SIAM J. DISC. MATH.
Vol. 8, No. 4, pp. 638–651, November 1995
© 1995 Society for Industrial and Applied Mathematics
011

UPPER AND LOWER BOUNDS ON CONSTRUCTING ALPHABETIC BINARY TREES*

MARIA KLAWE† AND BRENDAN MUMEY‡

Abstract. This paper studies the long-standing open question of whether optimal alphabetic binary trees can be constructed in $o(n \lg n)$ time. We show that a class of techniques for finding optimal alphabetic trees which includes all current methods yielding $O(n \lg n)$-time algorithms are at least as hard as sorting in whatever model of computation is used. We also give $O(n)$-time algorithms for the case where all the input weights are within a constant factor of one another and when they are exponentially separated.

Key words. alphabetic binary trees, data structures, algorithms

AMS subject classifications. 05C10, 05C35

1. Overview.

The problem of finding optimal alphabetic binary trees can be stated as follows: Given a sequence of n positive weights $w_1, \ldots, w_n$, construct a binary tree whose leaves have these weights, such that the tree is optimal with respect to some cost function and also has the property that the weights on the leaves occur in order as the tree is traversed from left to right. A tree that satisfies this last requirement is said to be *alphabetic*. Although more general cost functions can be considered (as is done in [4] and [7]), we concentrate here on the usual function, namely $\sum w_i l_i$, where l_i is the level of the ith leaf from the left in the tree. The first $O(n \lg n)$-time solution was given in Hu and Tucker [5] in 1971, following algorithms with higher complexity in [3] and [6]. If we remove the restriction that the tree must be alphabetic, then the problem becomes the well-known problem of building Huffman trees, which is known to have $\Theta(n \lg n)$-time complexity in the comparison model. Modifications of the Hu–Tucker algorithm also running in $O(n \lg n)$ time but with simpler proofs are given in [2] and [4]. The only recent progress on this problem has been made by Ramanan [8], who showed that it is possible to verify that a given alphabetic tree on a sequence of weights is optimal in $O(n)$ time when the weights in the sequence are either within a constant factor or exponentially separated (notions we define precisely later). However, it seems substantially more difficult to actually construct the optimal tree in linear time in the constant factor case.

The next section summarizes current methods and introduces the concepts needed to frame our results. In §3, we introduce a technique, *region processing*, which forms the basis of our linear-time algorithms. We start with a fairly simple $O(n)$-time algorithm for finding the optimal alphabetic tree when the weights are within a factor of two. We also observe that the basic region-processing method solves the case where the input weights are exponentially separated in $O(n)$ time. We generalize this technique in §4 to the case where all the weights are within a constant factor of one another. The generalization depends on solving a new *generalized selection* problem, which may be of interest in its own right. In §5, we give reductions of sorting

* Received by the editors September 1, 1993; accepted for publication (in revised form) October 12, 1994. This research was supported in part by the Natural Sciences and Engineering Council of Canada.

† Department of Computer Science, University of British Columbia, Vancouver, British Columbia V6T 1Z2, Canada.

‡ Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195.

problems to Hu–Tucker-based algorithms and region-processing methods. This provides $\Omega(n \lg n)$-time lower bounds for Hu–Tucker-based algorithms in the comparison model and indicates that region-processing methods are unlikely to yield an $o(n \lg n)$ algorithm.

2. Current methods. We give a brief description of the Hu–Tucker algorithm to the extent necessary to explain our results. Complete descriptions and explanations can be found in [5], [4], [7]. All Hu–Tucker-based methods begin by building an intermediate tree, called the *lmcp tree*, whose leaves hold the given set of input weights, though not necessarily in the correct order. The levels of the input weights in the lmcp tree are recorded, and this information is used to build an alphabetic tree on the input weights, with each input weight occurring at the same level as in the lmcp tree. Constructing this alphabetic tree can easily be done in $O(n)$ time, as shown in [5]. Since the cost function depends only on the levels of the leaf nodes, the cost of the alphabetic tree is the same as the cost of the lmcp tree. Hu and Tucker proved that the lmcp tree has optimal cost in a class of trees that contains all alphabetic trees, and hence it follows that the alphabetic tree constructed is optimal. We are able to prove that, in the comparison model, constructing the lmcp tree requires $\Omega(n \lg n)$ time in the worst case, but since it suffices to know only the levels of the leaf weights in the lmcp tree and not its full structure, we can improve on the performance of the Hu–Tucker algorithm in a number of cases.

The Hu–Tucker algorithm maintains a *worklist* of weighted nodes in the lmcp tree that have not yet been assigned their sibling and parent. The basic step in the algorithm consists of selecting two nodes from the worklist to be paired off as siblings in the lmcp tree, removing these nodes from the worklist, and inserting a new node (their parent) in the position of the leftmost replaced node with weight equal to the sum of the two removed nodes. Initially the worklist is the list of leaf nodes with the weights $w_1, \ldots, w_n$ in order. Nodes in the worklist are designated either *crossable* or *noncrossable*. Initially all nodes are noncrossable. When any two nodes are paired off, the resulting parent node is designated crossable. Two nodes in the worklist are *compatible* if they are adjacent, or if all the nodes which separate them are crossable. The symbol v will refer to a node in the worklist and $w(v)$ will refer to its weight. The level of a node v in the tree is denoted by $l(v)$. Define an order on the nodes in the worklist by $v_x < v_y$ if $w(v_x) < w(v_y)$ or if $w(v_x) = w(v_y)$ and v_x is to the left of v_y in the list. A pair of compatible nodes (v_a, v_b) is said to be a *local minimum compatible pair* (*lmcp*) if and only if the following two conditions hold:

1. $v_b \leq v_x$ for all nodes v_x compatible with node v_a.
2. $v_a \leq v_y$ for all nodes v_y compatible with node v_b.

We note that the order relationship given captures the tie-breaking rules of [5] and [2].

The lmcp tree is constructed by repeatedly combining lmcps from the worklist until a single node remains which will be the root of the lmcp tree. This is usually implemented by a stack-based algorithm that starts at the beginning of the worklist and moves a pointer along the worklist until an lmcp is found. After removing the nodes in the lmcp and inserting the new parent node, the pointer is moved back one node and the search for lmcps resumes. To check whether an lmcp has been found, the algorithm compares the smallest node x before the pointer node y that is compatible with y with the smallest node z after y that is compatible with y. If $x < z$, the algorithm concludes that x and y form an lmcp; otherwise, it moves the pointer forward one node. The total number of pointer moves is $O(n)$, since $O(n)$ nodes are

placed in the worklist in total, and the number of backward moves is bounded by the number of lmcps found, which is also $O(n)$. Hu–Tucker methods take $O(n \lg n)$ time because they maintain information on which node has the minimum weight in intervals of crossable nodes in order to find the nodes x and z. Updating this information when an lmcp is found can take $O(\lg n)$ time. In general, the *construction* of the lmcp tree is not unique, since the lmcps may be combined in different orders, but, as proved in [5], the resulting tree is unique. Thus, for any node v in the worklist, we can define the *lmcp partner* of v to be the node that is the sibling of v in the lmcp tree.

3. Region-based methods. We present a new approach for finding optimal alphabetic binary trees based on partitioning nodes in the worklist in consecutive runs. Define the *category* of weight w to be $\lfloor \lg (w/w_{\min}) \rfloor$, where $w_{\min}$ is the smallest of the initial weights. A maximal-length sequence of nodes with the same category is called a *region*. In our presentation, we assume that we explicitly compute the category of each weight, since this simplifies the description and explanation of our approach. However, it is possible to avoid the possibly nonunit cost of the lg operations needed to determine the categories explicitly by modifying the algorithm to treat the category numbers of weights as unknowns that can be compared at unit cost. We omit the details of this modification, as they are not crucial to the understanding of the main algorithm.

By keeping a stack of regions and considering only regions whose adjacent regions have higher category, we can restrict most of our attention to the pairings occurring within these regions. We call this *region processing*. This is motivated by the situation where all input weights are within a factor of two. If this is the case, it is easy to determine the leaf levels in the lmcp tree using Theorem 3.1.

THEOREM 3.1. *Given a sequence of n crossable nodes that are within a factor of two, after the first $\lfloor (n+1)/2 \rfloor$ lmcps have been found and combined, the new sequence will consist of $\lfloor n/2 \rfloor$ nodes whose weights are again within a factor of two. Furthermore, if we keep combining lmcps, the resulting lmcp tree will be balanced, with the leaves differing in level by at most one. Specifically, the $2(n - 2^{\lfloor \lg n \rfloor})$ smallest weights will be at level $\lfloor \lg n \rfloor + 1$ and the others will be at level $\lfloor \lg n \rfloor$.*

Proof. We note that, since all the nodes are crossable, this reduces the problem to building a Huffman tree, where the result is known. We present a new proof, which provides insight to the actual behavior of the algorithm and motivates our results to follow.

Let the initial sequence of nodes in the worklist be $v_1, \ldots, v_n$ and let c be a real number such that $c \leq w(v_i) < 2c$ for $i = 1$ to n. Whenever two nodes form an lmcp and combine, the weight of the new node is greater than $2c$, so it will not be involved in another lmcp until there are less than two nodes smaller than $2c$. When n is odd, after $(n-1)/2$ pairings have occurred, the worklist contains only one node of weight less than $2c$, namely the largest-weight node present in the original sequence. We call this node the *wallflower*. The wallflower forms an lmcp with the smallest-weight newly formed node. When n is even, the largest-weight node present in the original sequence merges with another original node. Thus, regardless of whether n is odd or even, the rightmost (there may be more than one) largest-weight node will merge during the $\lfloor (n+1)/2 \rfloor$th lmcp pairing. At this stage, the worklist will contain exactly $\lfloor n/2 \rfloor$ nodes, none of which are original nodes, and their weights will be within a factor of two, as we show below.

This is obvious if n is even, so suppose n is odd, and let v be the node with the smallest weight, $w(v) = w(v_i) + w(v_j)$, among the first $(n-1)/2$ newly formed nodes.

Clearly, the rest of the first $(n-1)/2$ newly formed nodes have weights less than $2w(v)$. Let v_k be the wallflower. The next node formed is the parent of v and v_k and has weight $w(v_k)+w(v_i)+w(v_j)$. Now, since the original weight sequence was within a factor of two, $w(v_k) < w(v_i) + w(v_j) = w(v)$, so $w(v_k) + w(v_i) + w(v_j) < 2w(v)$, which completes the proof. One further observation that will be important is that the weight of the parent of the wallflower is strictly greater than the weight of the other $(n-1)/2$ nodes in the current worklist.

Let us call the pairings up to this point a *phase* of the algorithm and consider how the phase affects the levels of the leaves in the lmcp tree. Obviously, the phase contributes one to the level of each leaf in the lmcp tree if n is even. When n is odd, this is true for all the leaves except for the two whose parent was paired with the wallflower. These two, which we call the wallflower's stepchildren, have had their level increase by exactly two. Since the wallflower's parent has the unique largest weight in the worklist at the end of the phase, at the end of each later phase this node's ancestor always has the unique largest weight in the worklist. Thus each later phase contributes exactly one to the level of the wallflower's stepchildren. Applying this argument to the stepchildren of wallflowers from later phases proves that the level of any two leaves in the lmcp tree differs by at most one. Since the lmcp tree has optimal cost, the smallest-weight original nodes must be at the bottom level, i.e., the largest-numbered level. Thus for some integer x, we have the $2x$ smallest-weight original nodes on level $\lfloor \lg n \rfloor + 1$ and the remaining $n - 2x$ original nodes on level $\lfloor \lg n \rfloor$. We require $x + n - 2x = 2^{\lfloor \lg n \rfloor}$, so $x = n - 2^{\lfloor \lg n \rfloor}$. □

Based on this theorem, it is easy to give a simple linear-time algorithm for finding an optimal alphabetic binary tree on a sequence of input weights which differ at most by a factor of two. (Garcia and Wachs also give a linear-time method for this case in [2].) In point form, the algorithm for finding the levels of the leaves in the alphabetic tree is:

1. Initialize the worklist to contain the original input sequence. Note that all nodes are noncrossable.
2. Use a stack-based method to find lmcps and pair them off, removing each pair of nodes from the worklist and placing the parent in a temporary list but not in the worklist. These newly formed nodes can be left out of the worklist because their weights are greater than any of the original weights, and hence need not be considered in the search for lmcps. This process continues until there are zero or one nodes left in the worklist, and as discussed in the remarks on stack-based algorithms in §2, requires only $O(n)$ time because of the absence of crossable nodes in the worklist. If a single node x remains (n is odd and x is the wallflower), scan through the temporary list of newly formed crossable nodes to find the smallest node y. Pair x with y and replace y in the temporary list by its parent.
3. At this stage we have $m = \lfloor n/2 \rfloor$ crossable nodes in the temporary list. Moreover, the new nodes are still within a factor of two, by the same argument as in the proof of the preceding theorem.
4. We can now, by the preceding theorem, directly find the levels of every leaf in the lmcp tree for the remaining m crossable nodes in $O(n)$ time, using a linear-time selection algorithm [1] to find the $2(m - 2^{\lfloor \lg m \rfloor})$th weight in the temporary list. This node and nodes with smaller weights have level $\lfloor \lg m \rfloor + 1$, and the remaining nodes are assigned level $\lfloor \lg m \rfloor$. Given this, it is trivial to compute the levels of the nodes in the original input sequence in

an additional $O(n)$ time.

5. With knowledge of the leaf levels, we can construct the optimal alphabetic tree for the input sequence in $O(n)$ time, using the technique in [5].

A similar technique can be applied to predict how nodes in a region R with lowest category number combine to form nodes in a region with the next category number. Notice that when the number of nodes in R is odd, its wallflower will pair with the smallest-weight node in the set consisting of the lmcps formed out of R and the compatible nodes from the two regions adjacent to R. When the gap in category number between adjacent regions is large enough, this method yields faster performance than the Hu–Tucker algorithm. The complete algorithm is described in [7]. Its basic idea is to maintain a stack of the current regions in the worklist and process the region at the top of the stack if its adjacent regions have greater category. If not, the stack pointer is advanced. The cost of processing a region of size r is $O(r \lg r)$. Since processing a region yields a new region of half the size, it is easy to verify that this method has $O(n \lg n)$ running time. If the input weights $\{w_i\}$ are exponentially separated, i.e., if there is a constant C such that for all integers k, $|\{i : \lfloor \lg w_i \rfloor = k\}| < C$, then it is also easy to verify that this method yields an $O(n)$-time algorithm, since each region can be processed in constant time as the size is bounded by $2C$. The ideas in Theorem 3.1 can also be used to reduce the cost of processing a region of size r to below $O(r \lg r)$ when the difference in category numbers is great enough, which may be useful in implementations. Details are given in [7].

4. The constant factor case. We now describe the linear-time algorithm for weights within a constant factor, i.e., such that $\max\{w_i/w_j\} < \sigma$ for some constant σ. As before, it suffices to determine the levels of the leaf nodes in the lmcp tree. We use a region-based method to process the weights region by region in increasing order by category number until we are left with a single region of crossable nodes. We then apply Theorem 3.1 to determine the lmcp tree levels of the nodes in this final region and work backwards to find the lmcp tree levels of the original weights. In order to achieve the linear-time bound, when processing a region, we cannot afford to determine which nodes pair together in lmcps or the weights of the lmcps formed. Instead, we work with coarser information about the structure of the lmcp tree. An interval of nodes in a region's worklist is *lmcp-closed* if the lmcp partner of each node in the interval is also in the interval. Our algorithm works by partitioning the region's worklist into lmcp-closed intervals and replacing each lmcp-closed interval by a *node group* representing the lmcps formed out of that interval. From the definition of the lmcp, it is easy to see that moving an interval of larger crossable nodes to the right of an interval of smaller crossable nodes or pushing a larger crossable node to the right of a smaller noncrossable node does not affect the construction of the lmcp tree. Our algorithm uses such rearrangements of the worklist in finding the partition into lmcp-closed intervals.

The worklist thus is now an ordered list of node groups in which each noncrossable node appears as a singleton node group but intervals of crossable nodes within a region may appear in groups of arbitrary size. A set of nodes in the worklist is *realizable* if it is the union of a set of node groups in the worklist. The algorithm performs certain types of selection operations on realizable sets of nodes in the worklist. For example, when the worklist consists of crossable nodes whose weights are within a factor of two, the algorithm determines the smallest k of these nodes in order to apply Theorem 3.1.

Since we will generally not have an explicit list of the weights of the nodes in the realizable set on which we wish to perform selection, we will introduce the concept of *a coarse-selection system*, namely a structure for (nonexplicitly) representing a set of elements, together with a particular set of selection operations that can be performed efficiently on the set. We will then show that each realizable set has a coarse-selection system. Performing a selection operation on a realizable set may require that some of the node groups in the realizable set be refined, in order that the result be in the form of realizable sets. For example, suppose N is a realizable set of nodes in the worklist. Determining the largest (smallest) node v in N requires replacing the node group containing v by a node group list in which v is a singleton node group, unless v is already a singleton. Similarly, determining the k smallest nodes in N requires a node-group list in which the desired set is the union of a set of node-groups in the refined list. Thus we will ensure that the selection operations we provide for realizable sets determine the appropriate refinements. We now define coarse-selection systems.

DEFINITION 4.1. *For any $\Delta \geq 1$, we say a (multi) set S has a Δ* coarse-selection system *if:*

1. *$\forall \alpha \in [0,1]$, in $\Delta|S|$ time we can produce two disjoint sets S_α^- and S_α^+, each with Δ coarse-selection systems such that $S = S_\alpha^- \cup S_\alpha^+$, $\forall x \in S_\alpha^-$ and $\forall y \in S_\alpha^+$, $x \leq y$, and $|S_\alpha^-| = \lfloor \alpha |S| \rfloor$. (We call this an α-*partition *of S.)*
2. *$\forall x \geq 0$, in $\Delta|S|$ time, we can compute the rank of x in S, denoted by $r_S(x)$, and produce two sets $S^{\leq x}$ and $S^{>x}$, each with Δ coarse-selection systems such that $S^{\leq x} = \{y \in S : y \leq x\}$ and $S^{>x} = \{y \in S : y > x\}$. (The rank of x in S is the number of elements in S less than or equal to x.)*
3. *In $\Delta|S|$ time we can compute $|S|$.*
4. *If $|S| = 1$ we can determine the unique element of S explicitly in Δ time.*

In addition, when interpreted in the context of node-group lists, we require that the sets $S_\alpha^-, S_\alpha^+, S^{\leq x}, S^{>x}$ be realizable. Note that the definition of a coarse-selection system implies that, given a Δ coarse-selection system for S, we can explicitly determine the largest (smallest) element of S in $2\Delta|S|$ time. We use the term *layer h* for the regions in the worklist with category number h and process the regions in the worklist a layer at a time beginning with the smallest numbered layer. Processing layer h consists of creating node-group lists representing the new nodes formed in layer $h+1$. Consider the question of creating a node-group list representing the new nodes, T, formed from a single region R of r nodes. If r is even, because the regions adjacent to R in the worklist have higher category numbers, R is lmcp-closed and the node-group list for T is a single node group. If r is odd, then the only node of R whose lmcp partner is not in R is its wallflower z. It is straightforward to prove that z is the largest node in the subset $\{y \in R : y$ is crossable or y is noncrossable and is in an odd-numbered position from an end of $R\}$. Note that this subset is realizable and that z can be identified by coarse selection. Thus we create a node group, g_l, representing the lmcps formed from the nodes on the left of z, and another one, g_r, for those from the right, respectively. To determine the lmcp partner of z, we need to know the smallest node v in $g_l \cup g_r$, which again is realizable. We complete the processing of z by comparing v with the smallest compatible nodes on either side of g_l, g_r in the worklist (found using selection on realizable sets), and we replace z and its partner by a singleton node group representing this lmcp. This singleton node group may be in layer $h+2$, in which case we place it as far to the right as possible (in front of the first node to the right that is in layer $h+2$ or higher). The remaining challenge is to construct the coarse-selection systems for realizable sets, which is done

by induction on layer number.

Our inductive hypothesis will be that, for any node-group list representing the nodes in a region of layer h, and any set of nodes, A, that is realizable with respect to that node-group list, there is a coarse-selection system for A. The base case is covered by the usual linear-time selection algorithm, since all nodes in the bottom layer are noncrossable. Thus the only possible node-group list for the bottom layer is the standard list of single nodes, so all the weights of the nodes in the list are known explicitly. A key tool is the construction of a coarse-selection system for the union of sets with coarse-selection systems. This is provided by the following theorem.

THEOREM 4.2. *Let $A = \cup_{i=1}^{n} A_i$, where the A_i are disjoint and nonempty and each A_i has a Δ coarse-selection system. Then A has a 36Δ coarse-selection system.*

Proof. Let x be any value. We can compute the rank of x in A easily, since $r_A(x) = \sum_{i=1}^{n} r_{A_i}(x)$. Moreover, $A^{\leq x} = \cup_{i=1}^{n} A_i^{\leq x}$ and $A^{>x} = \cup_{i=1}^{n} A_i^{>x}$. The time cost for this is the cost of finding $r_A(x)$ plus the cost of constructing the $A_i^{\leq x}$ and $A_i^{>x}$. This is $\sum_{i=1}^{n} \Delta|A_i| + \sum_{i=1}^{n} \Delta|A_i| = 2\Delta|A|$.

For $\alpha \in [0,1]$, we construct A_α^- and A_α^+ as follows. For each i, compute $A_{i_{1/2}}^-$, $A_{i_{1/2}}^+$, and $m_i = \min A_{i_{1/2}}^+$. This can all be done in $2\Delta|A|$ time. We now compute the median m of the multiset $M = \cup_{i=1}^{n} M_i$, where M_i contains exactly $|A_i|$ copies of m_i, by using a standard selection algorithm. This can be done in $6|A|$ time using the selection algorithm of Blum et al. [1]. Now compute $r_A(m)$ as above, in $\Delta|A|$ time. If $r_A(m) = \lfloor \alpha|A| \rfloor$, we are done, as we can take $A_\alpha^- = A^{\leq m}$ and $A_\alpha^+ = A^{>m}$. If not, we may assume $r_A(m) > \lfloor \alpha|A| \rfloor$, since a symmetric argument handles the other case. Let $J = \{i : m_i \geq m\}$, let $B = A - \cup_{i\in J} A_{i_{1/2}}^+$, and note that every element in $A - B$ is at least m. If $|B| < \lfloor \alpha|A| \rfloor$, since $r_A(m) > \lfloor \alpha|A| \rfloor$, there must be at least $\lfloor \alpha|A| \rfloor - |B|$ elements in $A - B$ that equal m. Thus, it suffices to identify a subset D of these elements, with $|D| = \lfloor \alpha|A| \rfloor - |B|$, and take $A_\alpha^- = B \cup D$. To find D, we first find $(A_{i_{1/2}}^+)^{\leq m}$ for each i in J. Every element in $\cup_{i\in J}(A_{i_{1/2}}^+)^{\leq m}$ must equal m, and thus it suffices to take D to be any subset of $\cup_{i\in J}(A_{i_{1/2}}^+)^{\leq m}$ of the appropriate size. Such a subset can easily be obtained by taking each $(A_{i_{1/2}}^+)^{\leq m}$ until adding another set will result in more than $\lfloor \alpha|A| \rfloor - |B|$. At this point, coarse selection can be used on this $(A_{i_{1/2}}^+)^{\leq m}$ to obtain a subset that will bring the total number of elements to exactly $\lfloor \alpha|A| \rfloor - |B|$. Thus, in this case, we will have obtained A_α^- and A_α^+ in at most $(6 + 5\Delta)|A|$ time. If $|B| \geq \lfloor \alpha|A| \rfloor$, we may take $(A - B) \subset A_\alpha^+$, since every element in $A - B$ is at least m. Note that $\sum_{i\in J} |A_i| \geq \frac{1}{2}|A|$ by the definition of M. Hence $|A - B| = |\cup_{i\in J} A_{i_{1/2}}^+| \geq \frac{1}{4}|A|$, and so we reduce the problem to finding a β-partition in B, where $\beta = \alpha\frac{|A|}{|B|}$. We set $A_\alpha^- = B_\beta^-$ and $A_\alpha^+ = \cup_{i\in J} A_{i_{1/2}}^+ \cup B_\beta^+$. In this case, we reduce the problem to one at most $3/4$ of the original size in $(6 + 3\Delta)|A|$ time. Since B is a union of sets with Δ coarse-selection systems, an easy inductive argument on the size of A shows that we can produce A_α^- and A_α^+ in $\frac{1}{1-3/4}(6 + 3\Delta)|A| \leq 36\Delta|A|$ time.

The fact that $A^{\leq x}$, $A^{>x}$, A_α^-, and A_α^+ each have 36Δ coarse-selection systems again follows easily by induction on $|A|$ since they are unions of sets with Δ coarse-selection systems. □

We are now ready to begin the inductive proof of the existence of coarse-selection systems. We assume that, given any node-group list representing the nodes in a region of layer h and a set of nodes that is realizable with respect to that node-group list, the set has a Δ coarse-selection system. Given this assumption, we show how

to construct a $D\Delta$ coarse-selection system for any set S of nodes in a region X of layer $h+1$ such that S is realizable with respect to a node-group list for X. The value of D is a constant independent of h. By the definition of a node-group list, it is clear that any node-group list for X inherently provides node-group lists for the regions in layer h that contain the children of nodes in X. By the preceding theorem, we may assume that there are no singleton node groups in the representation of S, since otherwise we can use the usual linear-time selection algorithm for the set S^* of nodes in S occurring as singletons, and we can use the selection systems for S^* and $S - S^*$ to get a selection system for S. This assumption says that there is a set $\{R_i\}$ of disjoint lmcp-closed realizable intervals in layer h such that S is the lmcps formed from $V = \cup_i R_i$. We first show how to find the smallest-weight node in S by proving that, in $O(\Delta|S|)$ time, we can reduce the problem to finding the smallest-weight node in a realizable subset S' of S, where $|S'| \leq |S|/2$. This reduction process may involve refining some node-group lists for regions in layer h, and such refinements increase the number of realizable sets. This is why our inductive assumption ensures the existence of Δ coarse-selection systems for realizable sets, independent of which node-group list is used in the definition of realizability. Finding the smallest node is a special case of finding an α-partition, but the algorithm is slightly simpler. Moreover, since it is a subroutine used in finding general α-partitions, presenting it first clarifies the exposition.

The set V is realizable, so, in $\Delta|V|$ time, we can find the 1/2-partition $V = V^-_{1/2} \cup V^+_{1/2}$. For each R_i, we write $R_i^- = R_i \cap V^-_{1/2}$ and $R_i^+ = R_i \cap V^+_{1/2}$. We assume, by reordering if necessary, that for each interval C of crossable nodes in R_i, we have $C \cap R_i^-$ preceding $C \cap R_i^+$.

We now describe an algorithm which we will run on R_i to partition its nodes into three lmcp-closed sets, $R_i = R_i^{--} \cup R_i^{++} \cup R_i^{-+}$, according to whether the node and its lmcp partner are in the same class in the partition $R_i = R_i^- \cup R_i^+$. The set R_i^{--} is the set of nodes x such that both x and its lmcp partner, $p(x)$, are in R_i^-. The sets R_i^{++} and R_i^{-+} are defined analogously. For each node x in R_i^{-+} (the set in which x and $p(x)$ are in different classes), the algorithm explicitly determines x and $p(x)$ and hence can create a singleton node group for the lmcp of x and $p(x)$.

We use the terms *−interval* [*+interval*] to refer to a maximal interval of nodes in R_i which lies entirely in R_i^- [R_i^+]. Obviously, R_i is an alternating sequence of −intervals and +intervals. Also, −intervals and +intervals are realizable sets. We first note that, if any two consecutive −intervals are separated by a +interval that does not contain noncrossable nodes, we may push the +interval to the right of the right-hand −interval without affecting the formation of lmcps. Thus, in linear time, we can rearrange each R_i so that there is at least one noncrossable node in each +interval, except for possibly one on the right end of R_i. If the number of nodes in a −interval, I, is even, then for each $x \in I$ we have $p(x) \in I$. This follows from the fact that S is realizable and that each node group of S represents the lmcps formed out of a consecutive interval in layer h. Next, for each −interval, I, with an odd number of nodes, we use the Δ coarse-selection system to find its *local wallflower*, i.e., the largest node in I which either is crossable or is noncrossable and in an odd-numbered position relative to I. Note that each local wallflower x is now represented by a singleton node group, and we know its weight. Let I' be the set resulting from removing the local wallflower from I, if it has one. It is not hard to prove that, for each $x \in I'$, we have $p(x) \in I'$, so we set R_i^{--} to be the union of the I'. We now remove the node groups representing the nodes in R_i^{--} from the node-group list of R_i.

We will process the list of node groups that remain in $O(\Delta|R_i|)$ time to determine the lmcp partner of each local wallflower and define R_i^{-+} as the set of local wallflowers (i.e., the nodes in R_i^- which still remain in the list) together with their lmcp partners. R_i^{++} is $R_i - (R_i^{--} \cup R_i^{-+})$.

In order to determine the lmcp partner of each local wallflower, we first identify, for each end of a +interval, the smallest-weight node in the +interval compatible from that end of the interval. For each +interval that contains at most one noncrossable node, we also identify its smallest-weight crossable node, if one exists. This can be done in $O(\Delta|R_i|)$ time using coarse-selection systems. We now use the standard stack-based method described at the end of §2, where the worklist consists of the local wallflowers surrounded by the identified neighboring nodes from the +intervals, in order. The stack pointer is initially placed on the leftmost local wallflower. We stop when all the local wallflowers have been paired off, i.e., when their lmcp partners have been determined. It is straightforward to check that, upon removal of an lmcp involving a local wallflower, x, the necessary information on the affected +intervals can be updated in constant time, and this guarantees the linear-time bound.

For $j = --, ++, -+$, let V^j be the union of the nodes in the R^j, and let S^j be the nodes formed from V^j. We note that all the nodes in S^{--} are less than or equal to the nodes in S^{++}, though it is possible that there are nodes in S^{-+} that are smaller than some in S^{--} and others in S^{-+} that are greater than some in S^{++}. In addition, we know that both $|S^{--}|$ and $|S^{++}|$ are less than $|S|/2$ since $|S^{--}| = |S^{++}|$. We also know all the nodes (and their weights) explicitly in V^{-+} and hence can find the smallest node in S^{-+} in $O(|S^{-+}|)$ time. Thus, it suffices to find the smallest node in S^{--}, and taking $S' = S^{--}$ completes the proof. The analogous technique works to find the largest node in S or the rank of a node x and the sets $S^{\leq x}$ and $S^{>x}$ in $O(\Delta|S|)$ time. We will call the process of determining the sets S^{--}, S^{-+}, S^{++} *sifting.*

Now suppose we wish to find S_α^- and S_α^+ for some $\alpha \in [0,1]$. We assume $\alpha \leq 1/2$, since the case $\alpha > 1/2$ is analogous. Let $\beta = \max(\alpha, 3/7)$. We repeat the sifting process as before, except that we find the β-partition $V = V_\beta^- \cup V_\beta^+$. For each set R_i, we now set $R_i^- = R_i \cap V_\beta^-$ and $R_i^+ = R \cap V_\beta^+$ and define the sets R^j, V^j, S^j as before for $j = --, -+, ++$.

Let $\gamma = |V^{--}|/|V| = |S^{--}|/|S|$. For the sake of simplicity, we ignore floors and ceilings for the moment. It is not hard to see that we have $|V^{-+}| = 2(\beta - \gamma)|V|$ and $|V^{++}| = (\gamma + 1 - 2\beta)|V|$. Thus $|S^{-+}| = 2(\beta - \gamma)|S|$ and $|S^{++}| = (\gamma + 1 - 2\beta)|S|$. Using the algorithm described above, we find, in $O(\Delta|S|)$ time, the largest node s^- in S^{--} and the smallest node s^+ in S^{++}, respectively. Let $S^{-+} = S_1 \cup S_2 \cup S_3$, where S_1 contains the nodes in S^{-+} less than or equal to s^- and S_3 contains the nodes in S^{-+} greater than or equal to s^+. We can find these sets using the Blum et al. [1] linear-time selection algorithm because the nodes (and their associated weights) in S^{-+} are known explicitly.

Let $A = S^{--} \cup S_1$, $\delta = |A|$, and $Z = S^{-+}$. If $|Z| \geq \alpha|S|$, we set $\rho = \alpha|S|/|Z|$, and using the standard linear-time selection algorithm, we find a ρ-partition $Z = Z_\rho^- \cup Z_\rho^+$. We now prove that there is always one of the sets $A, S - A, Z_\rho^+$ whose nodes we can remove from S, because we can assume that they are in one of the sets of the α-partition. Moreover, we prove that the set we remove contains at least 1/7th of the nodes in S.

First, note that each node in $S - A$ has weight at least as large as any node in A, so if $|A| \geq \alpha|S|$, then we place the nodes in $S - A$ in S_α^+ and reduce the problem to finding the $\alpha(|S|/|A|)$-partition of A. Symmetrically, if $|A| \leq \alpha|S|$, we place the nodes in A in

S_α^- and reduce the problem to finding the $(1-\alpha)(|S|/(|S|-|A|))$-partition of $S-A$. A similar argument applies to removing the nodes in Z_ρ^+ when we have $|Z| \geq \alpha|S|$, and we reduce the problem to finding the $\alpha(|S|/(|S|-|Z_\rho^+|))$-partition of $S-Z_\rho^+$. We now consider the sizes of the sets involved. If $\gamma \leq \beta/3$, we have $|Z| \geq 4\beta|S|/3 \geq \alpha|S|$ and $|Z_\rho^+| = (2(\beta-\gamma)-\alpha)|S| \geq (\beta-2\gamma)|S| \geq \beta|S|/3 \geq |S|/7$, since $\beta \geq \alpha$ and $\beta \geq 3/7$. Now suppose $\gamma \geq \beta/3$. We have $\gamma \geq 1/7$, so $|A| \geq |S^{--}| \geq |S|/7$ and $|S-A| \geq |S^{++}| \geq |S^{--}|$. Thus, in all cases, there is a set of size at least $|S|/7$ that can be removed, and we have reduced the problem to a realizable set of size at most $6|S|/7$ in $O(\Delta|S|)$ time.

It is easy to use the above ideas to compute, in $O(\Delta|S|)$ time, the rank in S of any node x, as well as find $S^{\leq x}$ and $S^{>x}$. Moreover, computing $|S|$ is trivial from the node-group list for S. Combining these observations yields a $D\Delta$ coarse-selection system for any realizable set in layer $h+1$, where the constant D is independent of h. It is interesting to note that the largest portion of D is a result of applying Theorem 4.2 to merge the selection system for the singleton node groups with the selection system for the larger node groups.

The arguments above yield an $O(D^h)$ coarse-selection system for realizable sets in layer h. By dividing all the original weights by the smallest weight, we may assume that they lie between 1 and σ, and hence we must process at most $\lceil \lg\sigma + 1\rceil$ layers before reaching the point where the worklist contains only crossable nodes. At this point the weights are within a factor of two, we have an $O(D^{\lg\sigma}|S|) = O(n)$ coarse-selection system, and we can apply Theorem 3.1 to determine the levels of these nodes, which we then use to determine the levels of the original weights.

5. Hardness results.

We begin with a simple hardness result that shows that constructing the intermediate lmcp tree produced by Hu–Tucker-based algorithms in any model of computation is at least as difficult as sorting in that model. We also give a more complicated reduction from sorting to any algorithm which computes enough partial information about the lmcp tree. This partial information is something we expect any region-based method must compute.

5.1. Finding the lmcp tree.

We will need the following simple lemma, whose proof we omit since it follows immediately from the observations made at the beginning of the proof of Theorem 3.1.

LEMMA 5.1. *Let $x_1, x_2, \ldots, x_n$ be distinct real numbers drawn from $[2,4)$. Let $y_i = \frac{1}{2}x_{\lceil i/2\rceil}$, for $i = 1\ldots 2n$. If $(y_1,\ldots,y_{2n})$ is given as input to any lmcp-finding algorithm, the set of the first n lmcps found, disregarding order, will be*

$$\{(y_1,y_2),(y_3,y_4),\ldots,(y_{2n-1},y_{2n})\}.$$

THEOREM 5.2. *We can reduce sorting sequences of size n to finding the lmcp tree in $O(n)$ time.*

Proof. Assume n is even. Let $x_1, x_2, \ldots, x_n$ be drawn from $[2,4)$. Define the y_i as above and consider the behavior of some lmcp-combining algorithm on the input sequence $y_1,\ldots,y_{2n}$. According to Lemma 5.1, after n lmcps have been combined, there will be n crossable nodes in the worklist with the weights $x_1,\ldots,x_n$. The only lmcp in the list is the smallest pair of nodes in $\{x_1,\ldots,x_n\}$ that combine to form a new node with weight at least 4. The next lmcp will be the second smallest pair of nodes from $\{x_1,\ldots,x_n\}$ and so on. Hence the next $n/2$ lmcps found sort $\{x_1,\ldots,x_n\}$ by pairs. Moreover, the fully sorted order of the x_i can be recovered from the lmcp tree (independent of how it was constructed) by searching the tree depth first and always

searching the least-weight subtree first, since the nodes corresponding to $\{x_1, \ldots, x_n\}$ will be encountered in sorted order. This shows that sorting can be reduced to finding the lmcp tree in $O(n)$ time. □

5.2. Region-based methods. In light of the linear-time algorithm for the constant factor case, it is natural to look for an $o(n \lg n)$-time method based on region processing. As before, we would hope to avoid determining all the lmcps but still determine the leaf levels in the lmcp tree. The wallflower is the difficult case to handle because it seems necessary to know explicitly which node it pairs with (as this increases the level of the leaves of this node by one). In particular, the wallflower may pair with the lmcp formed from the two smallest nodes in its region, and so it seems necessary that this information be easy to find for every region considered. We will say that an alphabetic tree-finding algorithm is *region based* if, from the information it computes, it is possible in $O(n)$ time to determine, for the smallest two nodes at each level, the set of leaves in the subtree of the lmcp tree rooted by each of these nodes. Note that this information is easy to compute if regions in the worklist are processed by increasing category order and the smallest two nodes are explicitly found in every region processed. This is because the smallest two nodes at each level in the lmcp tree are the smallest two nodes for some region, and we can easily keep track of the eventual level of the pair of smallest nodes for every region and pick the smallest pair at every level. The following theorem provides an $\Omega(n \lg n)$ lower bound in all models of computation for which an information-theoretic argument can be applied.

THEOREM 5.3. *Any region-based algorithm for finding alphabetic trees can be used, with $O(n)$ additional work, to sort sequences possessing a particular structure. Moreover, the number of distinct orderings among sequences with this structure is $2^{\Omega(n \lg n)}$.*

Proof. We show the existence of a sufficently large class of input sequences, such that for any sequence in the class, a region-based algorithm determines the structure of the lmcp tree. The proof is completed by showing that, for these sequences, the sorted order can be determined from the lmcp tree in $O(n)$ time.

The input sequences we consider consist of approximately $\sqrt{n}$ regions, each containing about $\sqrt{n}$ nodes and such that the category of a given region is one more than the region on its left. We assume $n = k^2 + 3k + 4$, where k is a positive integer. The first region will contain weights with values in $[1, 2)$, the next $[2, 4)$, then $[4, 8)$, etc. Denote the jth value in the ith region by $y_{i,j}$. The first region will have $4k + 4$ weights; the remaining have $2(k-1), 2(k-2), 2(k-3), \ldots, 2$ weights, respectively. Note that $4k+4+2(k-1)+2(k-2)+\cdots+2 = k^2+3k+4$. Let $x_1 < x_2 < \cdots < x_{2^{k+1}}$ be real numbers in $[2, 4)$. The values for the $\{y_{i,j}\}$ will be determined from the $\{x_i\}$. As the proof depends on the crossability of nodes, the values come in pairs so that the leaf nodes initially combine in pairs (this will be proved in Lemma 5.4).

Consider the following recursively generated binary tree built from the $\{x_i\}$. If internal nodes are assigned the sum of the weights of their children, then it has the property that the left child of any node is always less than the right.

Figure 1 shows the tree built for $k = 3$. The tree built for $k = 2$ is the subtree rooted at the left child of the root. The tree for $k = 4$ has this tree as the left child of its root, with the right child of the root consisting of an arm with leaf weights $x_{17} + \cdots + x_{24}, x_{25} + \cdots + x_{28}, x_{29} + x_{30}, x_{31}, x_{32}$ from left to right.

The purpose of this tree is to assign values to the $\{y_{i,j}\}$. Randomly distribute consecutive pairs $(y_{1,j}, y_{1,j+1})$, $j = 1, 3, \ldots, 4k + 3$, among the $2k + 2$ lowest terminal leaves in this tree. For $j = 1, \ldots, 4k + 4$, let $y_{1,j}$ be half the weight of the leaf that

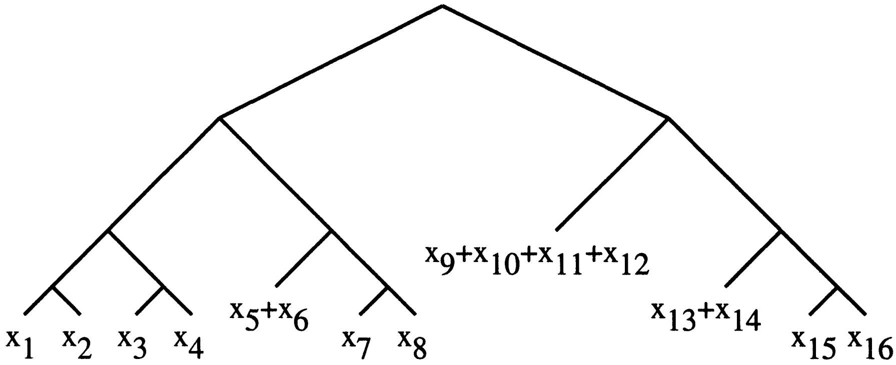

FIG. 1. *Tree generated from* $\{x_i\}$.

it is associated with. Then assign values to consecutive pairs of the $2(k-1)\{y_{2,j}\}$ by distributing them among the next lowest terminal leaves and so on. This new tree is called the *ordering tree* and is shown in Fig. 2. It records how the weights were assigned, and also their sorted order.

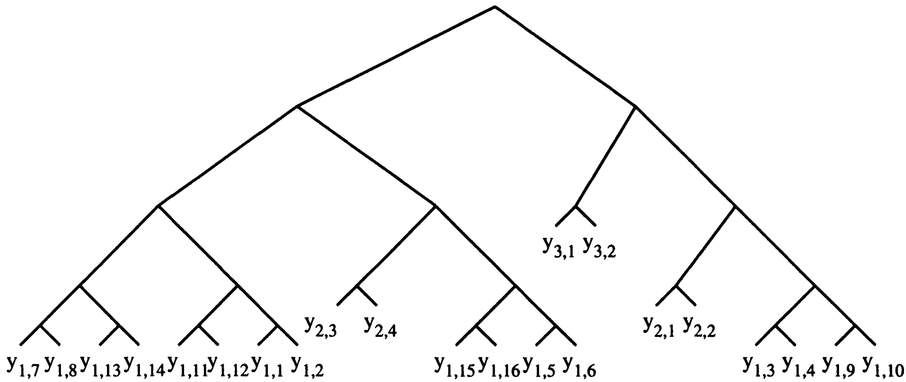

FIG. 2. *The ordering tree.*

The input weight list is as follows, with regions distinguished by height.

$$\begin{array}{lll} & & y_{3,1}, y_{3,2} \\ & y_{2,1}, y_{2,2}, y_{2,3}, y_{2,4} & \\ y_{1,1}, y_{1,2}, \ldots, y_{1,15}, y_{1,16} & & \end{array}$$

With an additional $O(n)$ time, we can determine the smallest two nodes at each level of the lmcp tree from the information computed by any region-based algorithm. To finish the proof, we first need a lemma.

LEMMA 5.4. *If the children in the lmcp tree are ordered according to weight, then the lmcp tree is isomorphic to the ordering tree.*

Proof. We may assume that we begin by combining all the lmcps in the lowest (largest-level) region. From Lemma 5.1 we know that since the weights come in

consecutive pairs of the same weight, these pairs will eventually form lmcps and combine, in agreement with the ordering tree. At this stage, the lowest region in the worklist consists of crossable nodes interspersed with some noncrossable ones, which again come in pairs. It is easily seen from the ordering tree that there is always an even number of crossable nodes smaller than the consecutive pairs of noncrossable nodes in the lowest region. Thus, we know that these crossable nodes will pair off first, and then the consecutive pairs of noncrossable leaf nodes will pair off as is shown in the ordering tree. It is clear from the ordering tree that this process continues and the lmcp tree, with every internal node's children ordered by increasing weight, is isomorphic to the ordering tree. Lemma 5.4 is proven. □

Consider the children of the root of the lmcp tree. By assumption, we know their weights and which leaves each child roots. By the lemma, the smallest node roots all the leaves on the left branch of the ordering tree, while the second smallest node will root all the leaves on the right branch. In time proportional to the number of leaves we find, we can traverse the right branch of our tree and find all the leaves and hence weights $\{y_{i,j}\}$ that are on the right branch of the ordering tree. Since there are only a constant number of leaves per level, we can afford to sort each level, and hence we begin sorting each of the regions in the initial input list. We now use this idea recursively on the subtree rooted at the smallest node of the root. This lets us find all the leaves in the right branch of the left branch from the root in the ordering tree. Again, we may sort the weights present at each level and append them to the beginning of the sorted region lists created previously. This will take time proportional to the number of nodes in this branch. By repeating this process, we will completely determine every input weight's location in the ordering tree, and, from this information, we can produce sorted lists of the weights in each region in the input. All this takes only $O(n)$ time to do.

The input sequences that we consider are subject to the restriction that the first $4k+4$ weights come before the next $2(k-1)$, which come before the next $2(k-2)$, and so on. The total number of different orderings of these sequences is

$$\begin{aligned}
&(2k+2)!(k-1)!(k-2)!\cdots(2)! \\
&> (\lfloor k/2 \rfloor !)^{\lfloor k/2 \rfloor} \\
&> \lfloor k/4 \rfloor^{\lfloor k/4 \rfloor \lfloor k/2 \rfloor} \\
&= \Omega(k^{\Theta(k^2)}).
\end{aligned}$$

Since $k = \Theta(n^{\frac{1}{2}})$, this number is $\Omega(n^{\Theta(n)}) = 2^{\Omega(n \lg n)}$. Theorem 5.3 is proven. □

6. Conclusions. In this paper, we have extended the ideas of Hu and Tucker for constructing optimal alphabetic binary trees. In particular, we have used their basic idea of *lmcp-tree construction* together with the new idea of *region processing* to give $O(n)$-time algorithms to solve the cases where the input weights are within a constant factor, or exponentially separated. The constant factor case makes use of a new technique for doing generalized selection in $O(n)$ time. We show that any natural method employing either the idea of lmcp-tree construction or the idea of region processing may force us to sort a substantial amount of the input. The basic question of whether there is a general $o(n \lg n)$-time algorithm for finding optimal alphabetic binary trees for this problem remains open. In fact, this question is open even for the highly restricted case where no wallflowers are encountered in the construction of the lmcp tree before the point where the worklist contains only crossable nodes.

REFERENCES

[1] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1972), pp. 448–461.

[2] A. M. Garsia and M. L. Wachs, *A new algorithm for minimum cost binary trees*, SIAM J. Comput., 6 (1977), pp. 622–642.

[3] E. N. Gilbert and E. F. Moore, *Variable length encodings*, Bell System Technical Journal, 38 (1959), pp. 933–968.

[4] T. C. Hu, D. J. Kleitman, and J. K. Tamaki, *Binary trees optimum under various criteria*, SIAM J. Appl. Math., 37 (1979), pp. 246–256.

[5] T. C. Hu and A. C. Tucker, *Optimal computer search trees and variable-length alphabetical codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[6] D. E. Knuth, *Optimum binary search trees*, Acta Inform., 1 (1971), pp. 14–25.

[7] B. M. Mumey, *Some new results for constructing optimal alphabetic binary trees*, Master's thesis, University of British Columbia, 1992.

[8] P. Ramanan, *Testing the optimality of alphabetic trees*, Theoret. Comput. Sci., 93 (1992), pp. 279–301.