Optimal Integer Alphabetic Trees in Linear Time

T.C. Hu[1], Lawrence L. Larmore[2,⋆], and J. David Morgenthaler[3]

[1] Department of Computer Science and Engineering,
University of California, San Diego CA 92093, USA
hu@cs.ucsd.edu
[2] Department of Computer Science, University of Nevada, Las Vegas NV 89154, USA
larmore@cs.unlv.edu
[3] Applied Biosystems, Foster City CA 94404, USA
jdm123@gmail.com

Abstract. We show that optimal alphabetic binary trees can be constructed in $O(n)$ time if the elements of the initial sequence are drawn from a domain that can be sorted in linear time. We describe a hybrid algorithm that combines the bottom-up approach of the original Hu-Tucker algorithm with the top-down approach of Larmore and Przytycka's Cartesian tree algorithms. The hybrid algorithm demonstrates the computational equivalence of sorting and level tree construction.

1 Introduction

Binary trees and binary codes are fundamental concepts in computer science, and have been intensively studied for over 50 years. In his 1952 paper, Huffman described an algorithm for finding an optimal code that minimizes the average codeword length [1]. Huffman coding is a classic, well known example of binary tree or binary code optimization, and has led to an extensive literature [2]. The problem of computing an optimal Huffman code has $\Theta(n \log n)$ time complexity, but requires only $O(n)$ time if the input is already sorted.

The problem of finding an optimal search tree where all data are in the leaves, also called an *optimal alphabetic binary tree* (OABT), was originally proposed by Gilbert and Moore [3], who give an $O(n^3)$ time algorithm based on dynamic programming, later refined by Knuth to $O(n^2)$ [4]. The first of several related $O(n \log n)$ time algorithms, the Hu-Tucker algorithm (HT), was discovered in 1971 [5]. Similar algorithms with better performance in special cases, though all $O(n \log n)$ time in the general case, are given in [6–9]. Different proofs of the correctness of these algorithms appear in [10–13].

We give a new algorithm for the OABT problem that takes advantage of additional structure of the input to allow construction of an OABT in $O(n)$ time if weights can be sorted in linear time, *e.g.*, if the weights are all integers in a small range. Our algorithm combines the bottom-up approach of the original

⋆ Research supported by NSF grant CCR-0312093.

G.S. Brodal and S. Leonardi (Eds.): ESA 2005, LNCS 3669, pp. 226–237, 2005.

Hu-Tucker algorithm [5] with the top-down approach of Larmore and Przytycka's Cartesian tree algorithms [9].

Klawe and Mumey reduced sorting to Hu-Tucker based algorithms, resulting in a $\Omega(n \log n)$ lower bound for such *level tree* based solutions [8]. Larmore and Przytycka related the complexity of the OABT problem to the complexity of sorting, and gave an $O(n\sqrt{\log n})$ time algorithm for the integer case when sorting requires only $O(n)$ time [9]. Their Cartesian tree based algorithm provided new insight into the structure of the OABT problem, which we elaborate here. Our new algorithm requires sorting $O(n)$ items, which together with the reduction given by Klawe and Mumey [8], shows the computational equivalence of sorting and level tree construction.

2 Problem Definition and Earlier Results

Recall the definitions used in the Hu-Tucker algorithm [11]. We wish to construct an optimal alphabetic tree T from an initial ordered sequence of weights $S = \{s_1, \ldots, s_n\}$, given as n *square nodes*. The square nodes will be the leaves of T, and retain their linear ordering. An optimal tree has minimal cost subject to that condition, defining the cost of tree T as:

$$cost(T) = \sum_{i=1}^{n} s_i l_i$$

where l_i is the distance from s_i to the root.

The first phase of the Hu-Tucker algorithm combines the squares to form internal nodes of a *level tree*. The second and third phases use the level tree to create an OABT in $O(n)$ time. The internal nodes created during the first (combination) phase are called *circular nodes*, or *circles*, to differentiate them from the square nodes of the original sequence. The *weight* of a square node is defined to be its original weight in the initial sequence, while the *weight* of a circular node is defined to be the sum of the weights of its children. The level tree is unique if there are no ties, or if a consistent tie-breaking scheme (such as the one given in [12]) is adopted. Algorithms that use the level tree method produce the same level tree, although the circular nodes are computed in different orders. The (non-deterministic) *level tree algorithm* (LTA) given below generalizes those algorithms. As an example, the Hu-Tucker algorithm is an LTA where the circular nodes are constructed in order of increasing weight.

We remark that circular nodes have also been called *packages* [9], *crossable nodes* [8], and *transparent nodes* [12].

The *index* of each square node is its index in the initial sequence, and the *index* of any node of the level tree is the smallest index any leaf descendant. We will refer to square nodes as s_i, circular nodes as c_i and nodes in general as v_i, where i is the index of the node. By an abuse of notation, we also let v_i denote the weight of the node v_i. If two weights are equal, we use indices as tie-breakers. See [12] for the detailed description of this tie-breaking scheme.

Initially, we are given the sequence of items which are all square nodes. As the algorithm progresses, nodes are deleted and circular nodes are inserted into the node sequence.

Definition 1. *Two nodes in a node sequence are called a* compatible pair *if all nodes between them are circular nodes. We write* (v_a, v_b) *to represent the pair itself, and also, by an abuse of notation, the combined weight of the pair,* $v_a + v_b$.

Definition 2. *A compatible pair of nodes* (v_b, v_c) *is a* locally minimum compatible pair*, written* $\mathrm{lmcp}(v_b, v_c)$*, when the following is true:* $v_a > v_c$ *for all other nodes* v_a *compatible with* v_b *and* $v_b < v_d$ *for all other nodes* v_d *compatible with* v_c*. By an abuse of notation, we will use* $\mathrm{lmcp}(v_b, v_c)$ *to refer to the pair of nodes and also to their combined weight.*

2.1 The Hu-Tucker Algorithm and LTA

We now describe the Hu-Tucker Algorithm [5]. Define COMBINE(v_a, v_b) to be the operation that deletes v_a and v_b from S and returns the new circular node $c_a = (v_a, v_b)$. (Note that if v_a is a circular node, then the new node will have the same name as its left child, but that will not cause confusion in the algorithm since the child will be deleted from the node sequence.)

HU-TUCKER ALGORITHM returns the OABT for $S = \{s_1, \ldots, s_n\}$.

1. While S contains more than one node:
   1.1. Let (v_a, v_b) be the least weight compatible pair in S.
   1.2. Insert $c_a = \text{COMBINE}(v_a, v_b)$ into S at the same position as v_a.
2. Let c^* be the single circular node remaining in S.
3. For each $1 \le i \le n$, let d_i be the depth of s_i in the tree rooted at c^*.
4. Let T be the unique alphabetic tree whose leaves are $s_1, \ldots s_n$ such that, for each i, the depth of s_i in T is d_i.
5. Return T.

The tree rooted at c^* is the level tree of the original sequence. The level tree can be constructed by combining locally minimal compatible pairs in any order. Thus, we generalize HT to the *level tree algorithm* (LTA) as follows:

LEVEL TREE ALGORITHM(LTA) returns the OABT for $S = \{s_1, \ldots, s_n\}$.

1. While S contains more than one node:
   1.1. Let (v_a, v_b) be **any** lmcp in S.
   1.2. Insert $c_a = \text{COMBINE}(v_a, v_b)$ into S.
2. Let c^* be the single node remaining in S.
3. For each $1 \le i \le n$, let d_i be the depth of s_i in the tree rooted at c^*.
4. Let T be the unique alphabetic tree whose leaves are $s_1, \ldots s_n$ such that, for each i, the depth of s_i in T is d_i.
5. Return T.

In the insertion step of LTA, namely step 1.2 above, the new circular node c_a is placed in the position vacated by its left child. However, LTA gives the correct level tree if c_a is placed anywhere between $\mathit{leftWall}(c_a)$ and $\mathit{rightWall}(c_a)$, where

leftWall(c_a) is the nearest square node to the left of c_a whose weight is greater than c_a, and *rightWall*(c_a) is the nearest square node to the right of c_a whose weight is greater than c_a. We will place fictitious infinite squares s_0 and s_∞ at either end of the initial sequence, as described in Section 2.2, so *leftWall*(c_a) and *rightWall*(c_a) are always defined. The choices in LTA do not alter the level tree, but may change the order in which the circular nodes are computed [6–8].

LTA consists of $n-1$ iterations of the main loop. Its time complexity is dominated by the amortized time to execute one iteration of this loop. The Hu-Tucker algorithm takes $O(n \log n)$ to construct the level tree, because it requires $O(\log n)$ time to find the minimum compatible pair and update the structure.

Since the number of possible level trees on a list of length n is $2^{\Theta(n \log n)}$, the time complexity of LTA, in the decision tree model of computation, must be $\Omega(n \log n)$ in general. Our algorithm, also a deterministic version of LTA, makes use of $O(n)$-time sorting for integers in a restricted range, and takes $O(n)$ time to construct the OABT, provided all weights are integers in the range $0 \ldots n^{O(1)}$.

In our algorithm, we do not necessarily actually insert a circular node into the node sequence. Instead, we make use of data structures which are associated with certain parts of the sequence, which we call *mountains* and *valleys* (see Section 2.2). Each circular node is *virtually* in the correct position, and our data structures ensure that we can always find a pair of nodes which *would have been* a locally minimal compatible pair if the nodes had actually been inserted.

During the course of the algorithm, nodes can be moved from one data structure to another before being combined, and nodes are also sorted within data structures. We achieve overall linear time by making sure that the combined time of all such steps amortizes to $O(1)$ per node, and that the next locally minimal compatible pair can always be found in $O(1)$ time.

The contribution of this paper is that by restricting the Cartesian tree (see Section 2.3) to *mountains* we reduce the complexity of the integer algorithm given in [9] from $O(n\sqrt{\log n})$ to linear.

2.2 Mountains and Valleys

Any input sequence contains an alternating series of pairwise local minima (the lmcps) and local maxima, which we call *mountains*. It is the mountains and their structure that enable us to relocate each new circle in constant amortized time.

"Dual" to the definition of locally minimal compatible pair, we define:

Definition 3. *A compatible pair of square nodes* (s_d, s_e) *is a* locally maximum adjacent pair *if its weight is greater than all other compatible pairs containing either element.*

Definition 4. *A* mountain *is the larger of a locally maximum adjacent pair of nodes in the initial weight sequence. If node* s_i *is a mountain, we also label it* M_i*.*

We extend the initial sequence S by adding two "virtual" mountains $s_0 = M_0$ and $s_\infty = M_\infty$ of infinite weight to the ends. Write S^+ for the resulting *extended weight sequence*. The virtual mountains are never combined, but are only for notational convenience, giving us a uniform definition of "valley" in S^+.

Definition 5. *A* valley *is a subsequence of the initial weights between and including two adjacent mountains in the extended weight sequence S^+. We label the valley formed by mountains M_i and M_j as $V(M_i, M_j)$. The* top *of valley $V(M_i, M_j)$ is its minimum mountain* $\min\{M_i, M_j\}$*, while the* bottom *is its lmcp.*

Valleys (or the equivalent *easy tree* [9]) are a basic unit of the combination phase. All nodes within a single valley can be combined in linear time (see Section 3). During the combination phase of our algorithm, we repeatedly compute the minimum compatible pair in each valley; As nodes combine, this pair may not be an lmcp. To handle this situation, we first generalize locally minimum compatible pair to apply to any subsequence.

Definition 6. *Let $S_i^p = (v_i, v_j, \ldots, v_p)$ be a subsequence of nodes. Then the* minimum compatible pair *of S_i^p, written* $\mathrm{mcp}(i,p)$*, is the compatible pair of minimum weight:* $\mathrm{mcp}(i,p) = \min\left\{(v_a, v_d) \,|\, i \le a < d \le p\right\}$

Note that if $\mathrm{mcp}(i,p)$ does not contain either v_i or v_p, it must be an lmcp. That is, the local minimum in the subsequence is also a local minimum in the full sequence because we must see a full 'window' of four nodes in order to apply the definition of lmcp. If $\mathrm{mcp}(i,p)$ includes either end, we cannot do so. This fact motivates the use of valleys in our algorithm, and the need to distinguish mountains for special handling.

2.3 Cartesian Trees

The Cartesian tree data structure is originally described in [14]. Larmore and Przytycka base their OABT algorithms on Cartesian trees over the entire input sequence [9], but here we will limit the tree to contain only mountains. We recursively define the Cartesian tree of any sequence of weights:

Definition 7. *The* Cartesian tree *for an empty sequence is empty. The root of a Cartesian tree for a non-empty sequence is the maximum weight, and its children are the Cartesian trees for the subsequences to the right and left of the root.*

We construct the Cartesian tree of the sequence of mountains. We label mountain M_i's parent in the Cartesian tree as $p(M_i)$.

2.4 Algorithm Overview

At each phase of our algorithm, nodes in every valley are combined independently as much as possible. Any new circular node which is greater than the top of its valley is stored in a global set U for later distribution.

When no more combinations are possible, we remove all mountains which have fewer than two children in the Cartesian tree (that is always more than half the mountains) and move the contents of the global set of circles U to sets associated with the remaining mountains. As mountains are removed, their associated sets are combined and sorted to facilitate the next round of combinations.

3 Single Valley LTA

This section gives an LTA for a single valley containing only one lmcp, to improve the reader's intuition. We use a queue of circles in each valley to help find each new lmcp in turn. We label this queue $Q_{f,g}$ for valley $V(M_f, M_g)$. Rather than adding the circles back into the sequence, we put them into the queue instead. For purposes of operation COMBINE, we consider the queue to be a part of the sequence, located at the index of its initial circle.

SINGLE VALLEY LTA computes an optimal level tree in linear time since constant work is performed for each iteration. Only the main loop differs from the general LTA given above, so we omit steps 2–5.

SINGLE VALLEY LTA for S^+ containing single valley $V(M_0, M_\infty)$.

1. While the sequence S contains more than one node:
   1.1. Let $\text{lmcp}(v_a, v_b)$ be the result of VALLEY MCP for valley $V(M_0, M_\infty)$.
   1.2. Add $c_a = \text{COMBINE}(v_a, v_b)$ to the end of $Q_{0,\infty}$.

A little bookkeeping helps us determine each lmcp in constant time. We only need to consider the six nodes at the bottom of the valley in order to find the next lmcp. These six nodes are the valley's *active* nodes. Let the bottom two nodes on $Q_{i,j}$ be c_x and c_y, if they exist, where $c_x < c_y$. Any nodes that do not exist (*e.g.* the queue contains only one node) are ignored.

SUBROUTINE VALLEY MCP returns $\text{mcp}(i, j)$ for valley $V(M_i, M_j)$.

1. Let s_a be the square left adjacent to $Q_{i,j}$.
2. Let s_f be the square right adjacent to $Q_{i,j}$.
3. Return $\min\{(s_a, s_{a-1}), (s_a, c_x), (c_x, c_y), (c_x, s_f), (s_f, s_{f+1}), (s_a, s_f)\}$, ignoring any pairs with missing nodes. In addition, we require $a > i$ (otherwise ignore s_{a-1}), and $f < j$ (otherwise ignore s_{f+1}).[1]

SINGLE VALLEY LTA relies on the fact that each lmcp must be larger than the last within a valley. It also solves the Huffman coding problem in linear time if the input is first sorted [11].

4 Multiple Valleys

Consider the operation of SINGLE VALLEY LTA for an input sequence containing more than one valley. We can independently combine nodes in each valley only to a certain point. When the weight of a new circle is greater than one of the adjacent mountains, that circle no longer belongs in the current valley's queue, and must be *exported*, to use the terminology of [9]. But the valley into which the circle must be imported does not yet exist. Eventually, when the mountain between them combines, two (or more) valleys will merge into a new valley. If its adjacent mountains both weigh more than the new circle, this valley will import the circle. Mountains separate valleys in a hierarchical fashion, concisely represented by a Cartesian tree.

[1] The additional restrictions ensure that we stay within the valley.

4.1 Cartesian Tree Properties

For the k initial valleys separated by the nodes of this Cartesian tree, we have $k-1$ latent valleys that will emerge as mountains combine. Adding the initial valleys as leaves to the Cartesian tree of mountains yields a full binary tree of nested valleys with root $V(M_0, M_\infty)$. The internal nodes of this valley tree (the mountains) precisely correspond to the merged valleys created as our algorithm progresses. A mountain node *branches* in the Cartesian tree if its children are both mountains. Our algorithm takes advantage of the following property of Cartesian trees:

Property 1. Between each pair of adjacent leaf nodes in a Cartesian tree, there is exactly one branching node. Proof is by construction.

This property implies that for k leaves, there are $k-1$ branching nodes in a Cartesian tree. A tree may have any number of additional internal nodes with a single child, which we call *non-branching nodes.* Our algorithm handles each of these three types of mountains differently, so we will take a closer look at their local structure.

Consider three adjacent mountain M_h, M_i, and M_j, where $h < i < j$. We can determine the type of mountain M_i in the Cartesian tree by comparing its weight to the weights of its two neighbors. There are three cases:

- If $M_h > M_i < M_j$, then M_i is a leaf in the Cartesian tree.
- If $M_h < M_i > M_j$, then M_i is a branching node separating two *active regions.*
- If $M_h < M_i < M_j$, or $M_h > M_i > M_j$, then M_i is a non-branching node and the top of the valley separating M_i and its larger neighbor.

An *active region* is the set of all the valleys between two adjacent branching nodes. During execution of our algorithm, the active regions of each iteration will form single valleys in the next iteration. To bound the number of iterations our algorithm must execute, we need one additional Cartesian tree property. Let $|C|$ be the number of nodes in tree C. As every non-empty binary tree C contains between 1 and $\lceil |C|/2 \rceil$ leaves, with the help of Property 1 we have:

Property 2. Cartesian tree C contains fewer than $|C|/2$ branching nodes.

4.2 Filling a Valley

The structure of the mountains and valleys creates two transition points during the combination phase in each valley. The first is the export of the first circle. Before that point, all nodes stay within a valley and new circles are added to its queue. After the first transition point, all new circles are exported until the valley is eliminated. The removal of an adjacent mountain is the second transition point, which merges the current valley with one or more of its neighbors. After each combination, we maintain the following valley invariants needed by VALLEY MCP. Each valley $V(M_i, M_j)$ must contain:

- Two adjacent mountains.
- A possibly empty sorted queue of circles $Q_{i,j}$, with all circles $< \min\{M_i, M_j\}$.
- A cursor pointing into $Q_{i,j}$, initialized to point to the front of the queue.

- A possibly empty pairwise monotonic decreasing subsequence of squares to the right of M_i.
- A possibly empty pairwise monotonic increasing subsequence of squares to the left of M_j.

First, we note that while $\text{mcp}(i,j) < \min\{M_i, M_j\}$, a valley is isolated and we define the following subroutine INITIAL FILL to reach the first transition point. We call this subroutine the first time we handle a new valley, whether that valley appears in the initial sequence or after merging.

As new valleys are created, they will be prepopulated with a sorted queue containing imported circles. We will use the cursor to help us insert newly created circles into this queue in case it contains circles larger than the next lmcp.

SUBROUTINE INITIAL FILL for valley $V(M_i, M_j)$.

1. Find the initial $\text{lmcp}(v_a, v_b)$ in $V(M_i, M_j)$.
2. While $\text{lmcp}(v_a, v_b) < \min\{M_i, M_j\}$:
   2.1. Insert $c_a = \text{COMBINE}(v_a, v_b)$ into $Q_{i,j}$, advancing the cursor as needed.
   2.2. Let $\text{lmcp}(v_a, v_b)$ be the result of VALLEY MCP for valley $V(M_i, M_j)$.

Notice that INITIAL FILL is nearly the same as SINGLE VALLEY LTA. We have added a stronger condition on the while loop, and we need to use the cursor to merge new circles into the queue. We now show how to fill a valley:

SUBROUTINE FILL VALLEY returns queue of circles Q for valley $V(M_i, M_j)$.

1. Call INITIAL FILL for $V(M_i, M_j)$.
2. Create empty queue Q.
3. While $\min\{M_i, M_j\} \notin \{\text{VALLEY MCP for } V(M_i, M_j)\}$:
   3.1. Add $c_a = \text{COMBINE}(\text{VALLEY MCP for } V(M_i, M_j))$ to end of Q.
4. Return Q.

After FILL VALLEY returns, the top of the valley, $\min\{M_i, M_j\}$, is an element of $\text{mcp}(i,j)$, and we say that valley $V(M_i, M_j)$ is *full*. Combining or otherwise removing the minimum mountain requires merging valleys and reestablishing the necessary valley invariants, and also handling the exported queue of circles. For each mountain M_i, we will store the circles imported by its corresponding latent valley in an unsorted set U_i, where $c_s \in U_i \implies M_i < c_s < p(M_i)$. When a new valley is created, this set will become its imported queue Q.

4.3 Merging Valleys

For valley $V(M_i, M_j)$, $\text{mcp}(i,j)$ is an lmcp if it does not contain M_i or M_j, since nodes inside a valley, *i.e.*, between the mountains, cannot be compatible with any outside the valley (lemma 5 in [11]). However, the mountains themselves each participate in two valleys, thus these nodes are compatible with other nodes outside the single valley subsequence $\{M_i, \ldots, M_j\}$.

First, consider a mountain M_i smaller than its neighbors M_h and M_j. M_i must be a leaf in the Cartesian tree of mountains, and after adjacent valleys $V(M_h, M_i)$ and $V(M_i, M_j)$ are full, $M_i \in \text{mcp}(h,i)$ and $M_i \in \text{mcp}(i,j)$. We are

ready to merge these two valleys into $V(M_h, M_j)$ by combining the mountain. Note that $\mathrm{mcp}(h,j) = \min\{\mathrm{mcp}(h,i), \mathrm{mcp}(i,j)\}$, which will form an lmcp if it does not contain M_h or M_j. To explain the combination of valleys, we start with this leaf mountain case where $M_h \notin \mathrm{mcp}(h,j)$ and $M_j \notin \mathrm{mcp}(h,j)$.

Two Valley Case. When both valleys adjacent to leaf mountain M_i are full, we merge them into a single new valley and establish the valley invariants needed by Valley MCP. Subroutine Merge Two Valleys provides the bookkeeping for the merge. The result is new valley $V(M_h, M_j)$, ready to be filled. Merge Two Valleys requires that both valleys are full, and so each contain at most one node (square or circle) smaller than and compatible with M_i. In addition, the lmcp may not contain two mountains; this omitted three valley case is in fact even simpler.

Subroutine Merge Two Valleys returns set of circles W for mountain M_i.

1. Create empty set W.
2. Let $V(M_h, M_i)$ and $V(M_i, M_j)$ be the valleys to the left and right of M_i.
3. Merge queues $Q_{h,i}$ and $Q_{i,j}$, each of which contains at most one circle, to create $Q_{h,j}$. Initialize a new cursor to point to the front of $Q_{h,j}$.
4. Sort U_i, the imported circles for the new valley, and add to the end of $Q_{h,j}$.
5. Find the smallest node $v_{\min}$ compatible with M_i from among the only three possible: the front of $Q_{h,j}$, or the squares left or right adjacent to M_i.
6. Let circle $c_m = \textsc{Combine}(v_{\min}, M_i)$, assuming $v_{\min}$ is to the left of M_i.
7. Create new valley $V(M_h, M_j)$ with $Q_{h,j}$, removing M_i, U_i, $V(M_h, M_i)$ and $V(M_i, M_j)$ from further consideration.
8. If circle $c_m < \min\{M_h, M_j\}$ then:
   8.1. Insert c_m into $Q_{h,j}$, advancing the cursor as needed.
   else:
   8.2. Add c_m to W.
9. Return W.

4.4 Removing Non-branching Mountains

In this section we explain the removal of non-branching mountains. These mountains are adjacent to one larger and one smaller neighbor, and lie between a branching mountain and a leaf mountain, or at either end of the sequence.

For example, consider valley $V(M_i, M_j)$, where $M_i < M_j$. Once $V(M_i, M_j)$ is full, we remove mountain M_i by either combining it or turning it into a normal square. When M_i no longer separates its adjacent valleys, we merge them into one. This process continues repeatedly until a single valley separates each adjacent branching and leaf mountain.

Since a full valley contains at most one node smaller than the minimum mountain M_i, we need to consider three cases. Let $M_h < M_i < M_j$, and let s_a and s_d be the squares left and right adjacent to M_i, respectively.

1. The filled valley has an empty queue and no square smaller than M_j. Then M_i is no longer a mountain, as $s_a < M_i < s_d$. We make M_i into a normal

square s_i by removing the mountain label. Square s_i becomes part of the pairwise monotonic increasing sequence of squares to the left of M_j.
2. The filled valley contains a single square s_d smaller than M_i. So $s_d < M_i < v_b$ for all v_b compatible with s_d. If $s_a > s_d$, then M_i and s_d must form $\text{lmcp}(M_i, s_d)$. Combining $\text{lmcp}(M_i, s_d)$ lets us connect the pairwise monotonic increasing subsequence of squares to the left of M_i with the subsequence to the left of M_j, forming a single pairwise increasing subsequence. Otherwise, we can convert M_i into a normal square node as in the case above.
3. The filled valley contains a single circle c_d. This is case is similar to the previous case, but may require that we convert c_d into a *pseudo-square*. We can treat this circle as though it were square because it is smaller than either neighbor and must combine before its transparency has any effect.

Subroutine Remove Mountain removes non-branching, internal nodes from the Cartesian tree. Without loss of generality, assume that the left adjacent mountain $M_h < M_i$, while the right adjacent mountain $M_j > M_i$.

Subroutine Remove Mountain returns set of circles W for mountain M_i.
1. Create empty set W.
2. Let $(M_i, v_a) = \text{mcp}(i, j)$.
3. If (M_i, v_a) is an lmcp then:[2]
   3.1. Add $c_i = \text{Combine}(M_i, v_a)$ to W.
   else:
   3.2. Rename M_i to s_i and remove M_i from other data structures.
   3.3. If $Q_{i,j}$ is not empty, convert remaining circle into a pseudo-square.
4. Add U_i to U_h.
5. Create new valley $V(M_h, M_j)$, removing U_i, $V(M_h, M_i)$ and $V(M_i, M_j)$.
6. Return W.

When Remove Mountain completes, mountain M_i has been removed from the sequence and many valley invariants established for the newly created valley. The missing invariants involve $Q_{h,j}$, which we will postpone creating until we have merged all the valleys between adjacent branching mountains. All that is needed is to sort U_h, which occurs in subroutine Merge Two Valleys.

5 K Valley LTA

Let $\widehat{C}$ be the Cartesian tree formed from the mountains in the original sequence. Each node of $\widehat{C}$ separates two valleys in that sequence. As mentioned earlier, we call a non-leaf node in $\widehat{C}$ with a single child a *non-branching node*. Our final algorithm shrinks $\widehat{C}$ by removing at least half of the nodes at each iteration. When all the nodes of $\widehat{C}$ have been combined, we compute the level tree using Initial Fill. The input sequence S^+ is the extended sequence created by adding two nodes of infinite weight to either end of the original sequence S.

K Valley LTA computes level tree for S^+.
1. Compute the Cartesian tree $\widehat{C}$ from the $k-1$ mountains in the original S.
2. Create a global empty set U, and an empty set U_i for each mountain M_i.

[2] (M_i, v_a) is an lmcp if and only if $s_b > v_a$ for the square node s_b to the left of M_i.

3. While $\widehat{C}$ contains more nodes:
   3.1. For each valley $V(M_i, M_j)$ in the current sequence:
      3.1.1. Add $Q =$ FILL VALLEY for $V(M_i, M_j)$ to U.
   3.2. For each non-branching mountain M_i in $\widehat{C}$:
      3.2.1. Add $W =$ REMOVE MOUNTAIN for M_i to U.
      3.2.2. Delete all the mountains combined in the previous step from $\widehat{C}$.
   3.3. Use Static Tree Set Union (see below) to place U's circles into the U_i.
   3.4. For each leaf mountain M_j in $\widehat{C}$:
      3.4.1. Add $W =$ MERGE TWO VALLEYS for M_j to U.
4. Use the sorted set from the final step 3.3 to create $Q_{0,\infty}$ and call INITIAL FILL for $V(M_0, M_\infty)$. The final circle remaining between M_0 and M_∞ is the root of the level tree.

Static Tree Set Union (STSU) is a specialized version of the Union-Find problem that applies to a tree, and computes a sequence of *link* and *find* operations in linear time [15]. We can use this algorithm to find the set U_i in which each circle in global set U belongs in linear time if we first sort U together with the mountains remaining in $\widehat{C}$.

We apply the approach described in [9], first initializing the family of named sets with a singleton set for each node of $\widehat{C}$. We attach each circle of U to $\widehat{C}$ as a child of its exporter, the argument of the call to REMOVE MOUNTAIN where it was created. Next, we sort all the nodes of this extended tree by weight to create sorted list L'. Processing nodes of L' in increasing weight order, we execute a sequence of *link* and *find* operations in $O(n)$ time as follows:

- If the current node is M_j, perform *link*(M_j), which adds the contents of the set containing M_j to the set containing $p(M_j)$, then deletes the old set containing M_j.
- If the current node is circle c_m, perform *find*(c_m), obtaining the set containing mountain $p(c_m)$. That set is named with the minimum ancestor mountain M_j in $\widehat{C}$ that dominates c_m. Determine whether c_m is to the right or left of M_j. Add c_m to the U_i of the mountain child on that side of M_j.

6 Complexity

The algorithm is dominated by sorting, which occurs in steps 3.3 and 3.4.1 We now show that the algorithm sorts only $O(n)$ nodes.

Lemma 1. *The STSU in step 3.3 sorts $O(n)$ nodes altogether.*

Proof. Initially $|\widehat{C}| = k < \frac{n}{2}$. By Property 2, we remove at least half the mountains at each iteration of step 3. Thus, over all iterations, at most $2k < n$ mountains are sorted. No circle is sorted more than once by step 3.3, since set U contains only newly created circles. The total of all circles sorted by step 3.3 is therefore less than n. Over all steps 3.3, we sort a total of less than $2n$ items.

Lemma 2. *Step 3.4.1 sorts $O(n)$ nodes altogether.*

Proof. Each U_i is sorted at most once. Each circle appears in only one U_i being sorted, as it is moved to some $Q_{a,b}$ after sorting.

Theorem 1. K VALLEY LTA *sorts* $O(n)$ *weights.*

Proof. No other sorting is performed; apply lemmas 1 and 2.

Theorem 2. *For weights taken from an input domain that can be sorted in linear time, an optimal alphabetic binary tree can be constructed in* $O(n)$ *time.*

Proof. All other operations are linear in the size of the input. Proof follows from Theorem 1 and the linear creation of the OABT from the level tree.

7 Conclusion

We have given an algorithm to construct optimal alphabetic binary trees in time bounded by sorting. This algorithm shows that sorting and level tree construction are equivalent problems, and leads to an $O(n)$ time solution to the integer alphabetic binary tree problem for integers in the range $0 \ldots n^{O(1)}$.

References

1. Huffman, D.A.: A method for the construction of minimum redundancy codes. Proceedings of the IRE **40** (1952) 1098–1101
2. Abrahams, J.: Code and parse trees for lossless source encoding. In: Proceedings Compression and Complexity of Sequences. (1997) 146–171
3. Gilbert, E.N., Moore, E.F.: Variable length binary encodings. Bell System Technical Journal **38** (1959) 933–968
4. Knuth, D.E.: Optimum binary search tree. Acta Informatica **1** (1971) 14–25
5. Hu, T.C., Tucker, A.C.: Optimal computer search trees and variable-length alphabetic codes. SIAM Journal on Applied Mathematics **21** (1971) 514–532
6. Garsia, A.M., Wachs, M.L.: A new algorithm for minimal binary search trees. SIAM Journal on Computing **6** (1977) 622–642
7. Hu, T.C., Morgenthaler, J.D.: Optimum alphabetic binary trees. In: Combinatorics and Computer Science: 8th Franco-Japanese and 4th Franco-Chinese Conference. Lecture Notes in Computer Science, volume 1120, Springer-Verlag (1996) 234–243
8. Klawe, M.M., Mumey, B.: Upper and lower bounds on constructing alphabetic binary trees. SIAM Journal on Discrete Mathematics **8** (1995) 638–651
9. Larmore, L.L., Przytycka, T.M.: The optimal alphabetic tree problem revisited. Journal of Algorithms **28** (1998) 1–20
10. Hu, T.C.: A new proof of the T-C algorithm. SIAM Journal on Applied Mathematics **25** (1973) 83–94
11. Hu, T.C., Shing, M.T.: Combinatorial Algorithms, Second Edition. Dover (2002)
12. Karpinski, M., Larmore, L.L., Rytter, W.: Correctness of constructing optimal alphabetic trees revisited. Theoretical Computer Science **180** (1997) 309–324
13. Ramanan, P.: Testing the optimality of alphabetic trees. Theoretical Computer Science **93** (1992) 279–301
14. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proceedings of the 16th ACM Symposium on Theory of Computation. (1984) 135–143
15. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. Journal of Computer and System Sciences **30** (1985) 209–221