponding tree $T^?$ would be built as above, 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 i and j such that $i < j$ and $w(x) < w(y)$. 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 $n = 2$. Consider any positive integer k, where $w(x_1) + w(x_2) \geq w(x_{2k})$. It follows that the weight of each of the leaves $[x_1, \ldots, x_{2k}]$ does not exceed the weight of any of the internal nodes of $T^?$. As a result of the exclusion property, these $2k$ leaves must be at the highest one or two levels of $T^?$. Among these $2k$ leaves, following the fact that the number of leaves at the highest level is even, the number of leaves at the second highest level (if at all any) must be even. Hence, the way $T^?$ is built implies that x_{2i-1} and x_{2i} are siblings, for $i = 1, \ldots, k$. Using Lemma 1, it follows that there is an optimal binary tree with leaves $x_1, \ldots, x_n$, where x_{2i-1} and x_{2i} are siblings, for $i = 1, \ldots, k$. Remove the nodes $x_1, \ldots, x_{2k}$ from $T^?$ 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. As a result, $T^?$ is an optimal tree and corresponds to an optimal prefix code. □

Lemma 2 suggests that our verification problem is equivalent to checking the exclusion property for $T^?$. Once we have $T^?$, checking the exclusion property can be done in linear time by finding the nodes with the minimum and maximum weights within every level and verifying that the node with the minimum weight at level i is not smaller than the node with the maximum weight at level $i+1$. In general, building $T^?$ by the above algorithm requires $\Omega(n \log n)$. It is crucial to mention that, to check the exclusion property, we do not have to construct $T^?$ explicitly using such algorithm. Instead, we only need to evaluate the nodes with the minimum and maximum weights at every level. In Section III, we show that even evaluating the weights of these nodes requires $\Omega(n \log n)$ in the general case, implying the same lower bound for the verification problem. However, we show in Section IV special cases where the evaluation of the weights of these nodes can be done in linear time.

It is worth mentioning that unless $T^?$ is built as mentioned above, a tree having the exclusion property does not guarantee

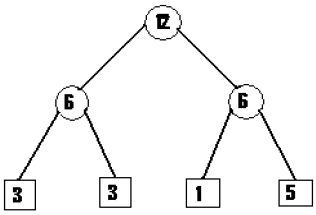

Fig. 1. A nonoptimal tree that has the exclusion property.

the optimality of the corresponding prefix code. This is illustrated with the tree in Fig. 1, which has $W = [1,3,3,5]$ and $L = [2,2,2,2]$ with a weighted external path length equals 24. On the other hand, the optimal tree has $L = [3,3,2,1]$ with a weighted external path length equals 23.

III. The Lower Bound

We show next that the verification of minimum-redundancy prefix codes requires $\Omega(n \log n)$ comparisons in the algebraic decision-tree model.

Using Lemma 2, it follows that the verification of prefix codes is equivalent to verifying the exclusion property for $T^?$. To verify the exclusion property, the verification algorithm needs to evaluate the internal nodes with the minimum weights for each level of $T^?$ that has leaves (we only mention the minimum weights, while the maximum weights are to be evaluated similarly). Otherwise, the weight of one of these leaves may change affecting whether the exclusion property holds or not and the verification algorithm will still be not aware of this consequence.

Consider the following example: Given $W = [2,3,4,5]$, the list of lengths corresponding to an optimal prefix code is either $L = [3,3,2,1]$ or $L = [2,2,2,2]$. Once the weighs change to $W = [2, 3+\epsilon, 4, 5]$ the corresponding optimal list of lengths must be $L = [2,2,2,2]$. Alternatively, once the weights change to $W = [2,3,4,5+\epsilon]$, the corresponding optimal list of lengths must be $L = [3,3,2,1]$.

The intuition behind the lower bound construction being that if we start with $\Theta(n)$ leaves at the highest level and we have adjacent consecutive levels that also have enough leaves, then checking the exclusion property at the second highest level will require identifying the smallest pair of leaves of the highest level. Checking the exclusion property at the next level will require identifying the smallest and the second smallest pairs of leaves of the highest level and so on, indicating that if we have enough adjacent levels containing enough leaves we may need to identify each of the smallest $\Theta(n)$ pairs of the highest level, a problem that requires $\Omega(n \log n)$. More precisely, we will show that, in such case, checking the exclusion property for the ith highest level will require identifying each of the smallest 2^{i-2} pairs of leaves of the highest level. In other words, the lower bound situation holds if there are enough leaves on at least $\log_2 n + O(1)$ consecutive levels.

The detailed construction is as follows. The smartest verification algorithm is given $\Theta(n)$ of the leaves at the highest level, allowing a number of leaves for each level that is a constant fraction of the leaves at the preceding higher level. For example, assuming without loss of generality that n is a power of 2, we may have at least $n/2$ leaves at the highest level, at least $n/4$ leaves at the second highest level, and so on. For a total of at least $\log_2 n + O(1)$ consecutive levels containing leaves. Of course, the relation $\sum_{i=1}^{n} 2^{-l_i} = 1$ must be satisfied. We prove by induction on the levels of $T^?$ that any such algorithm needs to evaluate each of the $n/4$ internal nodes of the second highest level of $T^?$. The base case follows from the fact that any algorithm that checks the exclusion property needs to evaluate an internal node at the shallowest level that has leaves. Using the induction hypothesis, assume that our smart algorithm evaluates the smallest, at least, $n/2^i$ internal nodes at the ith highest level. Consider any of these nodes and call it x, and let y and z be its two children. If the algorithm knows $w(y)$ (either because y is a leaf or an internal node that the algorithm evaluates), it would have evaluated $w(z)$ as $w(x) - w(y)$ in constant extra time. Assume that the algorithm evaluates neither of these children. Assume without loss of generality that $w(y) < w(z)$. Now, if we change the weight of a leaf at this level to become $w(z) - \epsilon$ for some small $\epsilon > 0$, this leaf should have replaced z in the evaluation of x; a fact that would have not been realized unless the algorithm evaluates z. It follows that the algorithm has to evaluate both y and z. Since there are at least $n/2^{i-1}$ leaves at the $i-1$ highest level, there are enough leaves to force the algorithm to evaluate all the children of the smallest $n/2^i$ internal nodes at the ith highest level. Hence, the algorithm needs to evaluate the smallest, at least, $n/2^{i-1}$ internal nodes at the $i-1$ highest level, and the induction hypothesis follows. For $i = 2$, the algorithm has to evaluate each of the $n/4$ internal nodes.

The following lemma shows that evaluating these $n/4$ internal nodes at the second highest level requires $\Omega(n \log n)$.

Lemma 3: Given a sequence $\mathcal{S}$ having an even number of elements n, partitioning $\mathcal{S}$ into $n/2$ pairs, every odd-positioned element with its successor (with respect to the sorted sequence of $\mathcal{S}$), requires $\Omega(n \log n)$ comparisons in the algebraic decision-tree model.

Proof: We present a linear-time reduction from the problem *Element-uniqueness*: given n comparable elements, check whether they are pair-wise distinct. It is known that Element-uniqueness requires $\Omega(n \log n)$ comparisons in the algebraic decision-tree model.

Given a sequence $\mathcal{S}$ with even number of elements, run the algorithm to find the successor of every odd-positioned element with respect to the sorted sequence of $\mathcal{S}$. In linear time, find the smallest element of $\mathcal{S}$ and replace it with an element that is larger than the largest element in $\mathcal{S}$. Apply the algorithm again on the new sequence. The successor of each element of $\mathcal{S}$ is now known, either from the first run or the second run. In linear time, we can compare every element with its successor to check whether they are pair-wise distinct. □

As a result of the above construction and Lemma 3, it follows that checking the exclusion property for a tree $T^?$, when only W and L are given while $T^?$ is not explicitly given, requires $\Omega(n \log n)$ comparisons in the general case. This together with Lemma 2 imply the following main theorem.

Theorem 1: In the algebraic decision-tree model, the verification of minimum-redundancy prefix codes requires $\Omega(n \log n)$ time in the general case.

Proof: Using the above lower-bound construction and Lemma 3, checking the exclusion property for $T^?$ needs $\Omega(n \log n)$ time. Given a binary tree corresponding to a prefix code, obtaining the codeword lengths and weights is done in linear time in n. Consider an Algorithm A that runs on weights and lengths and answers if these lengths correspond to an optimal prefix code or not. If A runs in $o(n \log n)$ time, then Lemma 2 implies an algorithm to check the exclusion property of $T^?$ in $o(n \log n)$ time. We thus have a contradiction, which implies that A has to run in $\Omega(n \log n)$. □

IV. Linear-Time Verification

Two checks are first done in linear time: first, the relation $\sum_{i=1}^{n} 2^{-l_i} = 1$ must be satisfied; second, the condition that the weights of the leaves at lower levels must not be smaller than the weights of those at higher levels must hold.

If the input sequence of weights is sorted, the verification can be done in linear time. One way to do that is to construct $T^?$ in linear time by merging the sorted sequence of leaves with the sorted sequence of internal nodes at every level. The exclusion property is incrementally checked while we are building $T^?$ bottom up. Another alternative is to use the sorted sequence of weights to construct a Huffman tree using Van Leeuwen's linear-time algorithm [4]. The cost of this tree is then evaluated and compared with the cost of the given code.

Other cases that can be verified in linear time include the following.

Case 1. A Tree Claimed to be a Huffman Tree is Given

The following lemma shows an easy linear-time verification algorithm for this case. As a prerequisite for the validity of the lemma, we impose that the weight of the right child of any node is not smaller than the weight of its left child. The given tree can be transformed in linear time to have this property.

Lemma 4: Consider a tree with the property that the weight of the right child of any node is not smaller than the weight of its left child. This tree is a Huffman tree if and only if the exclusion property holds and the weights of same-level nodes are nondecreasing from left to right.

Proof: The optimality of Huffman trees implies the exclusion property. Given a Huffman tree, let b and c be nodes at level i such that b is immediately to the left of c. If b and c are the children of the same parent, then $w(b) \leq w(c)$ and the property holds. Otherwise, let a be the left sibling of b and let d be the right sibling of c. We may assume, by induction on the nodes at level $i-1$, that $w(a)+w(b) \leq w(c)+w(d)$. Assume that $w(b) > w(c)$, then $w(a) < w(d)$. It follows that either $w(a) < w(d) \leq w(b)$ or $w(c) < w(b) \leq w(d)$. As a result, Huffman algorithm would have not combined either a and b or c and d. This contradiction implies that $w(b) \leq w(c)$, proving that the weights of same-level nodes are nondecreasing from left to right. Now, assume that for any given tree the weights of same-level nodes are nondecreasing from left to right. It follows that this tree is precisely the tree $T^?$. If this tree has the exclusion property as well, then Lemma 2 implies that it is a Huffman tree. □

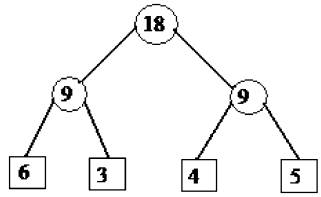

Fig. 2. An optimal tree that is not produced by a Huffman construction.

Once such a tree is available, the nodes are traversed from left to right higher levels first. The resulting sequence of weights is then checked in linear time whether sorted or not. We point out that unless the given tree is a Huffman tree, the above algorithm will fail to verify its optimality. As illustrated by Fig. 2, a tree may be optimal and not obtained by a Huffman construction.

For the next two cases we need the following lemma.

Lemma 5: The exclusion property for $T^?$ does not need to be checked between pairs of consecutive levels that only have internal nodes.

Proof: For two levels i and $i-1$, that only have internal nodes, the exclusion property holds if and only if the sum of the weights of any two nodes at level i is not smaller than the weight of any node at the same level. Assume that the exclusion property holds between levels $i+1$ and i. It follows that the sum of the weights of any two nodes at level $i+1$ is not smaller than the weight of any node at the same level. Since the nodes at level i are all internal nodes, the weight of each of these nodes is the sum of the weights of two nodes at level $i+1$. Therefore, the sum of the weights of any two nodes at level i is the sum of the weights of four nodes at level $i+1$, which is not smaller than the weight of any node at level i. This guarantees the validity of the exclusion property between levels i and $i-1$. Using the fact that the highest level must have leaves, the lemma follows by induction. □

Case 2. The Given Weights Are on at Most $c \log_2 n$ Consecutive Levels, $c < 1$

To check the exclusion property for this case, following Lemma 5, we need to only evaluate the nodes with the minimum and maximum weights within the highest $c \log_2 n + 1$ levels of $T^?$. We also rely on the fact that the evaluation of the minimum (maximum) node at level l_j only requires the, at most, smallest (largest) $2^{l_i - l_j}$ leaves from level l_i, for every $l_i \geq l_j$. Our verification algorithm starts by finding the number of leaves needed from every level; for example, the needed leaves from the highest level are the, at most, $2^{c \log_2 n} = n^c$, $c < 1$ with the smallest and largest weights. We proceed using a selection algorithm to extract these effective leaves in linear time. These effective leaves are then sorted within every level in $o(n)$, for an overall $o(n)$ cost for all the leaves. Finally, the necessary nodes of $T^?$ are evaluated, and the exclusion property is checked during the evaluation of these nodes level by level.

Case 3. The Given Weights Are on a Constant Number of Levels

We sketch the basic ideas of the linear-time verification algorithm, skipping some details for the sake of simplicity.

Let $l_1 < l_2 < \ldots < l_k$ be the levels that have leaves (not necessarily consecutive), for some constant k. Using Lemma 5, we only need to evaluate the nodes of T' with the minimum and maximum weights within these k levels as well as the, at most, $2k-1$ levels that are adjacent to these k levels and only have internal nodes. We claim that the weight of each of these nodes can be evaluated in linear time, which means that the verification can be done in linear time.

The basic idea is clarified through an example having $1.5m+2$ leaves, with m leaves at the highest level, $m/2$ at the following level. and two leaves at level 2. The $1.5m$ leaves, at the highest two levels, combine to produce two internal nodes at level 2. For the verification problem, we need to evaluate the smallest node x of these two internal nodes which amounts to identifying the smallest $m/2$ 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 the $m/2$ internal nodes resulting from the pair-wise combinations of the highest level m nodes. 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 $m/2$ leaves which are known and thus their median M can be found in linear time, and another set containing $m/2$ internal nodes which are not known but whose median M' can still be computed in linear time, by simply finding the two middle elements of the highest level m leaves and adding them. Assuming without loss of generality that $M > M'$, then the bigger half of the $m/2$ leaves at the second highest level can be safely discarded as not contributing to x, and the smaller half of the highest level m leaves are guaranteed to contribute to x. This procedure is repeated recursively on a problem half the size, giving an overall linear-time algorithm whose running time follows the recurrence $T(n) = T(n/2)+O(n)$. Next, we give a description of the general algorithm.

Consider the following algorithm that evaluates the internal node x with the minimum weight at level l (the nodes with maximum weights are evaluated analogously). The algorithm starts by constructing a list of the weights of the leaves that may contribute to the value of x. We call this list the *candidate list*. The candidate list initially contains the, at most, 2^{l_i-l} leaves with the smallest weights among the leaves of l_i, for every level $l_i > l$. Note that no other leaf of l_i may contribute to x. The algorithm next starts a pruning phase to the nodes of the candidate list. During the pruning phase, we classify the nodes of the candidate list as either: contributing to x, not contributing to x, or undetermined. We only keep the undetermined nodes in the candidate list; keep track of the count of the leaves of level l_i in the candidate list, and call it m_{l_i}. We also keep track of the count of the leaves of level l_i that are known so far as contributing to x, and call it m'_{l_i}.

The pruning phase works as follows. Let l_t be the closest level to l among the levels that have the leaves, such that $l_t > l$. Among the nodes at every level $l_i \geq l_t$ of the candidate list, we get the $2^{l_i-l_t}$ nodes with the median weights and partition the nodes of that level accordingly; a process that can be done in linear time with two calls per level to a linear-time selection algorithm. The weights of these median nodes are added together, per level, forming at most $k-t+1$ summations (if there are less than $2^{l_i-l_t}$ nodes at level l_i of the candidate list, these nodes are assumed to be undetermined and will not be considered until the end of the pruning phase). Let M_i be such a summation for level l_i.

The following pruning step is repeated several times. Among the summations, the minimum summation $M_{\min}$ and the maximum summation $M_{\max}$ are identified. The crucial observation is that we can either confirm that almost half the nodes with the largest weights within $l_{\max}$ are not contributing to x (the weights of these nodes are larger than almost half the weights of the nodes of the candidate list), or that almost half the nodes with the smallest weights within $l_{\min}$ are contributing to x (the weights of these nodes are smaller than almost half the weights of the nodes of the candidate list). More precisely, if $\sum_{i=t}^{k} m_{l_i}/2^{l_i-l}$ is less than twice $1-\sum_{i=t}^{k} m'_{l_i}/2^{l_i-l}$, we prune nodes from $l_{\min}$ and set j to min. Otherwise, we prune nodes from $l_{\max}$ and set j to max. In either case, we get the $2^{l_j-l_t}$ nodes with the median weights among the still undetermined nodes of l_j, reevaluate M_j, and partition the nodes of l_j within the candidate list accordingly (again, if there are less than $2^{l_j-l_t}$ nodes at level l_j, the leaves of this level are assumed to be undetermined and will not be considered until the end of the pruning phase). Note that the work done to determine these new median nodes is a linear function of the number of leaves of l_j. This cost can be charged to the nodes of l_j whose status have just been determined, and are thereby pruned from the candidate list, for an overall linear amount of work for the pruning phase. We repeat the pruning step as above until either

- $\sum_{i=t}^{k} m'_{l_i}/2^{l_i-l} = 1$: where the pruning phase ends by deciding that all the undetermined nodes are not contributing to x; or
- $\sum_{i=t}^{k} (m_{l_i}+m'_{l_i})/2^{l_i-l} = 1$: where the pruning phase ends by deciding that all the undetermined nodes are contributing to x; or
- $m_{l_i}/2^{l_i-l_t} = O(1)$ for every l_i: this means that the remaining undetermined nodes only contribute to a constant number of nodes at level l_t that may, in turn, contribute to the weight of x. In such case, we recursively evaluate these nodes one by one in order, using the remaining nodes of the candidate list, and use their weights to decide which of them contribute to x.

In summary, to evaluate x, the algorithm spends a linear amount of work and evaluates a constant number of nodes at levels higher than the level of x, among the k levels that have the leaves. To end the recursion, any node at level l_{k-1} is directly evaluated as the sum of the weights of the corresponding $2^{l_k-l_{k-1}}$ leaves at level l_k. Note that for the case $k=2$, we only need to evaluate the nodes with the maximum weight at level l_1 and level l_1-1, and the two nodes with the minimum and second minimum weight at level l_1.

Next, we sketch the intuition behind the correctness of the algorithm. Assume that

$$\sum_{i=t}^{k} m_{l_i}/2^{l_i-l} \geq 2\left(1-\sum_{i=t}^{k} m'_{l_i}/2^{l_i-l}\right)$$

(the case with the less than works analogously), i.e.,

$$\sum_{i=t}^{k} m_{l_i}/2^{l_i-l_t} \geq 2\left(2^{l-l_t} - \sum_{i=t}^{k} m'_{l_i}/2^{l_i-l_t}\right).$$

The left-hand side is basically the number of nodes that the current members of the candidate list contribute to l_t, while the right-hand side is twice the number of nodes from l_t that are left to contribute to x. Since we adopt the invariant that all the nodes that are left to contribute to x are members of the candidate list, the above inequality indicates that among the nodes of l_t, that the nodes of the candidate list constitute, only less than half contribute to x. Since almost all the nodes with weights larger than the median weights of $l_{\max}$ contribute to nodes whose weights are in the larger half of the nodes of l_t, among the nodes that the candidate list constitute, it follows that these nodes are not contributing to x.

V. Conclusion

Given a list of integer codeword lengths claimed to correspond to a prefix code for a list of positive weights, we showed that verifying the optimality of such code requires $\Omega(n \log n)$ in the algebraic decision-tree model. This directly implies the same lower bound for constructing a list of optimal codeword lengths. Despite the existence of an $\Omega(n \log n)$ lower bound for generating Huffman codes, that is directly inherited from the sorting problem, the new result establishes the lower bound for any prefix-code construction algorithm.

References

[1] J. Gailly. GZIP Program and Documentation. [Online]. Available: prep.ai.mit.edu:/pub/gnu/gzip_*.tar

[2] J. Zobel and A. Moffat, "Adding compression to a full-text retrieval system," *Software: Practice and Experience*, vol. 25, no. 8, pp. 891–903, 1995.

[3] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, pp. 1098–1101, 1952.

[4] J. Van Leeuwen, "On the construction of Huffman trees," in *Proc. 3rd Int. Colloquium for Automata, Languages and Programming*, Edinburgh, Scotland, U.K., Jul. 1976, pp. 382–410.

[5] B. Chazelle, "A minimum spanning tree algorithm with inverse-Ackermann type complexity," *J. Assoc. Comput. Mach.*, vol. 47, no. 6, pp. 1028–1047, 2000.

[6] J. Komlós, "Linear verification for spanning trees," *Combinatorica*, vol. 5, pp. 57–65, 1983.

[7] D. Karger, P. Klein, and R. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *J. Assoc. Comp. Mach.*, vol. 42, pp. 321–328, 1995.

[8] M. Buro, "On the maximum length of Huffman codes," *Inf. Process. Lett.*, vol. 45, pp. 219–223, 1993.

[9] A. Moffat and A. Turbin, "Efficient construction of minimum-redundancy codes for large alphabets," *IEEE Trans. Info. Theory*, vol. 44, no. 4, pp. 1650–1657, Jul. 1998.

[10] R. Milidiú, A. Pessoa, and E. Laber, "Three space-economical algorithms for calculating minimum-redundancy prefix codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 6, pp. 2185–2198, Sep. 2001.

[11] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.