SOME NEW RESULTS ON CONSTRUCTING OPTIMAL ALPHABETIC BINARY TREES

By

Brendan M. Mumey

B. Sc. Honors (Mathematics) University of Alberta

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1992

Computer Science
The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1Z1

Date: August 24, 1992

Abstract

This thesis studies the question of whether optimal alphabetic binary trees can be constructed in $o(n \lg n)$ time. Hu and Tucker [1] gave the first subquadratic algorithm in 1971, namely an $O(n \lg n)$ time algorithm. Optimal alphabetic binary trees have many applications and the question of whether an $o(n \lg n)$ time algorithm exists has remained an open problem in data structures for the last 20 years. 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 used. We introduce a new idea for finding optimal alphabetic binary trees which we refer to as *region-processing*. Region-processing methods have been successfully used to 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. Although the region-based method we give exhibits $o(n \lg n)$ time performance for many cases, we also show a reduction from sorting to it as well. It is possible that the ideas used in this reduction can be extended to yield a general $\omega(n)$ lower bound.

Table of Contents

List of Figures

Acknowledgement

I would like to thank my supervisor Maria Klawe for suggesting this interesting problem, and for her constant encouragement and help. In spite of running our department and running marathons, she always found time to meet with me. Thanks to David Kirkpatrick for suggesting improvements to the last chapter and for being my second-reader. I would like to acknowledge my fellow students and the pleasant research atmosphere at U.B.C. It's been a fun couple of years!

Brendan Mumey

August 1992.

Chapter 1

Introduction

The problem of finding optimal alphabetic binary trees can be stated as follows: Given a sequence of n weights $w_1, \ldots, w_n$, we wish to construct a 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 which satisfies this last requirement is said to be **alphabetic**. Although more general cost functions can be considered along the same lines as in the discussion to follow, the original cost function studied, and the one we will concentrate on is $\sum w_i l_i$ where l_i is the level of the i-th leaf node from the left in the tree. E. N. Gilbert and E. F. Moore [4] gave the first efficient ($O(n^3)$ time) algorithm for finding optimal alphabetic binary trees, using dynamic programming. Their method was improved by Knuth [8], in 1971, to $O(n^2)$ time for a more general problem where internal nodes are weighted as well. F. Yao also solves this problem in $O(n^2)$ time with her more general method for speeding up dynamic programming [5].

The first $o(n^2)$ time solution came in 1971, when T. C. Hu and A. C. Tucker [1] gave an $O(n \lg n)$ time algorithm. Their original implementation was $O(n^2)$, but this was improved to $O(n \lg n)$ by Knuth, using a better data-structure, and is mentioned in the original Hu-Tucker paper. 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. (The lower bound model will be discussed in Chapter 5.) Modified $O(n \lg n)$ algorithms for finding optimal alphabetic binary trees, based on the ideas of Hu and Tucker, but with simpler proofs, have been given by A. M. Garcia and M. L. Wachs in 1977 [3], and Hu, D. J. Kleitman and J. K. Tanaka in 1979 [2]. The only recent progress on this problem has been made by P. Ramanan [6] 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 trees in these cases.

As with Huffman trees, one application of optimal alphabetic binary trees is to build efficient codes. Given an optimal tree, one can build codewords for the words at terminal nodes according to how they are reached from the root node of the tree. This is done by writing a "0" for each left branch taken, and a "1" for each right. Although not as efficient as Huffman codes, alphabetic codes have the possibly desirable property that that they preserve order. If one word comes before another in the original order then the codeword for the first will be smaller than the codeword for the second.

Another interesting application is that of finding optimal binary search trees. This is the problem of devising efficient tests to differentiate a number of objects which may occur at different frequencies. An example mentioned in [1] is the problem of determining which of set of n lengths a particular object is assuming that you can only decide whether a particular object is longer or shorter than a given length and that the frequencies are given. (Imagine logs of different lengths to be stacked in different piles according to length.) To solve this problem, write down the lengths present in sorted order and create an external node for each one. Assign each node a weight corresponding to the frequency with which that length appears. An optimal alphabetic binary tree provides a classification strategy which minimizes the expected number of length comparisons necessary to identify the length of an object.

We begin by reviewing the current Hu-Tucker based algorithms, which are unified around the idea of building an optimal intermediate tree on the weights, from which the final optimal alphabetic tree is subsequently constructed. A simple observation will show that the construction of the intermediate tree requires at least as much time as sorting in the given model of computation. An alternative approach is considered in which input weights $\{w_i\}$ are first classified according to their order of magnitude. Define the **category** of a node of weight w to be $\lfloor \lg w \rfloor$. Maximal length **regions** of consecutive weights in the same category are simultaneously considered. Using this idea, we give $O(n)$ algorithms for the case when all the input weights are within a constant factor of one another, and the case where they are exponentially separated, in the sense that there are only a constant number of weights between 2^k and 2^{k+1} for any k. The $O(n)$ time algorithm uses a new result of Maria Klawe for performing generalized selection in linear time. Unfortunately, we are not able to give a general $O(n)$ algorithm based on region-processing. In fact, we show a reduction from sorting to a large class of region-based algorithms. We conclude with some remarks about the ideas we have introduced.

Chapter 2

Review of Current Algorithms

In this chapter we describe the basic Hu-Tucker algorithm and its variants. All Hu-Tucker based methods begin by building an intermediate tree on the input weight sequence. The levels of the weight-carrying leaves in this intermediate tree are recorded and then used to build an alphabetic tree whose weights are at the same level. Thus the cost of the alphabetic tree is the same as the cost of the intermediate non-alphabetic tree. The intermediate tree is proved to have optimal cost in a class of trees which contains all alphabetic trees, so it follows that the alphabetic tree constructed is optimal. Some of the proofs given are nearly identical to those given in [2], while others have been simplified. We feel that the proofs given are simplest available for what is quite a complicated argument.

The Hu-Tucker algorithm begins with a list of leaf nodes containing the weights $w_1, \ldots, w_n$ in order. This list is called the **work list** and is used to determine how nodes combine to form the intermediate tree. Nodes in the work list are designated either **crossable** or **noncrossable**. This affects which nodes may pair off together. Initially all nodes are noncrossable. When two nodes are paired off, the resulting internal parent node is designated crossable. The weight of the parent node is assigned the sum of the weights of its children. The nodes that paired off are removed from the work list, and the new parent node is put where the left child was. We say that two nodes in the list are **compatible** if they are adjacent in the work list, or if all the nodes which separate them are crossable. The symbol v_x will refer to a node in the work list and w_x will refer to its weight. The **level** of a node v_x in the tree, denoted by l_x, is length of the shortest path from v_x to the root of the tree. Define an order of the nodes in the work list by $v_x < v_y$ if $w_x < w_y$ or if $w_x = w_y$ and v_x is to the left of v_y in the list. Note that this ordering is transitive. A pair of nodes (v_a, v_b) are 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.

The first condition says that v_b is the best node for v_a to pair with, and the second says that v_a is the best for v_b to pair with. We refer to the new node formed after they pair as v_{ab}.

Only lmcps are combined. When only one node remains it will be the root node in the intermediate tree mentioned above. From now on we will refer to this intermediate tree as the lmcp tree. In general, the construction of the lmcp tree is not unique; the lmcps can combine in different orders. We now sketch a typical implementation of the Hu-Tucker algorithm.

1. Given the initial nodelist,

   $$(v_1)\ (v_2)\ (v_3)\ \ldots\ (v_n)$$

   form sorted priority queues of nodes which are compatible with each other. (See Figure 2.1.)

2. Since all nodes are compatible within a given priority queue above, the two smallest nodes at the top of the queue will be the only candidates for being an lmcp. To find an lmcp we can use a stack-based method. Beginning with the leftmost queue, maintain a pointer to the current queue being considered. By checking neighboring

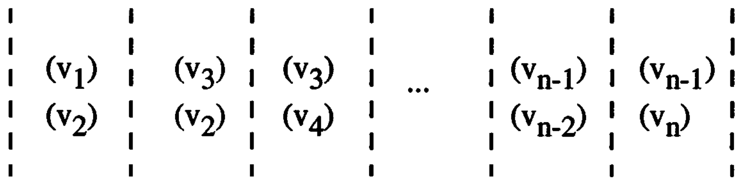


Figure 2.1: Sorted list of priority queues

regions, we can determine in $O(1)$ time whether or not the current pair is an lmcp. If it is, combine the two nodes and place the resulting node in the position previously occupied by the leftmost node. Move the pointer back to the previous queue, as it may now contain an lmcp. Otherwise move the pointer forward.

It will often happen that combining two nodes will allow several queues to merge. If we choose the appropriate data-structure for representing the queues, then merging the queues can be done in $O(\lg n)$ time. Leftist trees provide such a representation [10]. If we amortize the cost of finding lmcps over the entire course of the algorithm, they take only $O(1)$ time per lmcp to find. After each lmcp is combined we may require $O(\lg n)$ time to update the queue structures. Hence forming the lmcp tree in this way takes $O(n \lg n)$ time.

3. From the lmcp tree, it is easy to construct the optimal alphabetic tree in $O(n)$ time using $O(n)$ space. This is a simple process and is well described in the original Hu-Tucker and subsequent papers, so I will not describe it here.

2.1 The Garcia-Wachs Algorithm

An interesting modification to the basic Hu-Tucker algorithm presented above was made by A.M. Garcia and M.L. Wachs in 1977 [3]. Their modification rests on the observation that crossable nodes may be moved past smaller nodes, regardless of whether the latter are crossable or not. This follows from the fact that the smaller nodes will naturally combine and become crossable before the moved crossable node will be involved in an lmcp. By moving newly formed nodes judiciously, they ensure that all lmcp pairings are between adjacent nodes, and hence no information about crossability of nodes need be stored. Their algorithm begins with a list of the original input nodes:

$$(v_1)\,(v_2)\,(v_3)\,\ldots\,(v_n)$$

Define a consecutive pair (v_{i-1}, v_i) to be *right minimal* if it satisfies:

1. $w_{i-2} + w_{i-1} \geq w_{i-1} + w_i$ (when $i > 2$),
2. $w_{i-1} + w_i < w_{j-1} + w_j$ for all $j > i$.

First the right-most right minimal pair is located. Say this pair is (v_{i-1}, v_i). This will be an lmcp. Next the first (if any) entry to the right of v_i whose weight is greater than $w_{i-1} + w_i$ is found. Let this be v_{i+k+1}. Delete nodes v_{i-1} and v_i from the list and insert the new node $v_{i-1,i}$ into the list before v_{i+k+1}. The new list is then

$$(v_1)\,(v_2)\,\ldots\,(v_{i-2})\,(v_{i+1})\,\ldots\,(v_{i+k})\,(v_{i-1,i})\,(v_{i+k+1})\,\ldots\,(v_n).$$

If no such entry exists, then the new list is

$$(v_1)\,(v_2)\,\ldots\,(v_{i-2})\,(v_{i+1})\,\ldots\,(v_n)\,(v_{i-1,i}).$$

An $O(n \lg n)$ implementation of this idea is given at the end of their paper. It is due to Tarjan and makes use of some properties of balanced binary trees; namely that they can be used to represent lists such that most basic list operations on lists of length k require $O(\lg k)$ time.

2.2 Correctness of the Hu-Tucker Algorithm

We prove the correctness of the basic Hu-Tucker algorithm following the approach in [2]. Their proof is somewhat simpler than the original one given the original Hu-Tucker paper [1] and is slightly more general in that it proves the algorithm works for a class of cost functions. We have further simplified many of their proofs and present these simplifications in full.

There are two key issues addressed in the proof of the Hu-Tucker method. The first issue is feasibility. Are the leaf node levels in the lmcp tree realizable in an alphabetic tree? The second issue is optimality. Does this alphabetic tree have minimum cost?

Following the notation in [2], denote the cost of a tree T by $|T|$. Instead of simply adding the weights of the children to get the weight of the parent, they consider a more general combine function f, where the parent of nodes v_a and v_b is assigned the weight $f(w_a, w_b)$. The tree cost and combine functions must satisfy the following properties:

P1 $|T|$ cannot increase if any of its weights is replaced with a smaller weight.

P2 For all v_a, v_b, and v_x,

$$f(w_a, w_b) = f(w_b, w_a),$$

$$w_a \leq f(w_a, w_x)$$

and

$$w_a \leq w_b \iff f(w_a, w_x) \leq f(w_b, w_x).$$

P3 If $v_b < v_c$ and $l_b \geq l_c$, then interchanging v_b and v_c cannot decrease the cost of the tree.

P4 Let T be a tree with n nodes in which v_a and v_b are combined. Let T^* be the tree of $n-1$ nodes resulting from the combination, with the father of v_a and v_b

having weight $f(w_a, w_b)$. Then there exists a function $F(w_a, w_b)$ such that $|T| = F(w_a, w_b) + |T^*|$.

P5 The functions f and F must satisfy the following conditions if $w_b \leq w_c$:

$$f(f(w_a, w_b)), w_c] \leq f(f(w_a, w_c), w_b)$$

and

$$F(w_a, w_b) + F(f(w_a, w_b), w_c) \leq F(w_a, w_c) + F(f(w_a, w_c), w_b).$$

If the tree cost and combine functions satisfy these properties they are said to be **regular**. From [2] some examples of regular cost functions are:

1. $F(w_x, w_y) = w_x + w_y = f(w_x, w_y)$
   $|T| = \sum w_i l_i$ (Hu-Tucker)

2. $F(w_x, w_y) = 0, f(w_x, w_y) = t(w_x + w_y), t \geq 1$
   $|T| = w(root) = \sum w_i t^{l_i}$ (power summation)

3. $F(w_x, w_y) = 0, f(w_x, w_y) = t \times \max(w_x, w_y)$
   $|T| = w(root) = \max w_i t^{l_i}$ (min-max)

4. $F(w_x, w_y) = 0$ and let g be any function for which $g(x) > x$, and is increasing for all relevant x in the range of the problem.
   $f(w_x, w_y) = \max(g(w_x), g(w_y))$
   $|T| = \max g(w_i) l_i$ (min-max)

2.2.1 Feasibility

To prove feasibility we consider a class of trees whose external node levels can be realized in an alphabetic tree and show that all lmcp trees belong to this class. We begin with a definition of a class of trees which includes alphabetic trees, and as we will see, also lmcp trees. Recall that two nodes are compatible if there are no noncrossable nodes between them in the work list. A **construction** for a binary tree is just a list of node-pairings, which if performed in order, will construct the tree. A construction is said to be a **monotone construction** if the final levels of the nodes listed in the construction is a decreasing sequence. That is, if the tree has depth 5, then all nodes at level 5 are paired first, then all nodes at level 4, then all at level 3, and so on. Finally, a construction is said to be **compatible**, if all node-pairings occur between compatible nodes.

Definition 2.1 *Define C to be the class of all binary trees which have a monotone compatible construction. For a binary tree T in C, let c_T denote such a construction.*

We now give two complementary theorems about the class of alphabetic binary trees and the class C.

Theorem 2.1 *C includes all alphabetic binary trees.*

Proof: Let T be an alphabetic binary tree. The result is easily proved by induction on the height of T, with the base case of one node being trivial. Since T is binary, there must be an even number of leaves on the bottom (largest) level and they must come in pairs in the original list of weights. We can build a monotone construction for T recursively by appending a list of these pairs to the beginning of a monotone construction for the subtree of T with the bottom level nodes pruned. The monotone construction for the subtree exists by induction. □

Theorem 2.2 *The level sequence of the leaves of any tree T in C can be realized by an alphabetic tree.*

Proof: Let T be a tree in C. Assume that it is constructed via some c_T. To show that a tree can be made alphabetic, it suffices to show that the nodes at any given level in the construction occur in consecutive pairs in the work list. If this is true, one can pair off the nodes at the largest (bottom) level, then pair off the nodes at the next largest level and so on, as in the proof of the previous theorem. Consider the bottom level of T. If we can establish that nodes on this level come in consecutive pairs then we will be done, and the general result will follow by induction. If the nodes at the bottom level are not in consecutive pairs, then there must be two nodes which pair up and are separated by a node at a smaller (higher) level in the tree. But this node cannot be crossable, as we are following a monotone compatible construction. This is a contradiction as the nodes pairing would hence not be compatible. □

The next theorem shows that lmcp trees are in C, so from the last result we know that they can be made alphabetic. The proof of the next theorem relies on the following lemma which we give without proof. The proof can be found in [2].

Lemma 2.3 *Let v_{bc}, v_{ad} be nodes, where v_{bc} is crossed over in the formation of v_{ad}. Then $l_{bc} \geq l_{ad}$.*

Proof: Omitted. □

Theorem 2.4 *Any lmcp tree T' is in C.*

Proof: We can provide a monotone compatible construction for T' by listing all lowest level pairs, followed by all second lowest level pairs, etc. The only thing to show is that if these node pairs are combined in order, then all combinations are made between compatible nodes. It is easy to see that we can order the lowest level pairs so that no lowest level node prevents a pair from being compatible. The only issue is when a pair of nodes (v_a, v_d) are initially incompatible and only become compatible after a node v_{bc} at a higher (smaller) level has formed. Hence $l_b < l_a$. But by the last lemma, $l_{bc} \geq l_{ad}$, which implies that $l_b \geq l_a$. This contradiction proves that we can find an order to make all the lowest level node combinations compatible, and by repeating the argument, find a monotone compatible construction for T'. □

2.2.2 Optimality

This completes the proof of feasibility, so we now consider optimality. The crux of the proof that the lmcp tree T' is optimal in C is the next lemma.

Lemma 2.5 *Given an initial node sequence $v_1, \ldots, v_n$ where some nodes may be crossable, there is an optimal tree T^* in C such that all node combinations are between lmcps.*

Proof: The proof is by induction on the number of nodes present. The base case of two nodes is trivial. Assume the statement is true for any sequence of at most $n-1$ nodes; we need to prove it for sequences of n nodes. By the inductive hypothesis it suffices to show that there is is a construction of the tree in in which the first node pair is an lmcp.

If there is no optimal tree T^* in C, whose first combination is an lmcp then pick an optimal T^* and a construction c_{T^*} such that the first pair of c_{T^*} is (v_a, v_b) with $f(w_a, w_b)$ as small as possible among all constructions of optimal trees in C. After v_a and v_b combine, there will be $n-1$ nodes in the sequence. By the inductive hypothesis we may assume that the rest of the combinations in the construction are lmcps.

Since we are assuming that (v_a, v_b) is not an lmcp, there must be a node v_c compatible to one of these nodes (say v_b) with $v_a > v_c$. Further assume that v_c is minimal among all such nodes. Consider the second pair formed in c_{T^*}, which by assumption is an lmcp. There are four possible cases:

(i) The lmcp is (v_r, v_s) (two nodes other than v_{ab} and v_c).

(ii) It is (v_c, v_d).

(iii) It is (v_{ab}, v_c).

(iv) It is (v_{ab}, v_d).

We can immediately rule out case (iv); since v_c is compatible with v_b, it will be compatible with v_d, when v_{ab} is formed. From P2 and the fact that $v_a > v_c$, it follows that $v_{ab} > v_a > v_c$. Hence v_d would prefer to combine with v_c, and so (v_{ab}, v_d) could not have been an lmcp.

In cases (ii) and (iii), we can change the first two combinations in c_{T^*} to be

(ii) (v_b, v_c) followed by (v_a, v_d).

(iii) (v_b, v_c) followed by (v_{bc}, v_a).

In both cases, this would combine a smaller initial pair first, contradicting the the choice of c_{T^*}.

If case (i) happens, consider the sequence of combinations until v_c is combined. This will be either

$$(v_a, v_b), (v_{x_1}, v_{y_1}), \ldots, (v_{x_p}, v_{y_p}), (v_c, v_d)$$

or

$$(v_a, v_b), (v_{x_1}, v_{y_1}), \ldots, (v_{x_p}, v_{y_p}), (v_c, v_{ab})$$

where their intermediate combinations (v_{x_i}, v_{y_i}) do not involve v_c (or v_{ab} by the argument used to handle case (iv) above). We assume that v_{x_i} is to the left of v_{y_i} in the work list. We now change these sequences to

$$(v_b, v_c), (v_{x_1}, v_{y_1}), \ldots, (v_{x_p}, v_{y_p}), (v_a, v_d)$$

and

$$(v_b, v_c), (v_{x_1}, v_{y_1}), \ldots, (v_{x_p}, v_{y_p}), (v_{bc}, v_d)$$

As in cases (ii) and (iii), these new sequences give a construction of an optimal tree with a smaller initial pair. The only problem is that one of the intermediate pairs (v_{x_i}, v_{y_i}) might need to cross v_a, which may be noncrossable. We show this never happens.

Suppose that (v_{x_i}, v_{y_i}) is the first pair to cross v_a, and that v_a is noncrossable. Assume w.l.o.g. that v_{x_i} is to the left of v_a, and v_{y_i} and v_b are to the right of v_a in the initial sequence. We know that when the pair (v_{x_i}, v_{y_i}) combines, the node v_{y_i} is compatible with the node v_{ab} and thus at this point both v_{x_i} and v_{y_i} are compatible with v_c. This implies that v_{y_i} cannot be compatible with v_b initially since this would imply that $v_c < v_{y_i}$ and hence v_{x_i} would prefer to combine with v_c. Hence v_{y_i} becomes compatible with v_b sometime during the node combinations $(v_{x_1}, v_{y_1}), \ldots, (v_{x_{i-1}}, v_{y_{i-1}})$. We are specifically interested in which of these combinations are between nodes which block compatibility of v_{y_i} and v_b. Figure 2.2 shows a fairly general case of how compatibility of v_{y_i} and v_b can be blocked by noncrossable nodes. Crossable nodes are circles and noncrossable nodes are squares.

We note that all the nodes in the initial sequence between v_a and v_b are crossable, and that no combination is made that crosses v_b. This is because one of the nodes would then be compatible to v_b and thus also to v_c. Thus this node would prefer to combine with v_c. We now show how to define a sequence of node pairs $\{(v_{x_{j_k}}, v_{y_{j_k}})\}, k = 1 \ldots t$, with such that $1 \leq j_k < i$ and at least one of $v_{x_{j_k}}$ and $v_{y_{j_k}}$ is noncrossable. Furthermore this

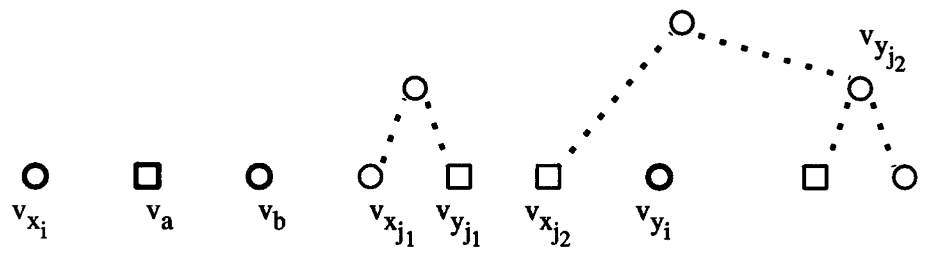


Figure 2.2: Compatibility blocking

sequence will have the property that $v_{y_{j_k}}$ and $v_{x_{j_{k+1}}}$ be compatible for $k = 1 \ldots t-1$ and that $v_{x_{j_1}}$ is compatible to v_b and $v_{y_{j_t}}$ is compatible to v_{y_i}. Let $(v_{x_{j_1}}, v_{y_{j_1}})$ be the closest pair to the right of v_b with at least one noncrossable node. Let $(v_{x_{j_2}}, v_{y_{j_2}})$ be the closest pair to the right of $v_{y_{j_1}}$ that combines after $(v_{x_{j_1}}, v_{y_{j_1}})$ with at least one noncrossable node, and so on. Eventually we will define a pair $(v_{x_{j_t}}, v_{y_{j_t}})$ so that $v_{y_{j_t}}$ is compatible with v_{y_i} and we are done. Notice that this last pair may cross over the location of v_{y_i}, as is shown in the diagram above. Local minimality of combining pairs together with the compatibility requirements imply that $v_c < v_{x_{j_1}} < v_{x_{j_2}} < \ldots < v_{x_{j_t}} < v_{y_i}$. Since v_c is compatible to v_{x_i} just before the pair (v_{x_i}, v_{y_i}) forms, this pair cannot be an lmcp. This contradiction ensures that no lmcp combination ever crosses v_a, and thus finishes the proof. □

We require one more lemma which we state without proof. The proof can be found in [2].

Lemma 2.6 *Given a node sequence consisting of crossable and noncrossable nodes, then combining any lmcp in the sequence will not decrease the value of the smallest node compatible to any other node in the sequence. In particular all other lmcps not combined* *will remain lmcps.*

Proof: Omitted. □

We can now state and prove the final theorem.

Theorem 2.7 *Given an initial node sequence $v_1, \ldots, v_n$ of noncrossable nodes, an optimal tree in C for any regular cost function can be built by successively combining lmcps in any order.*

Proof: We know from Lemma 2.5 that there exists a sequence of lmcp combinations c_{T^*} which constructs an optimal tree in C on these nodes. Assume that sequence is $(v_{x_1}, v_{y_1}), \ldots, (v_{x_m}, v_{y_m})$. By Lemma 2.6, no lmcp can prevent another from combining, hence all will eventually combine and an optimal tree will be built. □

This last theorem proves the correctness of all optimal alphabetic tree-finding algorithms which are based on combining lmcps to find a level assignment for the external nodes, and then building the alphabetic tree. The algorithms we present in the subsequent chapters are all based on this method.

Chapter 3

Faster Methods?

For certain classes of inputs, linear time algorithms for finding optimal alphabetic binary trees exist. In this section we will present simple methods for some of these cases. We will also introduce a new idea for finding optimal alphabetic binary trees in $O(n \lg n)$ time, which to some extent is a hybrid of the two simple methods presented. We call methods which use this idea **region-based**. Although all region-based methods will later be shown to have at least the time complexity of sorting in the model of computation being used, there are some interesting speed-ups which can improve their performance substantially in a number of cases.

3.1 Some Simple Cases Solvable in $O(n)$ Time

The first special case considered is that where the values $w_1, \ldots, w_n$ in the initial weight sequence are all within a factor of two. That is, there is a real number c such that $c \leq w_1, \ldots, w_n < 2c$. The next theorem provides the basis for a straightforward linear-time algorithm given as a corollary afterwards.

Theorem 3.1 *Given a sequence of n crossable nodes which 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*

$[\lg n] + 1$ *and the others will be at level* $[\lg n]$.

Proof: We note that 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 be $v_1, \ldots, v_n$ and let c be a real number such that $c \leq w_i < 2c$ 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, a solitary original node less than $2c$ will remain. It will be the largest node present in the original sequence and will form an lmcp with the smallest available newly formed node. When n is even the largest node present in the original sequence will merge with another original node. Regardless of whether n is odd or even, the last (there may be more than one) largest node will merge during the $[(n+1)/2]$-th lmcp pairing. At this stage there will be exactly $[n/2]$ nodes present and they will be within a factor of two, as is easily observed below:

If n is even then it is clear that the $[n/2]$ nodes present will be within a factor of two since the weight of each new node is the sum of the weights of exactly two original nodes. If n is odd, take $c' = w_i + w_j$, the smallest value 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 $2c'$. Let v_k be the single remaining node. It will form an lmcp with the node v_{ij}. Since the original weight sequence was within a factor of two, $w_k < w_i + w_j = c'$. So the weight of the next node formed, $v_{ij,k}$, will equal $w_i + w_j + w_k$ and hence be less than $2c'$. Thus all $[n/2] = (n-1)/2$ nodes present fall strictly between c' and $2c'$.

We note that when n is even all leaves have had their level increase by exactly one. When n is odd, the leaves associated with the node which paired with the largest node have had their level increase by exactly two; all others (including those of the largest node) will have increased by exactly one. If the nodes were presented in ascending order we would have the case illustrated in Figure 3.3.

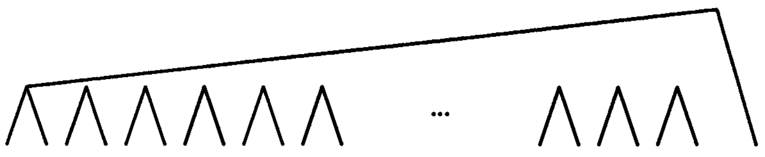


Figure 3.3: Odd number of nodes present

The crucial observation for this proof is that if n is odd, then the largest node present in the sequence of $\lceil n/2 \rceil$ newly formed nodes will be unique. If the largest node is v_b with weight w_b, then $w_i \leq w_b$ for nodes v_i in the original sequence. Suppose that v_b merges with v_{kl} the smallest of the newly formed nodes. Since the weight of v_{kl} is $w_k + w_l > 2c$, it follows that $w_j < w_{kl}$ for nodes v_j in the original sequence. So $w_i + w_j < w_{kl} + w_b = w_{b,kl}$ for any two nodes v_i and v_j in the original sequence. Hence the largest node is unique. Since it is unique, its parent will also be the largest node in next sequence of newly paired-off nodes. This means that although the leaves associated with the node that paired with the largest node had their levels increase by two after one stage of the algorithm, they will always be rooted by a largest node and so will only have their levels increase by one during all remaining stages. This argument also applies to nodes which have their levels increased by two at later stages of the algorithm. Thus when the tree is finished, the level of any two leaves can differ by at most one. As any lmcp-combining strategy is guaranteed to produce a tree of optimal weight, we can be sure that the smallest values are at the bottom (largest) level. A certain number, $2x$, of the smaller nodes will pair off and occupy the bottom level $\lfloor \lg n \rfloor + 1$, and the remaining $n - 2x$ larger nodes will get their own leaves. We require $x + n - 2x = 2^{\lfloor \lg n \rfloor}$, or $x = n - 2^{\lfloor \lg n \rfloor}$. $\square$

Corollary 3.2 *There exists 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.*

From chapter one, we know that it suffices to find the levels of the leaves in the lmcp tree. We know that this level sequence can be realized by an alphabetic tree which can be constructed in $O(n)$ time. In point form, the algorithm for finding the levels of the leaves in the alphabetic tree is as follows:

1. Begin with the work list containing the original input sequence. Note that all nodes are noncrossable.

2. Use a stack-based method to find lmcps and pair them off, removing them temporarily from the work list. This process will continue until there are zero or one nodes left in the work list. If a single node remains (n is odd), scan through the list of newly formed crossable nodes to find the smallest pair. Pair this node with the single noncrossable node.

3. At this stage we have $m = [n/2]$ crossable nodes left. 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 directly find the levels of every leaf in the lmcp tree for the m crossable nodes present by the preceding theorem. We know that if we continue to build up the lmcp tree that we will get a balanced binary of tree of depth $[\lg m]$. To find the $2(m - 2^{[\lg m]})$-th weight in the list we can use the well-known linear time selection algorithm based on median-finding [7]. We then scan through the list and find

$2(m - 2^{\lfloor \lg m \rfloor})$ nodes whose weights are less than or equal to the $2(m - 2^{\lfloor \lg m \rfloor})$-th weight, and assign them the level $\lfloor \lg m \rfloor + 1$. The remaining nodes are assigned level $\lfloor \lg m \rfloor$. In an additional $O(n)$ time, we can compute the levels of the nodes in the original input sequence.

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

The next special case that is easy to solve in linear time happens when the weights $w_1, \ldots, w_n$ in the input sequence are exponentially spread out; i.e. none of the nodes are within a factor of two. Formally, for all weights w_i and w_j, if $w_i \leq w_j$ then $|w_j/w_i| \geq 2$.

A simple bottom-up approach to building the optimal alphabetic tree on an exponentially spread out sequence is just to use the basic stack-based method, but ignore crossability of nodes (new nodes are still noncrossable). This reduces the running time to $O(n)$ time. The following lemma justifies this.

Lemma 3.3 *If for all input weights w_i, w_j, $w_i \leq w_j \Rightarrow |w_j/w_i| \geq 2$, no crossable node will ever be crossed over.*

Proof: In view of obtaining a contradiction, assume that the first lmcp combination that crosses over a section of crossable nodes occurs when v_a and v_b combine and cross over the nodes $v_{x_1y_1}, \ldots, v_{x_my_m}$ in the work list, and that of these in-between nodes, $v_{x_iy_i}$ was the last to form. Then v_a was compatible to v_{x_i} and v_b was compatible to v_{y_i} when the node $v_{x_iy_i}$ was formed. This means that $w_a > w_{y_i}$ and $w_b > w_{x_i}$. So $\max(w_a, w_b) > \max(w_{x_i}, w_{y_i})$. Let the weight of the largest original node present in the leaves rooted by either v_{x_i} or v_{y_i} be u. Since the leaves of v_{x_i} and v_{y_i} are disjoint and leaves decrease in weight geometrically by a ratio of a least two, it follows that $w_{x_iy_i}$, which equals the sum of the weights of the original nodes rooted by $v_{x_iy_i}$, is less than $2u$.

On the other hand, $\max(w_a, w_b) > \max(w_{x_i}, w_{y_i}) > u$. Since the weight u is not among the weights of the original nodes of v_a or v_b, there must be a larger original node present in order for the sum of the weights to exceed u. Such a node must have weight $v > 2u$ by assumption. Thus $\max(w_a, w_b) > 2u > w_{x_iy_i}$. This contradicts the pair (v_a, v_b) being an lmcp since $\min(v_a, v_b)$ would prefer to merge with $v_{x_iy_i}$. □

Hence it is not necessary to maintain information about crossability as it will never be used. It is relatively straightforward to see that ignoring crossability speeds up the standard stack-based method to linear running time. The speed-up comes from the fact that there are no longer crossable nodes lists to maintain and merge. We now give a formal description of the algorithm.

Let the input sequence be

$$\begin{array}{cccccccc} v_1 & v_2 & v_3 & v_4 & \ldots & v_{n-1} & v_n \\ & \Uparrow & & & & & \\ & P & & & & & \end{array}$$

with the stack pointer P initially pointing to the first node in the work list. The basic step consists of checking if the node above P and its neighbor to the right forms an lmcp; this just involves checking the next two nodes to the right as we will see shortly. If the pair is an lmcp it is combined, the new node inserted at P, and P reset to the previous node. If not, P is advanced one node to the right. Thus in each step we either combine two nodes or advance P. Assume the current work list is always $v_1, \ldots, v_n$ and that we are examining the node (v_i, v_{i+1}). This means that the nodes $(v_1, v_2), \ldots, (v_{i-1}, v_i)$ were not lmcps. Hence $v_1 > v_3, v_2 > v_4, v_3 > v_5, \ldots, v_{i-1} > v_{i+1}$. Recombining these inequalities shows that $v_1 > v_3 > v_5 > \ldots > v_{2[i/2]+1}$ and $v_2 > v_4 > v_6 > \ldots > v_{2[(i+1)/2]}$. In light of the fact that nodes are assumed noncrossable, we only need to compare v_i and

v_{i+2} to see if (v_i, v_{i+1}) is an lmcp. If $v_i \leq v_{i+2}$, it is. If we reach the end of the list and are checking (v_{n-1}, v_n) we see that automatically this pair will be an lmcp. After combining it, removing $v_n - 1$ and v_n from the list, and placing the new node in the last position, $n-1$, P will retreat to check (v_{n-3}, v_{n-2}) for local minimality. Either it is an lmcp or we need to move forward once to find one.

We now calculate the running time of this method, by counting the maximum number of comparisons necessary to build the tree. Let $T(n,p)$ be the maximum number of comparisons needed when there are n nodes and P is at position p in the list. We make the following observations on T:

1. $T(2,1) = 0$ (automatic pairing)

2. $T(n,p) = \max \begin{cases} T(n,p+1)+1 & \text{(no lmcp above P)} \\ T(n-1,1)+1 & \text{(lmcp above P)} \end{cases}$ for $p < 3$.

3. $T(n,p) = \max \begin{cases} T(n,p+1)+1 & \text{(no lmcp above P)} \\ T(n-1,p-2)+1 & \text{(lmcp above P)} \end{cases}$ for $2 < p < n-1$.

4. $T(n,n-1) = T(n-1,p-2)$ (automatic pairing)

It is easy to verify by induction that $T(n,p) \leq 3n - p$. Hence the total time cost of the algorithm will be $3n$ for the cost of comparisons plus an additional cost of $n-1$ for forming the $n-1$ new nodes between lmcps.

An alternative approach to finding an optimal alphabetic tree on n weights exponentially spread out is to build the tree top-down. This is based on the observation that placing the largest node as high as possible in the tree dominates over the consideration that other nodes will potentially be at larger levels. We leave the details to be worked out by the reader.

3.2 Region-based Methods

In this section we present a new method for finding optimal alphabetic binary trees. After the basic version has been explained, we will make observations on how to speed up its performance in special cases. The method will be referred to as region-based. Although it will later be shown that in the worst case it requires as much time as sorting, it exhibits $o(n \lg n)$ time performance for a significant variety of inputs. In particular, we are able to give $O(n)$ algorithms for the case when all the input weights are within a constant factor of one another, and the case where there are only a constant number of weights between 2^k and 2^{k+1} for any k. In all cases, its performance is at least as good as the Hu-Tucker algorithm. The method is based on the observation that nodes that are within a factor of 2 have approximately the same level. We begin with a complete description of the basic algorithm.

Input: a list of weights: $w_1, w_2, \ldots, w_n$

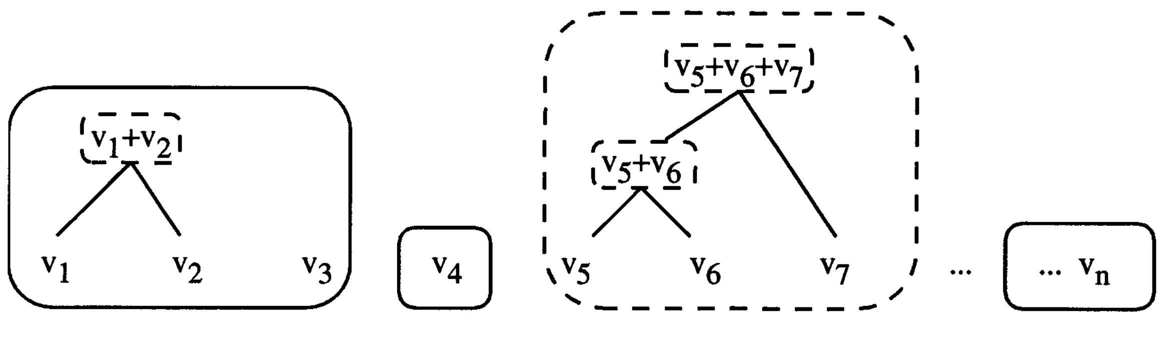
Output: a representation of an optimal binary tree built on these weights

Assume the input has been placed in a doubly-linked list called the work list. We define the category of a node v_x in the work list to be $[\lg w_x]$. We then re-scan the work list from left to right to build a new list (doubly-linked) of consecutive regions. A region is defined to be the maximal length list of consecutive nodes from the work list of the same category. It is represented as a 5-tuple $(c, p_{le}, p_{re}, p_{lm}, p_{rm})$ where c is the category of the region, p_{le} and p_{re} are pointers to the left and right endpoints of the actual region in the work list, and p_{lm} and p_{rm} are pointers to the minimum nodes in the region compatible to imaginary nodes to the immediate left and right of the nodes at p_{lm} and p_{rm} respectively. If, for example, the node at p_{le} is noncrossable, then $p_{lm} = p_{le}$. We have now partitioned the work list into regions:

$$\{v_1v_2\}\ \{v_3\}\ \{v_4\}\ \{v_5\ v_6\ v_7\}\ \ldots\ \{\ \ldots\ v_n\}$$

The algorithm proceeds by determining a certain region is suitable to process, and then combining nodes in that region. In some cases it will process several regions simultaneously. It then combines pairs of nodes in it (them) appropriately and in so doing, eliminates the old region(s). Newly formed nodes may then join adjacent regions or create new ones. If r is the number of nodes in the region processed, then we only require $O(r \lg r)$ time to do the entire operation (including the time spent to update the work list and region list). It is also the case that a region is examined in the region list at most twice until it is processed and eliminated. So the total cost of eliminating the r nodes and replacing them with $[r/2+1]$ pairs is still $O(r \lg r)$ time, if we amortize over the course of the algorithm. Hence we spend only $O(r \lg r)$ time to reduce the problem to size $[n - r/2]$; this yields an $O(n \lg n)$ time algorithm.

To illustrate this, after a few steps of the algorithm, it may be that we have the situation pictured in Figure 3.4.



Dashed lines indicate crossability

Figure 3.4: Region processing

3.2.1 Detailed description of the algorithm

We use a stack-based method to determine which regions to process. We keep track of node crossability and only combine lmcps. This ensures optimality by the results of the previous chapter.

Assume that at the start the region list is:

$$\begin{array}{lllllll} R_1 & R_2 & R_3 & R_4 & \ldots & R_{n-1} & R_n \\ \Uparrow \\ P \end{array}$$

The pointer P points to the region we are currently considering for processing. To determine if we can process the region above P we need to check adjacent regions in the region list. To describe this process assume we are in the general situation pictured below, with P below a region R_k, where $1 \leq k \leq m$:

$$\begin{array}{lllllll} R_1 & R_2 & \ldots & R_k & R_{k+1} & \ldots & R_m \\ & & & \Uparrow \\ & & & P \end{array}$$

Let $R_i < R_j$ iff the category of R_i is less than the category of R_j. The following invariant is maintained by the algorithm:

There exists a possibly empty set $J = \{j_1 < j_2 < \ldots < j_s\}$ *of nonconsecutive indices from the set* $\{1, \ldots, k-1\}$ *such that:*

1. R_{j_i} consists of a single non-crossable node for all $i = 1 \ldots s$
2. $R_1 > \ldots > R_{j_1 - 1} < R_{j_1} > R_{j_1 + 1} > \ldots > R_{j_2 - 1} < R_{j_2} > \ldots > R_{j_s - 1} < R_{j_s} > R_{j_s + 1} > \ldots > R_k$
3. $R_{j_i - 2} \geq R_{j_i}$ for all $i = 1 \ldots s$ where $j_i - 2 > 0$.

4. $|R_{j_i-1}|$ is odd

Depending on the adjacent regions to R_k, we either process R_k and possibly other regions or we advance P. In either case the invariant above is maintained. The cases are given below:

1. $R_k > R_{k+1}$: Move P under R_{k+1}.

2. $R_k < R_{k+1}, R_{k+1}$ consists of a single non-crossable node, and $R_k \geq R_{k+2}$: Move P under R_{k+2}.

3. Otherwise: In this case, we process R_k, and possibly other regions to the left. We begin by finding the maximum even value j such that for all $i = 0, 2, 4, \ldots, j$ we have $k - 1 - i$ in J and $R_{k-2-i} = R_k$. In the case that no such j exists we set $j = -2$. Let R_{le} refer to the node in the work list pointed to by the p_{le} pointer for region R. Define R_{re}, R_{lm}, and R_{rm} likewise. Now use a stack-based subroutine to process the following list of nodes:

   $R_{k-j-3_{rm}}, R_{k-j-2_{le}}, \ldots, R_{k-j-2_{re}}, R_{k-j-1_{le}}, R_{k-j_{le}}, \ldots, R_{k-j_{re}}, \ldots, R_{k_{le}}, \ldots, R_{k_{re}}, R_{k+1_{lm}}$

For example, when $j = -2$, we just have the list:

$$R_{k-1_{rm}}, R_{k_{le}}, \ldots, R_{k_{re}}, R_{k+1_{lm}}$$

The reason for keeping track of singleton regions, is that nodes in regions on either side of a singleton region may compete to combine with the singleton node separating them. This happens when $j = 0$ as is shown in Figure 3.5.

3.2.2 The region-processing step

The subroutine begins by placing pointers in the work list to delimit the region we are working on. This list will be a scratch list, and as we work on it we will remove nodes

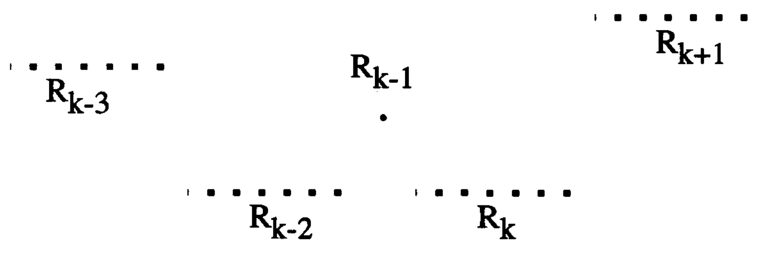


Figure 3.5: Typical region distribution

which are paired. We maintain a pointer Q in the scratch list that points to the current node being considered for pairing with its right neighbor. To illustrate this, let us assume the essentially general case when $j = 2$. For further convenience we will label the nodes in region R_{k-4} by $a_1, \ldots, a_p$, those in R_{k-2} by $b_1, \ldots, b_q$ and those R_k by $c_1, \ldots, c_r$. The initial set-up of the work list is:

$$\begin{array}{cccccccccccccc} \ldots & R_{k-5_{rm}} & a_1 & \ldots & a_p & R_{k-3_{le}} & b_1 & \ldots & b_q & R_{k-1_{le}} & c_1 & \ldots & c_r & R_{k+1_{lm}} \quad \ldots \\ & \Uparrow & & & & & & & & & & & & \\ & Q & & & & & & & & & & & & \end{array}$$

Some of the nodes in a, b and c may be crossable. The singleton regions R_{k-3} and R_{k-1} are not crossable and neither are the endpoint nodes $R_{k-5_{rm}}$ and $R_{k+1_{lm}}$.

We now use the fairly simple stack-based implementation of the Hu-Tucker algorithm to determine all the lmcps in the regions a, b and c, and pair them up. Since we know that all of the nodes in a given region, say a will pair up and form a consecutive block of crossable nodes, we can keep all newly formed nodes in a separate list, which we insert into the work list when all the original nodes have been combined. After this is done we can find the categories of all the new nodes and the new regions in this part of the work list with one scan over the newly created nodes.

If there were an odd number of nodes in one of the regions, say a, to begin with the last node will either merge with one of the newly formed nodes or with an endpoint or one of the singleton region nodes. After all the initial nodes have been merged, the list of newly formed nodes is inserted into the work list. The region list is also updated. The time complexity of the entire region-processing step is $O(r \lg r)$.

3.2.3 Time complexity of the basic region-based method

The proof that this method runs in $O(n \lg n)$ time is very similar to the proof that the stack based method in the beginning of the chapter ran in $O(n)$ time. The difference being that the stack maintained now contains regions, and the basic cost of processing a region of size m is $O(m \lg m)$. Let $S(n,p)$ be the maximum number of steps required by our method to build the lmcp tree on a list of n nodes, divided into some number number of regions, with the region pointer P at position p in the region list. We make the following observations on S:

1. $S(2,1) = 0$ (can automatically process region above P)
2. $S(n,p) = \max \begin{cases} S(n,p+1)+1 & \text{(cannot process region above P)} \\ S(n-[s/2],1)+O(s \lg s) & \text{(can process region above P)} \end{cases}$
   for $p < 3$.
3. $S(n,p) = \max \begin{cases} S(n,p+1)+1 & \text{(cannot process region above P)} \\ S(n-[s/2],p-2)+O(s \lg s) & \text{(can process region above P)} \end{cases}$
   for $2 < p <$ [last region].
4. $S(n,\text{[last region]}) = S(n-[s/2],p-2)+O(s \lg s)$
   (can automatically process region above P)

It is easy to verify by induction that $S(n,p) = O(n \lg n) - O(p)$, which implies that the running time of the method is $O(n \lg n)$.

Chapter 4

Speeding up Region-based Methods

There are many classes of inputs for which we can speed up the region-based method presented in the previous chapter. This is done by not combining every lmcp, which as we will see in the next chapter can be as time consuming as sorting. We begin by discussing a way of speeding up the region-based method for general inputs, and then consider the case where the inputs weights are within a constant factor of one another and the case where there is a bounded number of weights between 2^k and 2^{k+1} for every k. We refer to the input weights in these cases as being **bounded by a constant factor** and being **exponentially separated** respectively. The linear time algorithm for exponentially separated sequences is a simple observation on the behavior of the basic region-processing method, whereas the corresponding linear time algorithm for the constant factor case is very complicated. We only sketch the basic details of the method discovered by the author and his supervisor Maria Klawe.

4.1 Speeding up General Region-based Algorithms

The basic idea used is that instead of explicitly combining lmcps we only find the ones which will affect the levels of the leaves in the tree. For example, if the category of a given region is much less than that of its neighboring regions, we know that the nodes in the region will pair up and form a balanced subtree before pairing with any nodes in neighboring regions. Hence we can process the region using a linear time method by treating it as if it were the entire input sequence, as is shown in Figure 4.6.

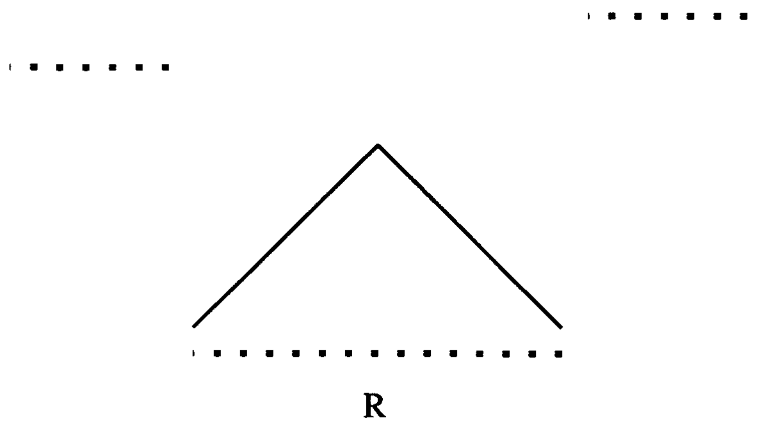


Figure 4.6: Can process R quickly

Unfortunately there are cases where our strategy still requires accurate knowledge of many of the lmcps which form in a given region. This happens when the categories of the adjacent regions are not too much greater than the region currently being processed. Newly formed nodes from the processed region recombine until they enter one of the adjacent regions. If the category of the adjacent regions was not much larger than that of the processed region many new nodes will enter it. For example, if the categories differ by two, and there were r nodes in the lower (larger level) region then $[r/4]$ nodes will enter the higher region. Although immediate knowledge of the exact weights of the entering nodes may not be necessary initially, it can happen that such information will be required later in order to properly determine future lmcps. This would be the case if, upon entering the new region, the input sequence is now within a factor of two. We know that at this stage a balanced tree will be built with some of the nodes occupying largest (lower) levels. It would then be necessary to have all the weight information because we have to assign a set of the smallest weights to these larger levels and some of the new nodes may well be among them. The region-based algorithm we give accurately determines the weights of all nodes entering a higher region; this strategy can save time if the gaps between the initial regions are large enough.

We follow the overall region-combining approach that we described previously. That is, we use a stack-based method to find regions to combine and then enter a subroutine to process individual regions. The difference will be in the region processing step. By checking the category of adjacent regions we can determine the height of the gap. If it is cost-effective to do so, we will use a different method than just successively combining lmcps until all the nodes enter one of the adjacent regions. We also sort the list of newly created crossable nodes into adjacent regions before merging them into adjacent regions. As we will see, this does not increase the time complexity of the method, as we spend at least this much time processing the region itself. (This is clearly the case if the region processing step given in chapter two is used; processing a region of size r cost $O(r \lg r)$ time.)

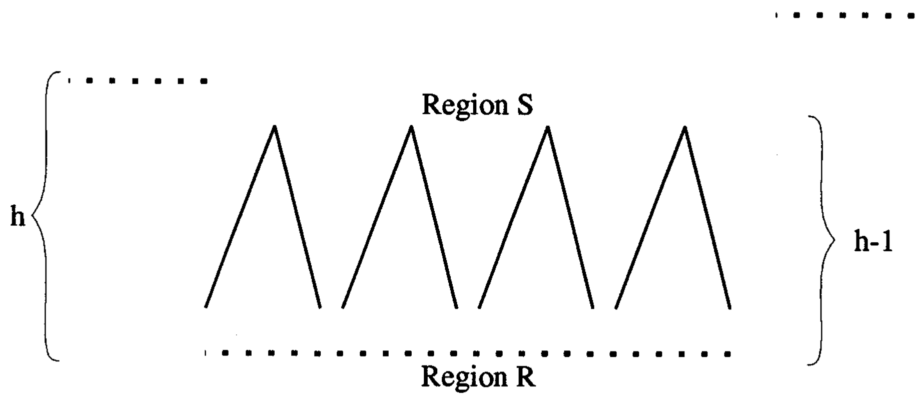


Figure 4.7: Gap between regions

Assume that there are r nodes in the middle region, R, which is the one currently being processed. If we were to follow the basic region-processing procedure, we would spend $O(r \lg r)$ time combining the nodes present to form a new region whose category would be one greater than the first, but would have half the nodes. This would continue until eventually the newly formed region would merge with one of the adjacent regions. The alternative is to use the idea of the linear time approach for weights within a factor of two to predict the values of the nodes in the last newly formed region S just before it merges with one of the neighboring regions. In this way, we will be able to construct S without computing any of the intermediate regions in between. Provided the gap height h is large enough, the time spent will be less than $O(r \lg r)$. Indeed if the gap is larger than $[\lg r]$ then we know that we will form a complete balanced binary on the region being processed, and that the root of this tree will later combine with a node in one of the adjacent regions. As was mentioned before we could thus use the algorithm given in the previous section to build this tree in linear time. We will begin with describing the method for predicting the nodes in the last newly formed region that would be formed by any lmcp-combining strategy and then we will provide a criterion on the gap height for its cost effectiveness.

Let the gap height between R and its lowest adjacent neighbor be h, as pictured in the previous diagram. We wish to find the nodes in S immediately below the lowest adjacent neighboring region without going through the expense of computing every region in between. Expand r as

$$r = 2^{a_1}(2^{a_2}(\ldots 2^{a_{k-2}}(2^{a_{k-1}}(2^{a_k}+1)+1)\ldots)+1).$$

This takes $O(\lg r)$ time. Note that $\sum a_i = [\lg r]$. This expansion will let us determine the partition of the original input sequence according to which nodes they descend from in S. By summing the weights of the nodes in a given partition, we compute the weight of their common ancestor in S. The next theorem shows how to find the partition of R based on the expansion of r. It assumes that all the nodes in R are crossable. If they are not all crossable then additional time must be spent in order to pair off all noncrossable nodes first. This will only require $O(r)$ time, since any sections of crossable nodes are in sorted order. It is not necessary to re-sort the crossable nodes afterwards.

Before we formally state a theorem about how R is partitioned, we will provide some nomenclature for describing the intermediate combination steps in the transitory regions formed between R and S. Assume that all nodes in R are crossable and that they are labelled v_1 to v_r. If r is even, then the lmcps we find and combine will be in order $(v_1, v_2), (v_3, v_4), \ldots, (v_{r-1}, v_r)$. At this point we will have formed a new region. Call this one r-step. If r is odd then the lmcps formed will be

$$(v_1, v_2), (v_3, v_4), \ldots, (v_{r-2}, v_{r-1}), ((v_1, v_2), v_r).$$

Also call this one r-step, even though two nodes have paired twice. This is pictured in Figure 4.8. In either case, after one r- step a new region is formed.

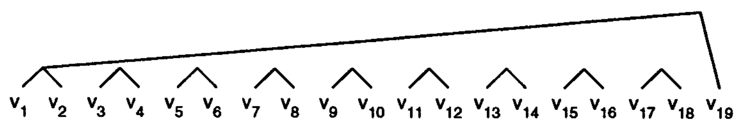

v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} v_{15} v_{16} v_{17} v_{18} v_{19}

Figure 4.8: One r-step

As was seen in the proof of (the first theorem of this section), the largest node v_{19} is a child of the largest node in the next region, and this node is unique.

After a number of r-steps the new region created will either have its smallest node at category $c - 1$, where c is the category of the nearest adjacent region or consist of a singleton. In both cases we must stop. In the former case we can no longer assume that all nodes present will combine with one another. This is because the largest node can now be at category c, and thus be at the same level as one of the adjacent regions.

For example this would happen if in the last diagram $w_1 = 2, w_2 = 2, w_{19} = 3.9$ and the category of one of the adjacent regions was 2. Assuming this is not the case, after a couple more r-steps we will have formed the forest shown in Figure 4.9. Notice that

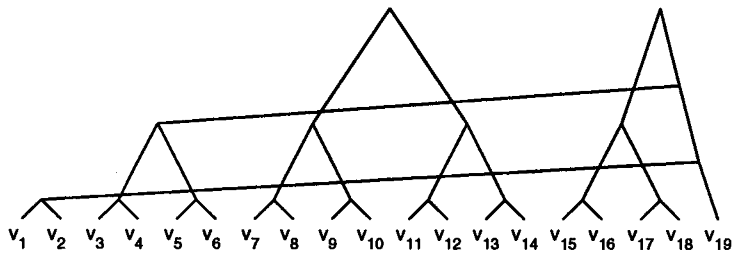

v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} v_{15} v_{16} v_{17} v_{18} v_{19}

Figure 4.9: Forest formed after three r-steps

that the smallest six nodes have increased their levels by 4, whereas all the others have increased their levels by 3. The next theorem shows it easy to predict these numbers.

Theorem 4.1 *Let* $r = 2^{a_1}(2^{a_2}(\ldots 2^{a_{k-2}}(2^{a_{k-1}}(2^{a_k}+1)+1)\ldots)+1)$ *and assume that* j *r-steps occur* ($j < [\lg r]$)*. Let* s *be maximal such that* $\sum_{i=1}^{s} a_i < j$*. Then there will be exactly*

$$c_{r,j} = \max \begin{cases} 2(2^{a_1} + 2^{a_1+a_2} + \ldots + 2^{a_1+\ldots+a_s}) & \textit{if } k > 1 \textit{ and } s \textit{ exists} \\ 0 & \textit{otherwise} \end{cases}$$

nodes in R *which have increased their level by* $j+1$ *and the remaining* $r - c_{r,j}$ *will have increased their level by* j*. Furthermore the nodes in the region formed after the* j*-th r-step will have weights in order,* $(w_{c_{r,j}+1} + \ldots + w_{c_{r,j}+2^j})$, $(w_{c_{r,j}+2^j+1} + \ldots + w_{c_{r,j}+2^j+2^{j+1}})$, $\ldots(w_{c_{r,j}+(m-1)2^j+1} + \ldots + w_{c_{r,j}+m2^j})$, $(w_1 + \ldots + w_{c_{r,j}} + w_{c_{r,j}+m2^j+1} + \ldots + w_r)$ *where* $m = [r/2j] - 1$.

Proof: The proof of this theorem mirrors the one given for Theorem 3.1. We know that in any r-step where the number of nodes present is odd, the smallest two nodes will combine to form a new node, and this new node will then combine with the largest node present, as was pictured in the last figure. We also know that this new node will be the sole largest node present in the next region. It was this observation which showed that any node would have its level increased by 2 during at most one r- step; after this, it would be a descendant of a largest node, whose levels only go up by one every r-step. The formal proof is by induction on k. The base case of $k = 1$ is simple; r is a power of two, and so there are never any leftover nodes to combine after any r-step. Assume that the theorem is true for all k up to $m - 1$. We need to show it works for $k = m$. Consider a_1.

If $a_1 = 0$, we know that there are an odd number of nodes present. Hence the two smallest nodes present will have their levels increased by 2 in the first r-step. The number of nodes which have their levels increased by two will thus be 2 plus twice the number of nodes in the next level which have their levels increased by two. This is because every node at the next level represents two in the original level. In the case $m > 2$, by induction this number is

$$\begin{aligned} & 2 + 2(2(2^{a_2-1} + 2^{a_2-1+a_3} + 2^{a_2-1+a_3+a_4} + \ldots + 2^{a_2-1+\ldots+a_s})) \\ = \; & 2(2^0 + 2^{a_2} + 2^{a_2+a_3} + 2^{a_2+a_3+a_4+\ldots+2^{a_1}+\ldots+a_s}) \\ = \; & c_{r,j}. \end{aligned}$$

If $m = 2$, note that $s = 1$ (otherwise there are not enough nodes to combine) and so the number of nodes which increase their level by 2 is $2 + 0 = 2(2^{a_1}) = c_{r,j}$.

If $a_1 > 0$ then there are an even number of nodes present and so in the next r-step, every node will have its level increased by exactly one. Thus the number of nodes which have their levels increased by two will be twice the number of nodes at the next level that do. Thus, by induction,

$$2(2(2^{a_1-1} + 2^{a_1-1+a2} + 2^{a_1-1+a_2+a_3} + \ldots + 2^{a_1-1+\ldots+a_s})) = c_{r,j}$$

nodes have their levels increased by two.

It remains to show how the nodes are partitioned. We will use induction on the number r-steps to do this. The statement about partitioning is obvious for one r-step, regardless of whether r is even or odd. Assuming it is true for $j-1$ r-steps, we shall prove it for j. Let the nodes in order be $v_1, \ldots, v_r$. If $a_1 = 0$, then there are an odd number of nodes present. After one r-step the nodes in order are $v_{3,4}, v_{5,6}, \ldots, v_{r-2,r-1}, v_{(1,2),r}$. By induction these nodes get partitioned as pictured in Figure 4.10.

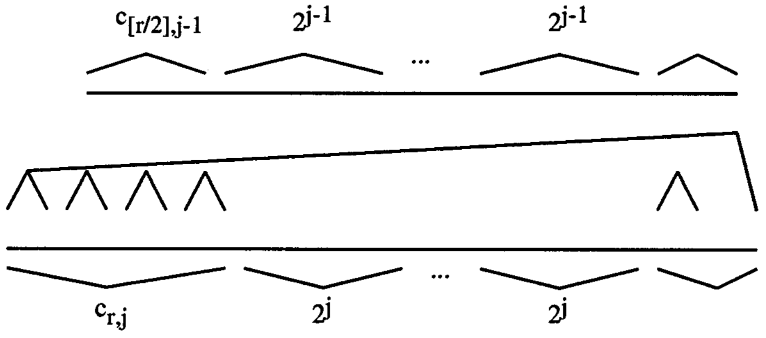


Figure 4.10: Partition according to inductive hypothesis

Observe that $c_{r,j} = 2 + 2c_{[r/2],j-1}$. From the diagram and the inductive hypothesis, we see that the first $c_{r,j}$ nodes and the $r - (c_{r,j+m2^j})$ largest nodes in R share a common ancestor in the region formed after j r-steps, . It is also clear that the $m2^j$ nodes in between are partitioned in sections of size 2^j. If $a_1 > 0$, then r is even and it is easy to see that $c_{r,j} = 2c_{[r/2],j-1}$ and the partition of the nodes in R is as stated. □

Having shown how to predict the nodes formed after j r- steps, we now need to find out when it is economical to do so.

Theorem 4.2 *Let the region currently being processed be R, and assume it contains r* *nodes. Let the gap height between R and its lowest adjacent neighbor be h. We can determine the region S formed after $h-1$ r-steps in $O(r\lg(\frac{r}{2^h}))$ time.*

Proof: There are approximately $(\frac{r}{2^h})$ nearly evenly spaced values to find in R. Using the linear time selection algorithm we can find the middle one in $O(r)$ time. Split the list on this value into two approximately equal halves and again find the middle value in each; this will take $O(\frac{r}{2})+O(\frac{r}{2})=O(r)$ time. After $O(\lg(\frac{r}{2^h}))$ steps, we will know the partitioning of R according ancestors in the new region S, at a cost of $O(r\lg(\frac{r}{2^h}))$ time, and so we can accurately form the nodes in S by just summing the weights in each nodes partition of R. □

Now use the conventional means of combining lmcps in S until all nodes have entered the nearest adjacent region and we are finished. This only costs an additional $O((\frac{r}{2^h})\lg(\frac{r}{2^h}))$ steps, including the time necessary to sort the list of entering nodes, so the entire running time is still $O(r\lg(\frac{r}{2^h}))$.

4.2 The Constant Factor Case

We now outline the linear time algorithm for weights within a constant factor. That is, for the case where there is a constant C such that $\max\{w_i\}/\min\{w_i\} < C$. We make use of new linear time generalized selection algorithm discovered by Maria Klawe. Because of the complexity of the result and its recent discovery we only provide an outline of the method and refer the reader to a forthcoming (as of August, 1992) paper of the author and Maria Klawe.

As before, our objective is to determine the levels of the leaf nodes in the lmcp tree. The underlying idea is an extension of the techniques in the case where the input weights are within a factor of two. Specifically, 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. At this point we apply Theorem 3.1 to determine the lmcp tree levels of the nodes in this final region, and then work backwards from this information to find the lmcp tree levels of the original weights.

There are a number of fundamental problems that arise in implementing this strategy. The first is that given a region of r nodes that may be either crossable or noncrossable, finding the lmcps that will be formed out of the region can take $\Omega(r \lg r)$ time since we may be forced to sort the crossable nodes. Thus we cannot afford to actually determine which nodes pair together in lmcps, nor the weights of the lmcps formed. Instead we only obtain coarser information about the structure of the lmcp tree. In particular, we will determine groups of nodes that pair together in the sense that the lmcp partner of each node in the group is also in the group. As this procedure is iterated, we must be able to "process" regions where we only have a coarse description of the region. Specifically the description of a region is an ordered list of node-groups, where each noncrossable node is a singleton node-group, and each crossable node-group, G, is described by an node-group list representing the children of the nodes in G. For each node-group we will know the number of nodes in the group, and the largest and two smallest weights of the nodes in the node-group.

Working with node-group lists introduces a number of technical challenges. For example, given a node-group list for a region, R, we must construct a node-group list for L, the set of lmcps containing nodes in R. If the number r of nodes in R is even, because the regions adjacent to R have higher category numbers, it is easy to show that the lmcp partner of each node of R is in R. This makes constructing the node-group list for L fairly trivial (i.e. it consists of a single node-group), except for the problem of determining the largest and smallest weights of the nodes in L. The reason we need to know the largest and smallest weights of the nodes in each node-group arises from handling the case when

r is odd. In this case, R has exactly one node, a wallflower, whose lmcp partner is not a node in R. We need to know the largest weight in each node group in order to find the wallflower, and we will need to know the two smallest weights in order to know, at the next stage, which node will be the lmcp partner of the wallflower. Our general strategy for efficiently computing the largest weight among the lmcps that will be produced from a region, is to construct (in linear time) a list of weighted nodes for which we can obtain all the lmcps in linear time, and to prove that the largest weight lmcp from this list is the same as that from the region. We use a similar method to find the two smallest weights. The choice of these approximate lists is fairly natural, but proving that they possess the appropriate extremal lmcps requires establishing a number of tools to deal with how modifying work lists affects the extremal lmcps.

Another problem that arises is in applying Theorem 3.1 to the final region of crossable nodes. In the case where the original weights differ by at most a factor of two, we know the actual weight of each crossable node in the region and hence can apply the usual linear-time selection algorithm to find the k smallest weight nodes for the appropriate value of k. In the constant factor case, however, we only know the largest and two smallest weights in each node-group, which makes selecting the k smallest weight nodes much more difficult. What we need is a linear-time technique for producing a node-group list that represents the k smallest weight nodes, and a node-group list representing the remaining nodes in the region. From these we can work backwards to deduce the lmcp tree levels of each of the original weights.

The linear-time algorithm for selection in node-groups consists of two pieces. The first piece is a generalized selection algorithm, which given a collection of sets and a "fast" selection algorithm for each set, does "fast" selection on the union of the sets. Here the term "fast" selection algorithm for a set A means that for any α between 0 and 1, in time linear in the number of nodes in A we can partition A into $A_1 \cup A_2$ according to the following conditions. First, we have "fast" selection algorithms for both A_1 and A_2; second, every element in A_1 is less than or equal to every element in A_2; and third, $A_1 = \lfloor \alpha A \rfloor$. The other piece of the node-group selection algorithm is a technique that does the following task: Given "fast" selection algorithms for each node-group in a node-group list for a region R and an integer k, produce a node-group list that represents the k smallest weight nodes in L, the set of lmcps constructed from R, as well as a node-group list representing the remaining nodes in L. By iteratively interleaving these two pieces we eventually obtain an algorithm which performs the desired selection in the final region in time linear in the number of weights in the original list (though as might be expected, the constant depends on C, the bound on the ratio between the original weights).

4.3 The Exponentially Separated Case

In this section we show that the basic region processing method yields an $O(n)$ time algorithm for the case where the input weights are exponentially spread-out. Formally we say the the input weight sequence $w_1, \ldots, w_n$ is **exponentially separated** if there exists a constant C such that for all n,

$$| \{i : [\lg w_i] = k\} | < C \text{ for all } k \in \mathrm{Z}.$$

Theorem 4.3 *If the input weights $w_1, \ldots w_n$ are exponentially separated then we can build an optimal alphabetic binary tree on them in $O(n)$ time.*

Proof: It is straightforward to observe that there are at most $2C$ nodes in any intermediate region we process. Since this is the case, we can easily process every region of size r in $O(r)$ time. Since this is the case, we can afford to use the basic region-processing method to build the entire lmcp tree in $O(n)$ time. In an additional $O(n)$ time we can build the optimal alphabetic binary tree from the leaf-levels in the lmcp tree. □

Chapter 5

Hardness Results

We begin with a simple hardness result that shows 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 constructing the optimal alphabetic tree by means of a region-based method. At this point we are uncertain what model of computation is realistic to use for this problem, so we do not state explicit lower bounds for a given model, but rather show that the problems stated are at least as hard as sorting in whatever model used. Given an unsorted list of n numbers, in $O(n)$ time we show we can transform the problem of sorting the list into a problem about constructing a tree, and from the information recorded in the structure of tree produced, compute the sorted order of the numbers in $O(n)$ time. Hence, if a $\omega(n)$ time lower bound exists for sorting in the model, then we have non-trivial lower bound on constructing the tree.

One easy way of getting a trivial lower bound is counting the number of possible outputs the algorithm must generate. In any binary decision tree model in which outputs appear at distinct leaves, the algorithm must be able to differentiate each of the different output cases, and so any decision tree must have maximum depth at least $\lg s$, where s is the number of different solutions to the problem. A trivial $\Omega(n)$ time lower bound follows easily by showing that the number of possible alphabetic trees built on a sequence of n weights is $2^{\Theta(n)}$. To see this, let c_n be the number of alphabetic binary trees with n leaves. As the root of the tree can split the input list in any of $n-1$ places it follows that

$c_n = c_1 c_{n-1} + c_2 c_{n-2} + \dots c_{n-1} c_1$ for $n > 1$, with $c_1 = 1$. The solution of this recurrence is given by the Catalan sequence, $\{C_n\}$, [9], with $c_n = C_{n-1} = \frac{1}{n}\binom{2(n-1)}{(n-1)} = 2^{\Theta(n)}$.

5.1 Finding the lmcp tree is as Hard as Sorting

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

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

Proof: First note that $1 \le y_i < 2$ for $i = 1, \dots, 2n$. Thus each new node formed from a pair will have weight at least 2. This means that any noncrossable node would prefer to combine with another noncrossable node, and thus the first n lmcp combinations are between noncrossable nodes. Since we are only concerned with the first n lmcps, we may assume that we just delete any two nodes involved in an lmcp. This ensures that all lmcps found are between consecutive nodes in the list. Moreover, it suffices to show that if (y_i, y_{i+1}) is among the first n lmcps formed, then i is odd. Since (y_i, y_{i+1}) is an lmcp we have $y_{i-1} > y_{i+1}$ and $y_i \le y_{i+2}$. If i were even it would follow that $y_{i-1} = y_i$ and $y_i = y_{i+1}$ which is a contradiction. Hence i is odd and the result follows. □

Theorem 5.2 *We can reduce sorting to finding the lmcp tree in $O(n)$ time.*

Proof: Assume n is even. Let $x_1, x_2, \dots, x_n$ be drawn from $[2, 4)$. Define the y_i as above and consider the behavior of some lmcp-combining algorithm on the input sequence $y_1, \dots, y_{2n}$. According to Lemma 5.1, after n lmcps have been combined there will be n nodes present in the node list. They will have weights $x_1, \dots, x_n$. Since these nodes are all crossable there will be only one lmcp present; the smallest pair of nodes in $\{x_1, \ldots, x_n\}$. This pair will combine to form a new node having 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 after the first n lmcp combinations have occurred sort $\{x_1, \ldots, x_n\}$ by pairs (only consecutive pairs may need to switched in order for the list to be totally sorted). This information can easily be recovered from an lmcp tree produced by any method by searching it depth-first, always searching the least weight subtree first. We will encounter the nodes corresponding to $\{x_1, \ldots, x_n\}$ in fully sorted order (a node with weight x_i will be the parent of leaf nodes with weights y_{2i-1} and y_{2i}). Hence we can reduce sorting to finding the lmcp tree in $O(n)$ time. □

5.2 Hardness Result for a Class of Region-based Methods

If it were possible to process any region of size r in $O(r)$ time, it is easy to show that the region-based method of chapter two would then yield a linear time algorithm for finding optimal alphabetic binary trees. Since we need only find the level of the nodes in the lmcp tree, and not its actual structure, it is natural to see if we might be able to avoid finding all the lmcps in every region we process. For example, we know that every node will pair up with another in the region we are working on, except possibly the last node when the number of nodes in the region is odd. This last node may form an lmcp with one of the newly created nodes, or with an accessible node in one of the adjacent regions. This suggests that we just need to know the smallest two nodes (they pair up to form the smallest newly created node) in each new region when we start processing, as we can then resolve in constant time which of the possible three alternative nodes the leftover node combines with. In light of this, perhaps there is an efficient way of maintaining this information for every region we create? Unfortunately, the answer is no. We show that any algorithm which follow this strategy will essentially sort the the input weights. The algorithm need not find all the lmcps, it need only find enough of them to compute the two smallest nodes in each region. On the surface this might be potentially easier to do. For example, in the case where all the weights are within a factor of two and n is a power of 2, it can be done in $O(n)$ time by combining the first $n/2$ lmcps (to ensure that every node is crossable). Let them be, in increasing order, $x_1, \ldots, x_{n/2}$. We can find $x_1, x_2, x_4, \ldots, x_{n/4}$ (the median) in $O(n)$ time by using the linear time median-finding algorithm [7] to find $x_{n/4}$ in $O(n)$ time, then using it again on the list of weights less than or equal to $x_{n/4}$ to find $x_{n/8}$ in $O(n/2)$ time and so on. In an additional $O(n)$ time we can scan through the list of nodes and determine whether they are in $[x_1, x_2], [x_3, x_4], [x_5, x_8], \ldots, [x_{n/4+1}, x_{n/2}]$. It is clear from Figure 5.11 that we can now easily predict the two smallest nodes in the k-th region formed after the first $n/2$ lmcps have combined. The smallest node will be the root of a balanced tree built on the weights $x_1, \ldots, x_{2^k}$ and the second smallest node will be the root of a balanced tree built on the weights $x_{2^k+1}, \ldots, x_{2^{k+1}}$.

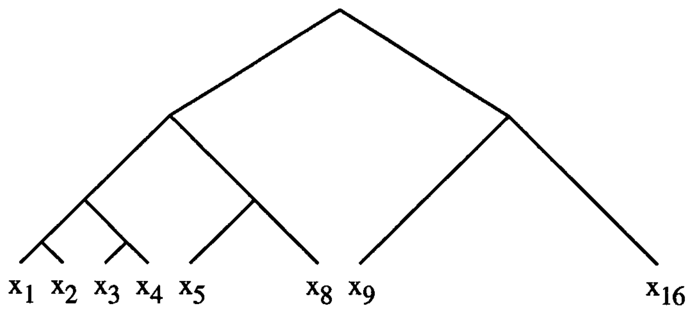


Figure 5.11: Simple case

Theorem 5.3 *We can reduce sorting to constructing a tree with the same leaf-levels as the lmcp tree by processing regions in such a way that the two smallest nodes in every region root the same leaves (and thus have the same weights) as they do in the lmcp tree in $O(n)$ time.*

Proof: We consider the behavior of any region-processing algorithm which accurately finds the smallest two nodes in each region on a class of inputs which have the property that with this information alone we know the structure of the lmcp tree. 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. The nodes in a region are distributed in such a way that in the lmcp tree they interleave with the newly formed nodes created from the region to the left. It turns out that insisting that the two smallest nodes in each region be accurate in the sense that they root exactly those leaves that the corresponding node in the lmcp tree does, forces enough structure on the tree being built, that it is essentially the lmcp tree up to some linear time reordering of nodes. From this tree it is possible in linear time to sort the entire input sequence. Finally, there are enough different input sequences in this class that it is as hard as sorting in the general case.

Assume $n = k^2 + 3k + 4$, where k is a positive integer. Begin with 2^{k+1} real variables $x_1 < x_2 < \ldots < x_{2^{k+1}}$ drawn from $[2, 4)$. (We assume that they have been sorted in ascending order.) The input list will consist of weights which form successively increasing regions. The first region will contain weights values in $[1, 2)$, the next $[2, 4)$, then $[4, 8)$, etc. Denote the j-th value in the i-th region by y_{ij}. 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 $4k + 4 + 2(k-1) + 2(k-2) + \ldots + 2 = k^2 + 3k + 4$. The values for the $\{y_{ij}\}$ 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.

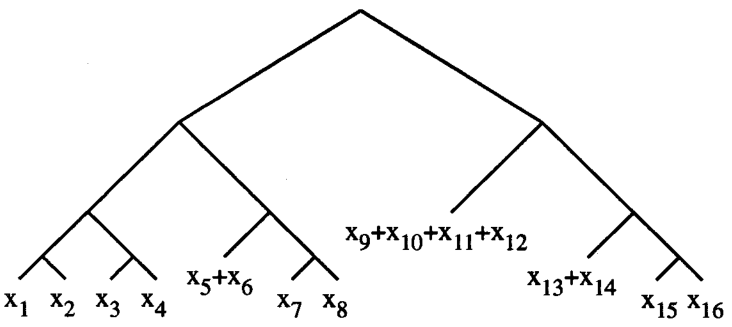


Figure 5.12: Tree generated from $\{x_i\}$

Figure 5.12 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 arm with leaf weights $x_{17} + \ldots + x_{24}, x_{25} + \ldots + x_{28}, x_{29} + x_{30}, x_{31}, x_{32}$ from left to right.

The purpose of this tree is to assign values to the $\{y_{ij}\}$. 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_{1j} be half the weight of the leaf that it is associated with. Then assign values to consecutive pairs of the $2(k - 1)\{y_{2j}\}$ by distributing them among the next lowest terminal leaves and so on. This new tree is called the **ordering tree**, and is shown in Figure 5.13. It records how the weights were assigned, and also their sorted order.

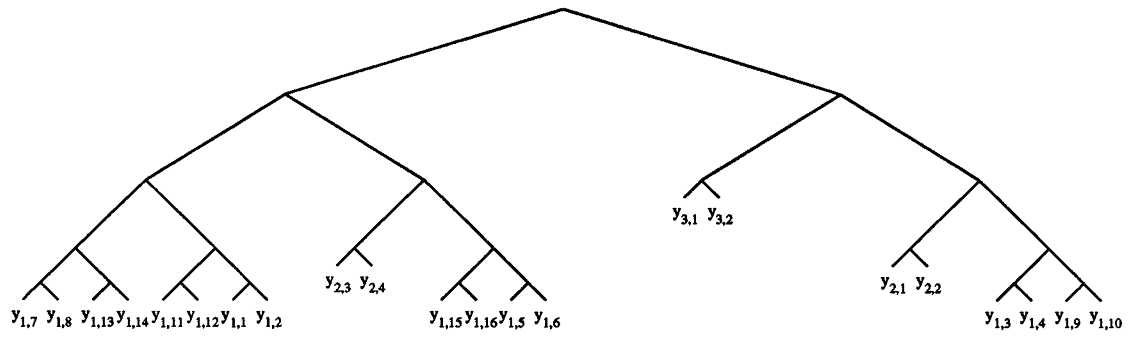

$y_{1,7}$ $y_{1,8}$ $y_{1,13}$ $y_{1,14}$ $y_{1,11}$ $y_{1,12}$ $y_{1,1}$ $y_{1,2}$
$y_{2,3}$ $y_{2,4}$
$y_{1,15}$ $y_{1,16}$ $y_{1,5}$ $y_{1,6}$
$y_{3,1}$ $y_{3,2}$
$y_{2,1}$ $y_{2,2}$
$y_{1,3}$ $y_{1,4}$ $y_{1,9}$ $y_{1,10}$

Figure 5.13: The ordering tree

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

$$y_{3,1}, y_{3,2}$$

$$y_{2,1}, y_{2,2}, y_{2,3}, y_{2,4}$$

$$y_{1,1}, y_{1,2}, \ldots, y_{1,15}, y_{1,16}$$

Now consider the behavior of any region-based algorithm which finds the smallest two lmcps in every region. The region chosen to process will always be the first one on the left, as the regions present are always sorted in increasing order. We also note that there are always be an even number of nodes in every intermediate region. To finish the proof of the main result, 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 work list 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. □

We can now easily predict how the weights will be distributed on the tree T that our algorithm produces. We know that the last region processed will contain just two nodes. Since these nodes will be the two smallest nodes in that region, their weights must match the weights of the same nodes in the lmcp tree. The smallest node will thus root all the leaves on the left branch of root the ordering tree, while the second smallest (the largest, in this case) 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 $\{y_{ij}\}$ 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 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 last region. This lets us find all the leaves in the right branch of left branch from the root in the ordering tree. Again, we may sort the weights present by region, 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 produce sorted lists of the weights in each region in the input. All this takes only $O(n)$ time to do, once the tree T is known.

We have not reduced the problem of sorting arbitrary sequences to using region-based methods to construct the optimal alphabetic binary tree because the sequences that we sort 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. However, this problem seems to be, asymptotically speaking, just as hard as sorting. An informal information-theoretic argument for this is that the same order of bits must be given to specify the sorted order. The total number of different orderings of the input sequences subject to the condition that the first $4k+4$ weights come before the next $2(k-1)$ which come before the next $2(k-2)$, etc. is

$$(2k+2)!(k-1)!(k-2)!\cdots(2)! > ([k/2]!)^{[k/2]} > [k/4]^{[k/4][k/2]} = \Omega(k^{\Theta(k^2)}).$$

Since $k = \Theta(n^{\frac{1}{2}})$, this number is $\Omega(n^{\Theta(n)})$, which the information necessary to specify the sorted order for arbitrary sequences. □

Chapter 6

Conclusions

In this thesis 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 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 discovered by Maria Klawe. 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. We also make some observations about speeding up region-based methods in the general case.

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, though it is conceivable that the ideas in the proof of Theorem 5.2 could be extended to obtain a superlinear lower bound for the general problem in a weaker decision tree model.

Bibliography

[1] T. C. Hu and A. C. Tucker. *Optimal Computer Search Trees and Variable-Length Alphabetical Codes.* SIAM Journal of Applied Mathematics, Vol. 21, No. 4, pp. 514-532, 1971.

[2] T. C. Hu, D. J. Kleitman and J. K. Tamaki. *Binary Trees Optimum Under Various Criteria.* SIAM Journal of Applied Mathematics, Vol. 37, No. 2, pp. 246-256, 1979.

[3] A. M. Garsia and M. L. Wachs. *A New Algorithm for Minimum Cost Binary Trees.* SIAM Journal of Computing, Vol. 6, No. 4, pp. 622-642, 1977.

[4] E. N. Gilbert and E. F. Moore. *Variable length encodings.* Bell System Technical Journal, Vol. 38, pp. 933-968, 1959.

[5] F. F. Yao. *Speed-up in dynamic programming.* SIAM Journal on Algebraic and Discrete Methods, Vol. 3, No. 4, pp. 532-540, 1982.

[6] P. Ramanan. *Testing the optimality of alphabetic trees.* Theoretical Computer Science. *to appear.*

[7] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest and R. E. Tarjan. *Time bounds for selection.* Journal of Computer and System Sciences, Vol. 7, No. 4, pp. 448-461, 1972.

[8] D. E. Knuth. *Optimum binary search trees.* Acta Informatica, Vol. 1, pp. 14-25, 1971.

[9] D. E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1968.

[10] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.