Partial alphabetic trees*

Arye Barkan † Haim Kaplan †

Corresponding author: Haim Kaplan, School of computer science, Tel Aviv University, Tel Aviv, Israel. phone: 972-3-6408814. Fax: 972-3-6409357. email: haimk@post.tau.ac.il

*Preliminary version of this paper appeared in the proceedings of ESA'02 [5]

†School of Computer Science, Faculty of exact sciences, Tel-Aviv University, Tel Aviv 69978, Israel. E-mail: {barkanar,haimk}@post.tau.ac.il.

Abstract

In the partial alphabetic tree problem we are given a multiset of nonnegative weights $W = \{w_1, \ldots, w_n\}$, partitioned into $m \le n$ blocks $B_1, \ldots, B_m$. We want to find a binary tree T where the elements of W resides in its leaves such that if we traverse the leaves from left to right then all leaves of B_i precede all leaves of B_j for every $i < j$. Furthermore among all such trees, T has to minimize $\sum_{i=1}^{n} w_i d(w_i)$, where $d(w_i)$ is the depth of w_i in T. The partial alphabetic tree problem generalizes the problem of finding a Huffman tree over W (there is only one block) and the problem of finding a minimum cost alphabetic tree over W (each block consists of a single item). This problem arises when we need an optimal binary code for a set of items with known frequencies, such that we have a lexicographic restriction for some of the codewords.

Our main result is a pseudo-polynomial time algorithm that finds the optimal tree. Our algorithm runs in $O(\left(\frac{W_{sum}}{W_{min}}\right)^{2\alpha} \log(\frac{W_{sum}}{W_{min}}) n^2)$ time where $W_{sum} = \sum_{i=1}^{n} w_i$, $W_{min} = \min_i w_i$, and $\alpha = \frac{1}{\log \phi} \approx 1.44$ [1]. In particular the running time is polynomial in case the weights are bounded by a polynomial of n. To bound the running time of our algorithm we prove an upper bound of $\lfloor \alpha \log(W_{sum}/W_{min}) + 0.56 \rfloor$ on the depth of the optimal tree.

Our algorithm relies on a solution to what we call the layered Huffman forest problem which is of independent interest. In the layered Huffman forest problem we are given an unordered multiset of weights $W = \{w_1, \ldots, w_n\}$, and a multiset of integers $D = \{d_1, \ldots, d_k\}$. We look for a forest F with k trees, $T_1, \ldots, T_k$, where the weights in W correspond to the leaves of F, that minimizes $\sum_{i=1}^{n} w_i d_F(w_i)$ where $d_F(w_i)$ is the depth of w_i in its tree plus d_j if $w_i \in T_j$. Our algorithm for this problem runs in $O(kn^2)$ time.

Keywords: Huffman tree, Huffman codes, alphabetic tree, alphabetic code, Hu Tucker algorithm, partial alphabetic tree

[1] $\phi = \frac{\sqrt{5}+1}{2} \approx 1.618$ is the golden ratio.

1 Introduction

Let T be a binary tree with n leaves each associated with a weight $w_i \geq 0$[2]. We define the *cost* of T to be $cost(T) = \sum_{i=1}^{n} w_i d_T(w_i)$ where $d_T(w_i)$ is the length of the path (number of edges) from the root to the leaf w_i. We call $d_T(w_i)$ the *depth* of w_i.

Given a sequence of n weights $w_1, \ldots, w_n$ the well known Huffman coding problem [9] looks for a binary tree of minimum cost whose leaves are associated with $w_1, \ldots, w_n$. A related and also well studied problem is the minimum cost alphabetic tree problem [7, 8]. In this problem we also look for a binary tree of minimum cost whose leaves are associated with $w_1, \ldots, w_n$ but this time we also require that w_i appears before w_j in an inorder traversal of the tree for every $i < j$.

The Huffman tree is used to construct optimal binary codes for a set of symbols with known frequencies of transmission. Optimal alphabetic trees are used to construct optimal alphabetic binary codes for a set of symbols with known frequencies of transmission. If we take the optimal alphabetic tree and distribute keys in internal nodes to allow search, then we get a search tree that minimizes the average access time to a set of items residing at the leaves (when the distribution of accesses is known in advance).

These two classical problems were defined about 50 years ago [9, 7], and have been heavily studied [2] since then. In 1952, Huffman [9] defined the well-known Huffman codes and suggested an algorithm for the Huffman problem that runs in $O(n \log n)$ time. About 7 years later, Gilbert and Moore [7] presented the minimum cost alphabetic tree problem, and suggested a dynamic programming algorithm to solve it in $O(n^3)$ time and $O(n^2)$ space. Later Knuth [13] improved the dynamic programming scheme of Gilbert and Moore and reduced its running time to $O(n^2)$. Hu and Tucker [8] gave a different algorithm, somewhat greedy in nature, that can be implemented to run in $O(n \log n)$ time. Their algorithm, though simple to describe, has a complicated correctness proof. Garcia and Wachs [6] describe another algorithm that solves the minimal cost alphabetic tree problem in $O(n \log n)$ time. The algorithm of Garcia and Wachs and the algorithm of Hu and Tucker are related [11].

In this paper we study what we call the *Partial Alphabetic Tree* (PAT) problem which is a natural generalization of both the Huffman problem and the minimal cost alphabetic tree problem. In the PAT problem we are given a set of weights $w_1, \ldots, w_n$ partitioned into m blocks $B_1, \ldots, B_m$. We want to find a minimum cost binary tree whose leaves are associated with $w_1, \ldots, w_n$ such that w_i precedes w_j in an inorder traversal of the tree if $w_i \in B_k$ and $w_j \in B_{k'}$ for $k < k'$. In less technical terms, we want to find a minimum cost binary tree such that all items in B_i are "to the left" of all items in B_j for every $i < j$. The order of the items within each block is arbitrary.

The Huffman problem is a special case of the PAT problem where there is only one block. The minimal cost alphabetic tree problem is also a special case of the PAT problem when there are $m = n$ blocks, and block i contains the single element w_i. Partial alphabetic trees are useful when we want to code a set of items with know frequencies subject to an alphabetic restriction

[2]One can think of the leaves as containing items $\{v_1, \ldots, v_n\}$ where $w(v_i) = w_i$. To simplify the notation we identify the item v_i with its weight w_i.

on some of the codewords.

For the optimal alphabetic tree problem the dynamic programming algorithm of Gilbert and Moore [7] is fairly straightforward. This algorithm solves $O(n^2)$ subproblems where each such problem is of the form "find the optimal alphabetic tree for $w_i, \ldots, w_j$". In contrast, for the PAT problem there is no such straightforward approach even if one is willing to settle for a worse but polynomial running time. Since the items are not totally ordered the number of subproblems in the straightforward approach of Gilbert and Moore potentially becomes exponential. We overcome this difficulty by using a dynamic programming scheme in which subproblems encapsulate structural conditions on the boundaries of the target trees.

Our main result is an algorithm for the the partial alphabetic tree problem with time complexity of $O(\left(\frac{W_{sum}}{W_{min}}\right)^{2\alpha} \log\left(\frac{W_{sum}}{W_{min}}\right) n^2)$ where $W_{sum} = \sum_{i=1}^{n} w_i$, and $W_{min} = \min_{i\in\{1,\ldots,n\}} w_i$, and $\alpha = \frac{1}{\log\phi} \approx 1.44$. In particular, our algorithm runs in polynomial time if the weights are bounded by a polynomial function of n. Our technique is general and can be used to find optimal partial alphabetic trees for several objective functions other than the average path length. The problem of whether there exists an algorithm for the partial alphabetic tree problem that runs in polynomial time for any set of weights remains an intriguing open question.

Our algorithm for the PAT problem relies on a solution to what we call the *layered Huffman forest* problem which is of independent interest. In the *layered Huffman forest* problem we are given a sequence of weights $w_1, \ldots, w_n$ and a sequence of *depths* $d_1, \ldots, d_k$. The goal is to find a forest F of k binary trees that minimizes $\sum_{i=1}^{n} w_i d_F(w_i)$ where $d_F(w_i) = d_{T_j}(w_i) + d_j$ if $w_i \in T_j$. We describe an algorithm for this problem that runs in $O(kn^2)$ time.

To bound the running time of our algorithm we have to bound the number of subproblems that we solve. Recall that our subproblems are defined based upon structural conditions on the boundaries of the target tree. Therefore to bound the number of such problems we prove an upper bound of $\lfloor \alpha\log(W_{sum}/W_{min}) + 2 - \alpha \rfloor \approx \lfloor 1.44\log(W_{sum}/W_{min}) + 0.56 \rfloor$ on the depth of the minimal-cost alphabetic tree for $w_1, \ldots, w_n$. We also show that this bound is tight.

Our immediate interest in the PAT problem stems from recent algorithms to label trees for ancestor and least common ancestor queries [1, 3, 10, 4, 14]. The need for such algorithms arises in XML search engines and internet routing. Recent algorithms for these problems first balance the tree by partitioning it into heavy-paths. Then they label the edges outgoing of the vertices of each heavy path such that labels of edges outgoing from deeper vertices get lexicographic larger labels. There is no restriction, however, on the relative order of labels of edges outgoing from the same vertex. Therefore, we get exactly an instance of the PAT problem for each such heavy path where block B_i contains the edges outgoing from the ith vertex on the path. The weights in these instances of the PAT problem are all bounded by n, where n is the number of nodes in the XML file.

An application of the layered Huffman forest problem, is finding an optimum variable length prefix encoding for some alphabet, where the codewords for some of the symbols in the alphabet are fixed in advance. Consider for example a scenario where you have different alphabets for different languages (German, Japanese, Russian, etc...). Each such alphabet contains symbols specific to the particular language and symbols common to all alphabets such as English letters, digits, and various punctuation symbols. To allow easy conversions of streams between alphabets and common applications that can work on the common parts of all alphabets one may want to encode the character sets such that common symbols have the same encodings across all alphabets. To encode a new alphabet (say Hebrew), we want to find codewords to the special symbols of the new set which are consistent with the (already assigned) codewords of the common symbols. When the frequencies of the special symbols are known in advance we can find an optimal encoding to the new characters by solving the layered Huffman forest problem (See Section 3).

The structure of the paper is as follows. Section 2 presents some special cases of the PAT problem for which we have a polynomial time algorithm. Section 3 describes our main result, which is an algorithm that solves the PAT problem in polynomial time when the weights are bounded by a polynomial of n. Section 4 describes an algorithm to solve the layered Huffman forest problem. This algorithm is used by our algorithm for the PAT problem in Section 3. Section 5 gives an upper bound, in terms of W_{sum}/W_{min}, on the depth of the minimal-cost alphabetic tree. This upper-bound is used to bound the number of subproblems our algorithm for the PAT problem needs to solve. Section 6 defines some closely related problems and suggests directions for further research.

2 Some simple partial orders

As a warm-up we present in this section four problems which can be solved in polynomial time with relatively simple algorithms. Three of those problems are special cases of the PAT problem. The last one is a special case of some other generalization of the Huffman and alphabetic tree problems. In the next section we give a general solution to the PAT problem.

Problem 1: This problem is a special case of the PAT problem, in which only the leftmost and rightmost blocks have more than one element. Note that this special case includes the case when there are only two blocks.

Let T be a partial alphabetic tree over a set of weights W partitioned into blocks $B_1, \ldots, B_m$. We say that the tree T has the valley property if the leaves of the first block appear in a non-increasing weight order when listed from left to right and the leaves of the last block appear in a non-increasing weight order when listed from right to left. Our algorithm for this problem relies on the following lemma.

Lemma 2.1 *For every set of weights W partitioned into blocks $B_1, \ldots, B_m$ there exists an optimal partial alphabetic tree over those weights, which has the valley property.*

Proof. Given an optimal partial alphabetic tree which does not have the valley property, we show how to transform this tree into a partial alphabetic tree which has the valley property. Our transformation does not change the depths of the leaves (so it does not change the cost of the tree), and thus the transformed tree is also an optimal partial alphabetic tree.

Suppose we have an optimal partial alphabetic tree T in which the k heaviest leaves of the first block are the k leftmost leaves, and that these leaves appear in a non-increasing order of weights from left to right. Let w be the $(k+1)$th leaf from the left and let w' be the next heaviest leaf. If $w \neq w'$ our transformation would switch w and w' without changing the order of the already positioned k leftmost leaves, and without changing the order of leaves in other blocks.

From the definitions of w and w' follows that $w' > w$, and that w' is to the right of w. Let d be the depth of w'. Since T is optimal the depth of w is no smaller than d, and the depth of all the k leaves to the left of w is no larger than d. Let v be the ancestor of w at depth d. Note that

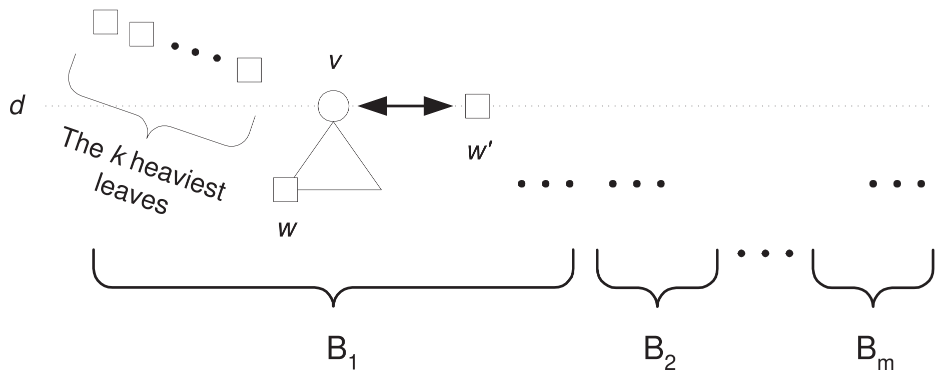

Figure 1: Transforming an optimal partial alphabetic tree into a tree which has a valley property (see Lemma 2.1)

v may be w itself. The subtree rooted at v does not contain any of the k leftmost nodes since their depth is at most d. From the same reason this subtree does not contain w'. It follows that all the leaves in the subtree rooted by v are leaves of the first block, since all of those leaves are to the left of w' which itself is in the first block.

Our transformation switches the nodes v and w' (see Figure 1). Since the leaves which hang on those two nodes are all from the first block, the transformed tree is still a partial alphabetic tree. Since w' and v have the same depth this transformation preserves the depths of all the leaves of the tree and thus the transformed tree is optimal.

By performing $n_1 = |B_1|$ such transformations on any optimal partial alphabetic tree we get another optimal partial alphabetic tree in which the leaves of the first block appear in a non-increasing order. By applying another $n_m = |B_m|$ similar transformations we can order the rightmost block as well, getting an optimal partial alphabetic tree which has the valley property. ■

Since there is always an optimal partial alphabetic tree with the valley property we can limit ourselves to finding such a tree instead of an arbitrary partial alphabetic tree. To do that we can order the weights of the first block from heavy to light and the weights of the last block from light to heavy. Then we can split the first and last blocks into many blocks, each containing a single weight. In the case that only the first and last blocks contain more than one item this will generate an instance of the minimal cost alphabetic tree problem which can be solved in $O(n \log n)$ time by the Hu-Tucker algorithm [8]. Since we can order the first and the last block in $O(n \log n)$ time, this case of the PAT problem can be solved in $O(n \log n)$ time where n is the total number of weights in all blocks.

Problem 2: This problem is a special case of the PAT problem, in which each block contains a constant number of elements. The algorithm which we suggest is based on dynamic programming and solves two types of subproblems.
(1) For every $1 \leq i \leq m$, and for every subset $X \subseteq B_i$, solve a Huffman problem over the weights X.
(2) For every i, j where $1 \leq i < j \leq m$, and for every two subsets $X \subseteq B_i$, $Y \subseteq B_j$, solve the PAT problem over the blocks $X, B_{i+1}, B_{i+2}, \ldots, B_{j-1}, Y$.
We solve subproblems of the first type using Huffman's algorithm. The following relation shows

how to obtain a solution to a subproblem of the second type using solutions to smaller subproblems.

$$cost(\mathrm{PAT}(B'_1,\ldots,B'_\ell)) = \min_{\substack{1\le k\le \ell\\ Z\subseteq B'_k}} \left[cost(\mathrm{PAT}(B'_1,\ldots,B'_{k-1},Z)) + cost(\mathrm{PAT}((B'_k\setminus Z),B'_{k+1},\ldots,B'_\ell)) + \sum_{w\in\cup_{i=1}^{\ell}B'_i} w \right]$$

If there are at most c items in each block, then there are $O(2^c m)$ subproblems of type (1), and $O(2^{2c}m^2)$ subproblems of type (2). It takes $O(c\log c)$ time to solve each subproblem of type (1), and $O(2^c m)$ time to solve each subproblem of type (2) when the smaller subproblems are already solved. Thus it takes a total of $O(2^{3c}m^3)$ time to solve the whole PAT problem. In case c is constant, this algorithm runs in $O(m^3) = O(n^3)$ time, and requires $O(n^2)$ space (as there are $O(n^2)$ subproblems).

Problem 3: This is a special case of the PAT problem where we do not put any additional restriction on the size or the number of blocks, but we do require that the weights would be in the range $[\lambda, 2\lambda)$ for some $0 < \lambda \in \mathbb{R}$. The following lemma states that in this case the minimal cost tree is balanced in the following sense:

Lemma 2.2 *Let W be a set of weights in which every weight $w \in W$ is in the range $[\lambda, 2\lambda)$ then in a minimal cost partial alphabetic tree for any block partition of W any leaf is either at depth d or depth $d-1$ for some d.*

Proof. Let T be a minimal cost partial alphabetic tree for some ordered block partition of the weights W. Let d be the depth of T. Suppose by contradiction that T contains a leaf w of depth $i \le d-2$. Let x and w be two leaves such that x is of depth d and w is of depth $\le d-2$ and the leaves between them, if there are any, are all of depth $d-1$. Assume w.l.o.g. that w is to the left of x, and denote the leaves between x and w by $y_1,\ldots,y_k$. We perform the following transformation (shown on Figure 2(a)): We replace w by a new node whose left child is w and whose right child is y_1. Then we replace each y_i by y_{i+1} for $1 \le i \le k-1$, and finally we replace y_k by x and the parent of x by the sibling of x. If there are no nodes between w and x we just replace w by a new node whose left child is w and whose right child is x, and replace the parent of x by the sibling of x. In both cases this transformation preserves the order of the leaves. Moreover the only node whose depth is increased is w, and its depth is increased by exactly one, while the depth of x and its sibling is decreased by at least one. Since all weights are within a factor of 2 from each other the total weight of the nodes that decrease its depth by at least one is larger than the weight of the node w which increases its depth by one and thus we get an alphabetic tree with lower cost contradicting the optimality of T. ■

For our dynamic programming algorithm we also need the following lemma.

Lemma 2.3 *Let $B_1,\ldots,B_m$, be blocks of weights where every weight is in the range $[\lambda, 2\lambda)$. Let the weights of block B_i be $w_{i,1},\ldots,w_{i,n_i}$, and assume that $w_{i,1} \le w_{i,2} \le \cdots \le w_{i,n_i}$ for every i. Then there exists a solution to the PAT problem over those weights, in which the order of weights of each block, B_i, when listed from left to right is either $w_{i,1},w_{i,2},\ldots,w_{i,n_i}$ or $w_{i,2},w_{i,3},\ldots,w_{i,n_i},w_{i,1}$.*

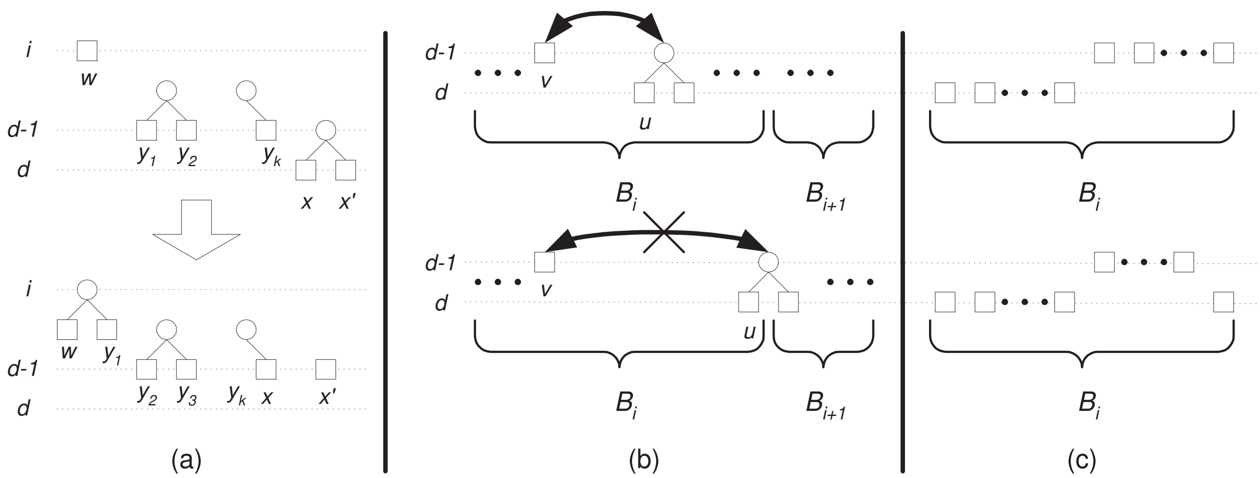

Figure 2: (a) The transformation described in the proof of Lemma 2.2. (b) The transformation described in the proof of Lemma 2.3 and the case when this transformation is not allowed. (c) The possible shapes of the block B_i after the transformation described in the proof of Lemma 2.3

Proof. Let T be an optimal partial alphabetic tree. We describe a transformation on that tree which preserves the cost of the tree, and changes the order of leaves within each block B_i to be either $w_{i,1}, w_{i,2}, \ldots, w_{i,n_i}$ or $w_{i,2}, w_{i,3}, \ldots, w_{i,n_i}, w_{i,1}$.

By Lemma 2.2 the depth of any leaf of block B_i is either d or $d-1$. First we show that we can change the tree while preserving its cost such that all the leaves of depth d, except maybe one, are to the left of all the leaves of depth $d-1$ in any block B_i. If all the leaves of block B_i are of the same depth then we are done. Otherwise let v and u be two leaves in block B_i, such that v is to the left of u, and the depth of v is $d-1$ while the depth of u is d.

We can change the order of these leaves by replacing v with the parent of u (see Figure 2(b) upper part). This transformation preserves the order of the blocks except in the case where u is the rightmost leaf of block B_i, and the sibling of u is in block B_{i+1} (see Figure 2(b) lower part). Thus by repeatedly applying this transformation we can always get a tree in which the leaves of depth d at block B_i except maybe one are to the left of the leaves of depth $d-1$, and if there is one leaf of depth d which is not to the left of all the leaves of depth $d-1$, then this leaf is the rightmost leaf in the block (see Figure 2(c)) .

Since the tree T has minimal cost, the weights of all the leaves of depth d in B_i is no larger than the weights of the leaves of depth $d-1$ in B_i (since otherwise we could reduce the cost of the tree by switching those leaves). Now, by changing places of leaves of the same depth in block B_i we can get a tree such that when listing the leaves of block B_i from left to right we get either $w_{i,1}, w_{i,2}, \ldots, w_{i,n_i}$ or $w_{i,2}, w_{i,3}, \ldots, w_{i,n_i}, w_{i,1}$, depending on whether the last leaf is of depth d or $d-1$. ■

Lemma 2.3 implies a dynamic programming algorithm similar to the one for Problem (2). In Problem (2) we used the fact that the number of items in each block is constant to bound the number of subproblems as well as the time to solve each subproblem of type 2. Here Lemma 2.3 bounds the number of possible orders of the items of each block. This also gives a bound on the number of subproblems and the time to solve each of them.

Problem 4 (Parallel alphabetic trees): Here we consider the case where the set of weights W is partitioned into a number of sets $W = W_1 \cup W_2 \cup \cdots \cup W_k$, where each set W_i is ordered. We denote the weights of W_i by $w_{i,1}, w_{i,2}, \ldots, w_{i,n_i}$. We say that a tree T over the weights W is a parallel-alphabetic tree if for every two weights $w_{i,p}$ and $w_{i,q}$ in the same set W_i the leaf associated with $w_{i,p}$ appears to the left of $w_{i,q}$ if and only if $p < q$. If the two weights are in different sets W_i and W_j they can appear in the tree in any order. We denote by PPO the problem of finding a minimal cost parallel alphabetic tree. In this section we solve a special case of PPO in which the number of sets k is constant.

Note that like the PAT problem, the PPO problem is also a generalization of both the Huffman and the minimal cost alphabetic tree problems. When there is only one set W_1, we get an instance of the minimal-cost alphabetic tree problem, and when there are n sets $W_1, \ldots, W_n$, and each set contains only one weight we get an instance of the Huffman problem.

We now show a straightforward dynamic programming solution to the PPO problem that runs in $O(n^{3k})$ time and requires $O(n^{2k})$ space. For k pairs of indices (ℓ_1, r_1), (ℓ_2, r_2), $\ldots$, (ℓ_k, r_k), where $0 \le \ell_i \le r_i \le n_i$, we denote by $\mathrm{PPO}(\ell_1, r_1, \ell_2, r_2, \ldots, \ell_k, r_k)$ the PPO problem restricted to $W' = W'_1 \cup \cdots \cup W'_k$ where $W'_i = \{w_{i,\ell_i}, \ldots, w_{i,r_i}\}$. It is easy to prove the following recursive relation from which we can immediately derive a dynamic programming algorithm. This relation shows how to compute a solution to $\mathrm{PPO}(\ell_1, r_1, \ell_2, r_2, \ldots, \ell_k, r_k)$ from solutions to smaller problems.

$$
\begin{aligned}
&\mathit{cost}\,(\mathrm{PPO}(\ell_1, r_1, \ldots, \ell_k, r_k)) = \\
&\qquad \min_{\substack{\ell_1 \le x_1 \le r_1 \\ \cdots \\ \ell_k \le x_k \le r_k}} [\mathit{cost}\,(\mathrm{PPO}(\ell_1, x_1, \ldots, \ell_k, x_k)) + \mathit{cost}\,(\mathrm{PPO}(x_1+1, r_1, \ldots, x_k+1, r_k)) + \sum_{v \in W'} w(v)]
\end{aligned}
$$

Note that the solution to the original PPO problem is a solution to $\mathrm{PPO}(1, n_1, 1, n_2, \ldots, 1, n_k)$. To solve a subproblem we calculate the minimum among $O(n^k)$ pairs of smaller subproblems. Since there are $O(n^{2k})$ subproblems it is easy to implement the algorithm to run in $O(n^{3k})$ time and $O(n^{2k})$ space[3].

3 Finding an Optimal Partial Alphabetic Tree

In this section we describe an algorithm for the PAT problem. This is a dynamic programming algorithm which uses solutions to two other problems discussed in later sections.

We use the term *binary tree* for a tree which is ordered and rooted, where each node has at most two children, a *left child*, and a *right child*.[4] A *complete binary tree* is a binary tree where each internal node[5] has exactly two children.

The *right-boundary* of a binary tree is the sequence of nodes on the path from the root to the rightmost leaf. Similarly the *left-boundary* of a binary tree is the sequence of nodes on the

[3] For each subproblem we store one number for the cost of the result, plus one node which is the root of the resulting tree. The two children of that root node are results of other subproblems, and thus their space is counted elsewhere. This requires a constant space for each subproblem.

[4] Even when there is only one child it is classified as either a left child or a right child.

[5] We use the term *internal node* to refer to nodes which are not leaves.

path from the root to the leftmost leaf. We write $rboundary(T)$ and $lboundary(T)$ to refer to the right-boundary and left-boundary of a tree T, respectively. We denote by v_i^R the node on the right-boundary at depth i, and by v_i^L the node on the left-boundary at depth i.

At a high level our approach is as follows. For every prefix of the sequence of the blocks $B_1, \ldots, B_k$ and every possible right-boundary c of an appropriate length, we will find the optimal binary tree over the items in $B_1, \ldots, B_k$ with c as its right boundary. Clearly when $k = m$ and the right boundary is a path consisting only of right edges we obtain the solution to the PAT problem. To specify the details we will need the following additional definitions.

Definition 3.1 *A* cut-tree T *is a binary tree* T *such that for every internal node* v *the followings hold.*
1) If $v \notin rboundary(T)$ *and* $v \notin lboundary(T)$ *then* v *has two children.*
2) If $v \in rboundary(T)$ *and* $v \notin lboundary(T)$ *then* v *has a left child (it might also have a right child).*
3) If $v \notin rboundary(T)$ *and* $v \in lboundary(T)$ *then* v *has a right child (it might also have a left child).*

A cut-tree T is called *right-cut-tree* if all the nodes on its left-boundary have a left child, and it is called *left-cut-tree* if all the nodes on its right-boundary have a right child. See Figure 3 for examples of cut-trees. We label each edge from a parent to a left child by "0", and from a parent to a right child by "1". We label a node v in the tree with a string of "0"s and "1"s that correspond to the path from the root to v. The *left-label* of a cut-tree T is the label of its leftmost leaf, and similarly the *right-label* of T is the label of its rightmost leaf. We denote those labels by $llabel(T)$ and $rlabel(T)$, respectively. For any tree T, $llabel(T) \leq rlabel(T)$ when compared lexicographically.

If s does not consist of all "1"s we define the *right suffix* of s, as the unique suffix of s of the form 01^*. Similarly if s does not consist of all "0"s we define the *left suffix* of s, as the unique suffix of s of the form 10^*. We denote the right suffix of s, and the left suffix of s by $rsuffix(s)$ and $lsuffix(s)$ respectively. We also define $rprefix(s) = s - rsuffix(s)$, and $lprefix(s) = s - lsuffix(s)$.

Definition 3.2 *We define the* left-code *of a cut-tree* T *to be* $lcode(T) = lprefix(llabel(T))$ *and the* right-code *of* T *to be* $rcode(T) = rprefix(rlabel(T))$.

Note that $lcode(T)$ is not defined when $llabel(T) =$ "000....0", and $rcode(T)$ is not defined when $rlabel(T) =$ "111....1", since $lsuffix$ and $rsuffix$ are not defined for strings of such forms respectively. Fortunately, we will never be interested in the $lcode$ of a tree which has a "000...0" $lboundary$ (right-cut-tree), nor we will be interested in the $rcode$ of a tree which has "111...1" $rboundary$ (left-cut-tree). See Figure 3 for examples of left-code and right-code.

Intuitively the left-code encodes the part of the left boundary containing all nodes missing left children and similarly the right-code encodes the part of the right boundary containing all nodes missing right children. We will be able to stick together two cut-trees into a larger one if the right-code of one is the same as the left-code of the other.

Let lc and rc be two binary strings. We say that lc *precedes* rc, and we write $lc <_p rc$ if one of the following three conditions holds: 1) $lc = prefix(rc)$ and the first digit of rc which is not in lc is "1". 2) $rc = prefix(lc)$ and the first digit of lc which is not in rc is "0". 3) lc and rc

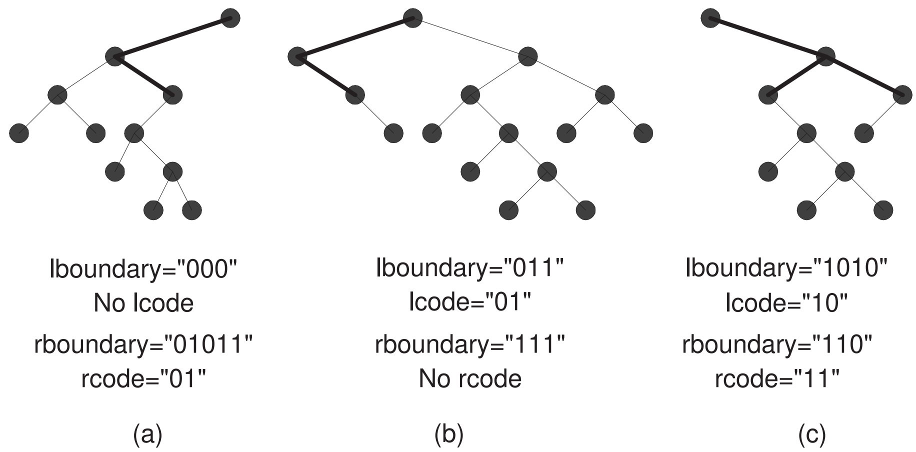

Figure 3: (a) Only right-code (Right-cut-tree). (b) Only left-code (Left-cut-tree). (c) Both right and left codes (Cut-tree). Bold edges represent the code.

are not prefix of one another, and $lc < rc$ lexicographically. The following lemma easily follows from our definitions.

Lemma 3.1 *(1) For every cut-tree T, which is not a right-cut-tree, and not a left-cut-tree, $lcode(T) <_p rcode(T)$ (2) For every two binary strings $lc <_p rc$, there exists a cut-tree T over some weight set W, such that $lcode(T) = lc$ and $rcode(T) = rc$.*

Proof. **(1)** Let p be the longest prefix which is common to $llabel(T)$ and $rlabel(T)$. Let a, b be such that $llabel(T) = p.a$ and $rlabel(T) = p.b$, where "." denotes concatenation of strings. There are two cases:

case 1: $a = \epsilon$, where ϵ is the empty string. In this case $b = \epsilon$ as well since otherwise the leftmost leaf would be an ancestor of the rightmost one. This is the case when $llabel(T) = rlabel(T)$, which happens when T is a chain that has only one leaf. We assume that p ends with a "0". The case when it ends with a "1" is similar. Since T is not right-cut-tree, $p \neq$ "00000...000", and thus $p = llabel(T) = rlabel(T) =$ "....1000...0" (there is a "1" followed by one or more "0"s). In this situation $rcode(T) = p -$ "0", i.e. p with the last digit removed, and $lcode(T) = p -$ "1000...0". This means that $rcode(T) = lcode(T).$"1000..." i.e. $rcode$ is $lcode$ followed by "1" and zero or more "0"s. By definition of $<_p$ this means that $lcode(T) <_p rcode(T)$.

case 2: $a \neq \epsilon$. In this case $b \neq \epsilon$ as well since otherwise the rightmost leaf would be an ancestor of the leftmost one. The first digit of a is "0" since it must be different from the first digit of b, and if it would be one, the right boundary would cross the left-boundary. Now there are four subcases:

Subcase a: String a contains the digit "1" and string b contains the digit "0". This subcase is illustrated on Figure 4(a). since a contains the digit "1", it properly contains the $lsuffix(llabel(T))$, and the first character of a is part of $lcode(T)$. Similarly, since b contains the digit "0", the first digit of b which is "1" is part of $rcode(T)$. Thus we get that $lcode(T) < rcode(T)$ lexicographically, and they are not prefix of one another. By the definition of $<_p$ that means that $lcode(T) <_p rcode(T)$.

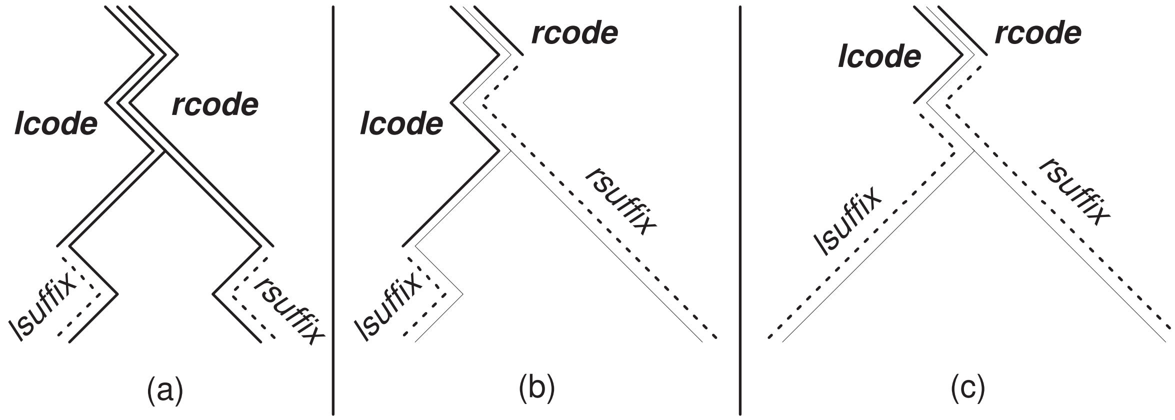

Figure 4: Illustration of the different cases in the proof of Lemma 3.1

Subcase b: String a contains the digit "1" but string b does not contain the digit "0". This subcase is illustrated on Figure 4(b). In this case $rsuffix(rlabel(T))$ contains all of b, it has at least one digit in p, and the first such digit in p is "0". Thus $rcode(T)$ is a prefix of $lcode(T)$, and the first digit in $lcode(T)$ that follows $rcode(T)$ is "0". By definition of $<_p$ this means that $lcode(T) <_p rcode(T)$.
Subcase c: String a does not contains the digit "1" but string b contains the digit "0". This case is symmetric to Subcase b.
Subcase d: String a does not contains the digit "1" and string b does not contain the digit "0". This subcase is illustrated on Figure 4(a). In this case $rprefix(rlabel(T)) = rprefix(p)$ and $lprefix(llabel(T)) = lprefix(p)$, and thus this case is similar to case 1 above.

(2) It is obvious that for any two labels ℓ_1 and ℓ_2 such that $\ell_1 <_{lex} \ell_2$ and none of which is prefix of the other, one can build a cut-tree T over some weight set W, such that $llabel(T) = \ell_1$ and $rlabel(T) = \ell_2$. We will show here that for any two strings $lc <_p rc$ we can build labels $\ell_1 <_{lex} \ell_2$ (none of them is prefix of the other) such that $lc = lprefix(\ell_1)$ and $rc = rprefix(\ell_2)$. This will prove (2).

By definition of $<_p$ there are three possible cases.
case 1 $lc = prefix(rc)$ and the first digit of rc which is not in lc is "1". Let k be the number of "0"s after this "1". Note that k can be 0. In this case we can define $\ell_1 = lc.$"100...0" where the number of "0"s is $k+1$ and $\ell_2 = rc.$"0".
case 2 $rc = prefix(lc)$ and the first digit of lc which is not in rc is "0". Let k be the number of "1"s after this "0". In this case we can define $\ell_1 = lc.$"1" and $\ell_2 = rc.$"011...1" where the number of "1"s is $k+1$.
case 3 $lc <_{lex} rc$ and none of them is the prefix of another. In this case we can simply define $\ell_1 = \ell_c.$"1" and $\ell_2 = rc.$"0".
It is easy to verify that in all those cases ℓ_1 and ℓ_2 are as desired. ■

Definition 3.3 *Let $lc <_p rc$ be two binary strings, and let $W = \{w_1, \ldots, w_n\}$ be a multiset of non-negative weights, then:*
$C(lc, rc, W)$ is the set of all cut-trees T with leaf set W, for which $lcode(T) = lc$ and $rcode(T) = rc$.

$RC(rc, W)$ is the set of all right-cut-trees T with leaf set W in which $rcode(T) = rc$.
$LC(lc, W)$ is the set of all left-cut-trees T with leaf set W in which $lcode(T) = lc$.

Note that depending on the values of rc, lc, and W, some or all of those three sets might be empty.

The *Huffman-cut-tree problem* is defined as follows. Given a multiset of weights $W = \{w_1, \ldots, w_n\}$, and two binary strings $rc <_p lc$, find a cut-tree T with minimum-cost in $C(lc, rc, W)$. We define *Huffman-right-cut-tree* and *Huffman-left-cut-tree* problems similarly.

The basic subproblems of our dynamic programming scheme would be Huffman-cut-tree problems where the set of weight W consists of all the weights belonging to a single block. Next we show how to solve these subproblems via a reduction to the layered Huffman forest problem.

Reducing the Huffman-cut-tree problem to the layered Huffman forest problem. We first define what is a layered forest and then define the *layered-Huffman-forest problem*. A *(binary)-layered-forest* is a forest of non-empty complete-binary trees, where each tree T_i has an integer $d_i \geq 0$ associated with it. We call d_i the *depth of T_i in the forest*. We define the depth of each node $v \in T_i$ in F to be its depth in T_i plus d_i, i.e. $d_F(v) = d_{T_i}(v) + d_i$. We define the cost of the forest F as $cost(F) = \sum_{i=1}^{n} d_F(w_i) w_i$

Let $W = \{w_1, \ldots, w_n\}$ where $0 \leq w_i \in \mathbb{R}$ be a multiset of weights, and $D = \{d_1, \ldots, d_k\}$ where $0 \leq d_i \in \mathbb{N}$ be a multiset of depths. We say that a layered forest F is *over W*, and *hanged on D*, if W is the set of leaves of F, and D is the set of the depths of its trees. We denote by $LF(W, D)$ the set of all layered-forests over W hanged on D. The *layered-Huffman-forest problem* is defined as follows. Given a multiset of weights $W = \{w_1, \ldots, w_n\}$ and a multiset of depths $D = \{d_1, \ldots, d_k\}$, find a forest $F \in LF(W, D)$ of minimal cost. In Section 4 we show how to solve the layered-Huffman-forest problem in $O(n^2 k)$ time. For now, lets assume we can solve it in $O(f(n, k))$ time, and show a reduction from the Huffman-cut-tree problem to the layered-Huffman-forest problem. This reduction implies a solution to the Huffman-cut-tree problem in $O(f(n, k))$ time where $k = length(lc) + length(rc)$.

Let T be a cut-tree, and let lc and rc be $lcode(T)$ and $rcode(T)$, respectively. Let j be the largest index for which $v_j^R = v_j^L$. Note that j is determined by rc and lc and is independent of the structure of the right and left boundaries of T which is not encoded in rc and lc. [6] Let $\ell = j$ if lc is a prefix of rc (Figure 5(b)) and $\ell = j + 1$ otherwise (Figure 5(a)). For any $\ell < i \leq length(rc)$ such that $rc[i] =$ "1", v_{i-1}^R has two children the left of which roots a complete binary tree. For such i we define R_i to be the left subtree of v_{i-1}^R. Note that the depth of R_i is i. We define L_i's similarly to be the right subtrees of nodes on the left-boundary that have two children. Note that if rc is a prefix of lc then the R_i's do not exist, and if lc is a prefix of rc then the L_i's do not exist (see Figure 5(b)). Furthermore, if rc is not prefix of lc, then we also define $R_{length(rc)+1}$ to be the left subtree of $v_{length(rc)}^R$. Notice that in this case the subtree $R_{length(rc)+1}$ must exist and it is a complete binary tree (see R_{10} on Figure 5(a)). If lc is not a prefix of rc we similarly define $L_{length(lc)+1}$. Note that the existence and depth of each R_i and L_i is determined solely by

[6](a) If rc and lc are not prefix of one another then j is the largest index for which $\forall_{1 \leq i \leq j} lc[i] = rc[i]$ (see Figure 4(a))

(b) If rc is a prefix of lc then j is one less than the minimum index i, $length(rc) + 1 < i \leq length(lc)$ for which $lc[i] =$ "0" (see Figure 4(b)). If no such i exists then $j = length(lc) + 1$ (see Figure 4(c)).

(c) If lc is a prefix of rc then j is one less than the minimum index i, $length(lc) + 1 < i \leq length(rc)$ for which $rc[i] =$ "1". If no such i exists then $j = length(rc) + 1$.

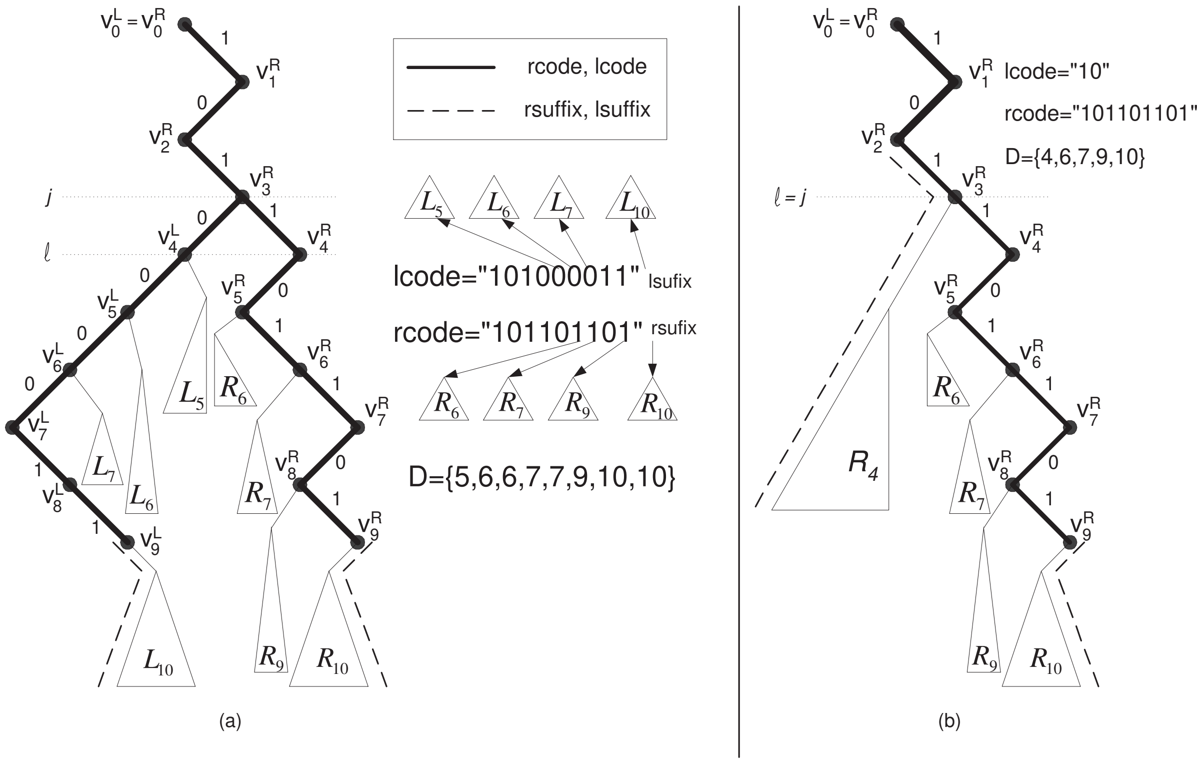

Figure 5: Illustration of the reduction from Huffman-cut-tree problem to the layered-Huffman-forest problem. (a) The case when rc and lc are not prefix of one another. (b) The case when lc is a prefix of rc.

rc and lc. Note also that if T is a right-cut-tree or a left-cut-tree we can similarly decompose it into R_i's and L_i's (see Figure 5(a,b)).

Let $lc <_p rc$ be two binary strings, and let D be the multiset of depths of the R_i's and L_i's determined by those strings. Then there is a bijection from $LF(W, D)$ to $C(lc, rc, W)$, i.e. we can decompose every cut-tree $T \in C(lc, rc, W)$ into a layered-forest $F \in LF(W, D)$ with the same cost, and for every layered-forest $F' \in LF(W, D)$ we can assemble a cut-tree $T' \in C(lc, rc, W)$ of the same cost. This implies that a solution to the layered-Huffman-forest problem gives us a solution to the Huffman-cut-tree problem.

Let $h = max(length(rc), length(lc))$. It takes $O(h)$ time to build D from lc and rc, and the number of roots in D is $O(h)$. In Section 4 we show how to solve the layered-Huffman-forest problem in $O(n^2k)$ time where n is the number of leaves and k is the number of roots. Using the reduction just described this implies that we can solve the Huffman-cut-tree problem in $O(n^2h)$ time. In Section 5 we give a bound on h in terms of the W_{sum}/W_{min} where $W_{sum} = \sum_{w \in W} w$ and $W_{min} = \min_{w \in W} w$.

It is also easy to do similar reductions from the Huffman-left-cut-tree and Huffman-right-cut-tree problems to the layered-Huffman-forest problem. In the first case there are only the R_i trees, and in the second case there are only the L_i trees. See Figure 5(b,c) for an example.

Back to our dynamic programming scheme. The following two lemma show how we combine solutions to the Huffman cut tree problems for single blocks to solutions to larger

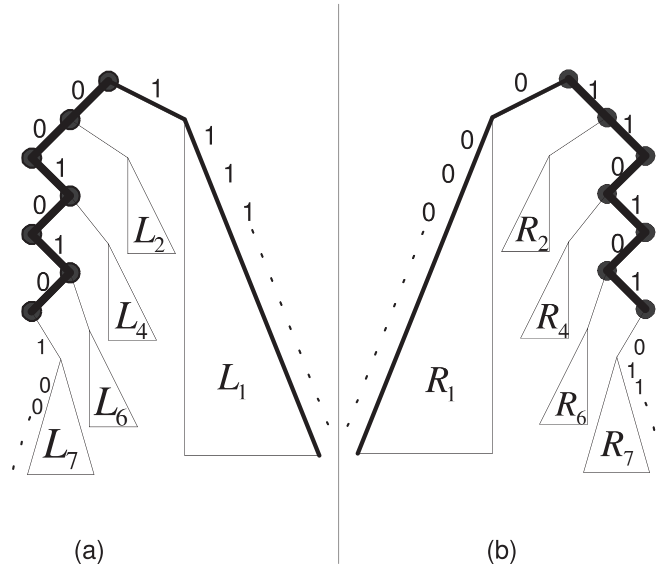

Figure 6: Illustration of the reduction from Huffman-cut-tree problem to the layered-Huffman-forest problem. (a) The case of a left-cut-tree. (b) The case of a right-cut-tree

subproblems that span a prefix of the sequence of blocks. These lemma are the basis for our dynamic programming algorithm. See Figure 7 for an illustration of these lemma.

Lemma 3.2 (Cut) *Let T be a cut-tree with leaves $w_1, \ldots, w_n$ (in that order) where $n \geq 2$. For every $1 \leq i \leq n-1$ there exists two cut-trees T_1 and T_2 with leaves $w_1, \ldots, w_i$ and $w_{i+1}, \ldots, w_n$ respectively such that:*
(1) $llabel(T_1) = llabel(T)$
(2) $rlabel(T_2) = rlabel(T)$
(3) $rcode(T_1) = lcode(T_2)$
(4) $cost(T) = cost(T_1) + cost(T_2)$
(5) T_1 and T_2 can be constructed from T in $O(d_T(lca(w_i, w_{i+1})) + 1)$ time where $lca(w_i, w_{i+1})$ is the lowest common ancestor of w_i and w_{i+1}.

Proof. Let $v = lca(w_i, w_{i+1})$ be the lowest common ancestor of w_i and w_{i+1}, and let $d = d_T(v)$ be the depth of v. We prove the lemma by induction on d. If $d = 0$ we can construct T_1 from T by creating a new node r_1, which will be the root of T_1, and making the left subtree of r_1 the same as the left subtree of v (r_1 has no right child). We create T_2 by creating a new node r_2, and making its right child equal the right child of v. Clearly the resulting trees are cut-trees. Conditions (1) and (2) hold since we did not alter any edge on the left boundary of T_1 or the right-boundary of T_2. Condition (3) holds since in this case $rcode(T_1) = lcode(T_2) = \epsilon$, condition (4) holds since the depth of each leaf remains the same in this construction, and condition (5) holds since we created T_1 and T_2 by a constant number of operations.

Assume the lemma is true for $d = k-1$. We prove it for $d = k$. Let v' be the ancestor of v at depth 1 (v' might be v itself). We assume v' is a left child. The case when v' is a right child is similar. Let T'_1 and T'_2 be the trees created by the induction hypothesis when "cutting" the

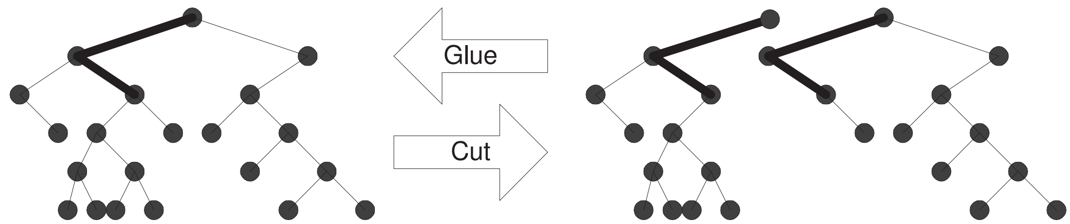

Figure 7: Illustration of the procedure described in Lemma 3.2 and Lemma 3.3. The *Glue* operation takes two trees which agree on their code (*rcode* of the first is equal to the *lcode* of the second) marked as bold edges, and creates one tree. The *Cut* procedure does the opposite.

subtree rooted at v' between w_i and w_{i+1}. We create T_1 by creating a new node r_1, which will be the root of T_1, and hanging T_1' on this new node as a left child, leaving the right child empty. We create T_2 by creating a new node r_2, hanging T_2' on r_2 as a left child, and if the root of T had two children, we hang its right subtree as a right subtree of r_2. It is straightforward to verify using the induction hypothesis that T_1 and T_2 satisfy conditions (1)-(5). ■

Lemma 3.3 (Glue) *Let T_1 be a cut-tree with leaves $w_1, \ldots, w_i$ and let T_2 be a cut-tree with leaves $w_{i+1}, \ldots, w_n$, such that $rcode(T_1) = lcode(T_2)$ then there exists a cut-tree T with leaves $w_1, \ldots, w_n$ such that:*
(1) $llabel(T) = llabel(T_1)$
(2) $rlabel(T) = rlabel(T_2)$
(3) $cost(T) = cost(T_1) + cost(T_2)$
(4) T can be constructed from T_1 and T_2 in $O(length(rcode(T_1)) + 1)$ time.

Proof. Let $c = rcode(T_1) = lcode(T_2)$. We prove the lemma by induction on $length(c)$. First we show it for $length(c) = 0$. In this case $rlabel(T_1) =$ "0111..." and $llabel(T_2) =$ "1000...", i.e. T_1 has only a left child T_L which is a left-cut-tree, and T_2 has only a right child T_R which is a right-cut-tree. We can create T by creating a new node r for the root of T, making T_L the left subtree of r, and making T_R the right subtree of r. Condition (1) holds since $llabel(T) =$ "0".$llabel(T_L) = llabel(T_1)$, and similarly condition (2) holds. Condition (3) is true since all the leaves remain at the same depth, and condition (4) is true since we described a constant number of operations to create T in this case.

Now we assume that the lemma is true for trees with common code c of length $\ell - 1$, and we prove it for trees with common code c of length ℓ. We assume that c starts with a "0". The proof for the other case is similar. In this case T_1 has only a left subtree T_L, and T_2 has a left subtree A and might also have a right subtree B. By our induction hypothesis we can "glue" T_L and A since $rcode(T_L) = lcode(A)$, to get a new tree T'. Now we construct T by creating a new node r for the root of T, and hanging T' as the left child of r. If B exist, we also hang B as the right child of r. It is easy to verify that T is a legal cut tree satisfying conditions (1)-(4). ■

Given a set of weights partitioned into blocks $B_1, \ldots, B_m$ we define the following problems for every $1 \leq k < m$ and binary strings $lc <_p rc$, $length(lc) < h$ and $length(rc) < h$ where h is the upper bound on the depth of the tree which is the solution to the corresponding PAT problem.

Definition 3.4
1) $RCT(rc)$ is the problem of finding the minimal cost Huffman-right-cut-tree T on weights B_1

which has $rcode(T) = rc$
2) $LCT(lc)$ *is the problem of finding the minimal cost Huffman-left-cut-tree* T *on weights* B_m *which has* $lcode(T) = lc$
3) $CT(lc, rc, k)$ *where* $2 \leq k \leq m-1$, *is the problem of finding the minimal cost Huffman-cut-tree* T *on weights* B_k *which has* $lcode(T) = lc$ *and* $rcode(T) = rc$
4) $RPAT(rc, k)$ *is the problem of finding minimal cost right-cut-tree* T, *over the weights* $B_1 \cup \cdots \cup B_k$, *such that all the weights of* B_i *are to the left of the weights of* B_j *for every* $i < j$, *and* $rcode(T) = rc$.

Using Lemma 3.2 and Lemma 3.3 we get the following dynamic programming solution to the PAT problem:

`Procedure` PAT$(B_1, \ldots, B_m)$

1. `Solve` $RCT(rc)$, $LCT(lc)$, `and` $CT(lc, rc, k)$ `for every` $lc <_p rc$, $length(rc) \leq h$, $length(lc) \leq h$, `and` $2 \leq k \leq m-1$. `(using a reduction to the layered Huffman forest problem)`

2. `for every` rc `set` $RPAT(rc, 1) = RCT(rc)$

3. `Loop for every` k `from 2 to` $m-1$

4.   `For every binary string` rc `of length no greater than` h

5.   `Find binary string` $c <_p rc$ `of length no greater than` h, `that brings to minimum the expression` $cost(RPAT(c, k-1)) + cost(CT(c, rc, k))$.

6.   `Set the solution of` $RPAT(rc, k)$ `to be a tree created by gluing` $RPAT(c, k-1)$ `and` $CT(c, rc, k)$ `using the glue lemma.`

7. `Find binary string` c, `of length no greater than` h `that brings to minimum the expression` $cost(RPAT(c, m-1)) + cost(LCT(c))$

8. `return the tree created by gluing` $RPAT(c, m-1)$ `and` $LCT(c)$ `using the glue lemma.`

Theorem 3.1 *The algorithm described above solves the* PAT *problem in* $O(\left(\frac{W_{sum}}{W_{min}}\right)^{2\alpha} \log(\frac{W_{sum}}{W_{min}}) n^2)$ *time and* $O(\left(\frac{W_{sum}}{W_{min}}\right)^{2\alpha})$ *space where* $W_{sum} = \sum_{w \in W} w$, $W_{min} = \min_{w \in W} w$, $n = |W|$, *and* $\alpha = \frac{1}{\log \phi} \approx 1.44$.

Proof. The correctness of the algorithm follows in a straightforward way from Lemma 3.2 and Lemma 3.3. Next we analyze the running time of step 1.

Let n_i, where $1 \leq i \leq m$ be the number of weights in B_i. Each $RCT(rc)$ problem can be solved in $O(hn_1^2)$ time, and each $LCT(lc)$ problem can be solved in $O(hn_m^2)$ time using the reduction to the layered Huffman forest problem. Furthermore, each $CT(rc, lc, k)$ problem can be solved in $O(hn_k^2)$ time. Since there are 2^{h+1} binary strings of length h we have 2^{h+1} $RCT(rc)$ problems, 2^{h+1} $LCT(rc)$ problems and $2^{2(h+1)}$ $CT(rc)$ problems for each block. So the time to solve all these problems is

$$O(2^{h+1}hn_1^2 + 2^{h+1}hn_m^2 + \sum_{k=2}^{m-1} 2^{h+1}2^{h+1}hn_k^2) = O(2^{h+1}hn^2 + 2^{2h+2}hn^2) = O(2^{2h}hn^2)$$

It is easy to verify that this bound on the running time of step 1 dominates the running time of the other steps. So the running time of the algorithm is $O(2^{2h}hn^2)$.

It is easy to see that we can implement the algorithm so that at any one time it is sufficient to store the set of solutions of CT subproblems corresponding to a single block. The number of CT subproblems of one block is $O(2^{2h})$ and this number dominates the number of all other subproblems. If we only want to find the cost of the optimal solution it is sufficient to store the cost of each subproblem. If we want to find the optimal tree itself we can also store for each $RPAT(rc, k)$ subproblem the code c for which $cost(RPAT(rc, k)) = cost(RPAT(c, k-1)) + cost(CT(c, rc, k))$ (this code is found on step 5 of the algorithm). After finding the optimal solution we can recursively reconstruct the optimal tree in the following way: In order to find the solution to $RPAT(rc, k)$ we construct the solution to $RC(c, rc, k)$ and glue it to the solution of $RPAT(c, k-1)$ where c is the code stored for the $RPAT(rc, k)$ subproblem. The total amount of space needed for all the subproblems is $O(2^{2h})$.[7]

The maximum depth of a solution to the PAT problem is bounded by the maximum depth of the optimal alphabetic tree. In Section 5 we show that the depth of optimal alphabetic tree is bounded by $\lfloor \alpha \log(W_{sum}/W_{min}) \rfloor + 1$. Using this upper-bound on h we obtain that the total running time of our algorithm is $O\left(\left(\frac{W_{sum}}{W_{min}}\right)^{2\alpha} \log(\frac{W_{sum}}{W_{min}}) n^2\right)$ and the total space is $O\left(\left(\frac{W_{sum}}{W_{min}}\right)^{2\alpha}\right)$. ∎

4 Layered-Huffman-forests

In this section we present a dynamic programming algorithm to solve the layered-Huffman-forest problem defined in the previous section[8]. Recall that the layered-Huffman-forest problem is to find the minimal cost layered forest over a given multiset of weights $W = \{w_1, \ldots, w_n\}$, hanged on a given multiset of depths $D = \{d_1, \ldots, d_k\}$. We assume without loss of generality that $w_1 \leq w_2 \leq \cdots \leq w_n$, and that $d_1 \geq d_2 \geq \cdots \geq d_k$. Let $F \in LF(W, D)$, we always denote the trees in F by $T_1, T_2, \ldots, T_k$ such that T_i is of depth d_i in the forest.

For every layered-forest $F \in LF(W, D)$, $F = \{T_1, \ldots, T_k\}$, we define a total order $<_F$ on W as follows. For every two leaves $w_i, w_j \in W$, we say that $w_i <_F w_j$ if 1) w_i and w_j are in the same

[7] We assume that a code and the cost of each subproblem can be stored in a constant amount of space.

[8] In the previous section we defined a layered-forest to be a forest of *non-empty* trees. Some applications of the layered-Huffman-forest problem might allow empty trees. The algorithm discussed in this section assumes that empty trees are not allowed, but it can be easily modified to handle forests with empty trees

tree T_a, and w_i appears before w_j in an inorder traversal of T_a, or 2) w_i and w_j are in different trees T_a and T_b respectively, and $a < b$.

Lemma 4.1 *Let $W = \{w_1, \ldots, w_n\}$ be a multiset of weights, and $D = \{d_1, \ldots, d_k\}$ be a multiset of depths. There exists a minimal cost forest $F \in LF(W, D)$ such that w_i is on the left of w_j only if $w_i \leq w_j$ (In other words $w_1 <_F w_2 <_F \cdots <_F w_n$).*

Proof. Let $OLF \subset LF(W, D)$ be the family of all minimal-cost layered-forests in $LF(W, D)$. Let OLF_i, $0 \leq i \leq n$, be the subset of OLF where every forest $F \in OLF_i$ satisfies that $w_1 <_F w_2 <_F \cdots <_F w_i <_F w_j$ for every $j > i$. To prove the lemma we need to show that $OLF_n \neq \emptyset$. We do it by induction on i. OLF_0 is not empty since $OLF_0 = OLF$. Now, assume that $OLF_{i-1} \neq \emptyset$. We prove that $OLF_i \neq \emptyset$ as well.

Let F_{i-1} be a forest in OLF_{i-1}. Let w be the leaf at position i. If $w = w_i$ then $F_{i-1} \in OLF_i$ and $OLF_i \neq \emptyset$. So we assume that $w \neq w_i$. Since trees in OLF_{i-1} are optimal, $d_F(w_i) \geq d_F(w)$ (since otherwise we can get a better tree by swapping w and w_i). Let v be ancestor of w_i, such that $d_F(v) = d_F(w)$. Note that v can be w_i itself. Node v exists since the depth in F of the tree that contains v is not greater than the depth in F of the tree that contains w. We can rearrange the subtree rooted at v, without changing the depth of any leaf, so that w_i will be the leftmost leaf of that subtree[9], and then swap w and v. Note that this process does not change the position of the weights $w_1, \ldots, w_{i-1}$ since they are to the left of w while v is on its right. This process creates a new forest F_i which has the same cost as F_{i-1}, but which has $w_1, \ldots, w_i$ as its leftmost leaves. $OLF_i \neq \emptyset$ since it contains F_i. ■

The following lemma is the basis for our dynamic programming algorithm. The proof is straightforward based on Lemma 4.1.

Lemma 4.2 *Let $W = \{w_1, \ldots, w_n\}$ be a multiset of weights, and $D = \{d_1, \ldots, d_k\}$ be a multiset of depths. Let $F = \{T_1, \ldots, T_k\} \in LF(W, D)$ be a minimal-cost forest with leaves ordered as defined in Lemma 4.1. Let n_k be the number of leaves in T_k. Then:*
1) T_k is the minimal cost tree over leaves $\{w_{n-n_k+1}, w_{n-n_k+2}, \ldots, w_n\}$.
2) $F' = \{T_1, \ldots, T_{k-1}\}$ is the minimum cost forest in $LF(W', D')$ where $W' = \{w_1, \ldots, w_{n-n_k}\}$ and $D' = \{d_1, \ldots, d_{k-1}\}$.
3) $cost(F) = cost(F') + cost(T_k) + d_k \sum_{i=n-n_k+1}^{n} w_i$

Lemma 4.1 and Lemma 4.2 suggest the following dynamic programming solution to the layered-Huffman-forest problem. Let $W = \{w_1, \ldots, w_n\}$ be a multiset of weights, and $D = \{d_1, \ldots, d_k\}$ be a multiset of depths. We define the following problems:

Definition 4.1 *$HTP(\ell, r)$ where $1 \leq \ell \leq r \leq n$ is the problem of finding the minimum cost binary tree over leaves $w_\ell, \ldots, w_r$.*
$HFP(n', k')$ where $1 \leq k' \leq k$ and $k \leq n' \leq n$ is the problem of finding minimum cost forest over leaves $w_1, \ldots, w_{n'}$ with tree depths $d_1, \ldots, d_{k'}$.

[9] This rearrangement is done by swapping any right child node on the path from w_i to v with its sibling

The following dynamic programming algorithm uses Lemma 4.2 to solve the layered-Huffman-forest problem:

Procedure $HuffmanForest(W, D)$

1. For every ℓ and r where $1 \leq \ell \leq r \leq n$ set $SUM(\ell, r) = \sum_{i=\ell}^{r} w_i$

2. Solve $HTP(\ell, r)$ for every ℓ and r where $1 \leq \ell \leq r \leq n$

3. For every $1 \leq n' \leq n$ set $HFP(n', 1) = \{HTP(1, n')\}$ and $cost(HFP(n', 1)) = cost(HTP(1, n')) + d_1 SUM(1, n')$

4. For every $k' = 2$ to k do

5. For every $n' = k'$ to n do

6. Find which $j \in [1, 2, \ldots, (n' - k' + 1)]$ brings the following expression to minimum:
$cost(HFP(n'-j, k'-1)) + cost(HTP(n'-j+1, n')) + d_{k'} SUM(n'-j+1, n')$

7. Set $HFP(n', k') = HFP(n' - j, k' - 1) \cup \{HTP(n' - j + 1, n')\}$

8. Set $cost(HFP(n', k')) =$
$cost(HFP(n'-j, k'-1)) + cost(HTP(n'-j+1, n')) + d_{k'} SUM(n'-j+1, n')$

9. Return $HFP(n, k)$

Theorem 4.1 *The above algorithm solves the layered-Huffman-forest problem in $O(kn^2)$ time, where $n = |W|$ and $k = |D|$.*

Proof. The correctness of the algorithm follows in a straightforward way from Lemma 4.2. The running times of the different steps of the algorithm are as follows.

Step 1 can be solved in $O(n^2)$ time using a simple dynamic programming algorithm. Step 3 takes $O(n)$ time. Step 6 takes $O(n)$ time for each iteration and it is performed $O(nk)$ times, thus it takes a total of $O(kn^2)$ time. Steps 7 and 8 takes $O(1)$ time for each iteration, which gives us a total of $O(kn)$ time for steps 7 and 8. Meanwhile the dominating step is 6, but we did not analyze step 2 yet.

In step 2 we have to solve $n(n-1)/2$ Huffman problems. If we sort the weights in advance we can solve each of those problems in $O(n)$ time. This gives us a total of $O(n^3)$ time for all of the problems. However Knuth in [13] shows a method to solve similar structure of $n(n-1)/2$ minimal-cost *alphabetical* problems in $O(n^2)$ time. Since we know that any Huffman tree can be rearranged so that the leaves will create a monotonic sequence, we can sort the multiset W and solve the alphabetical subproblems using Knuth's method. The alphabetical trees computed by Knuth's method will also be optimal Huffman trees. Thus we can implement step 2 in $O(n^2)$ time.

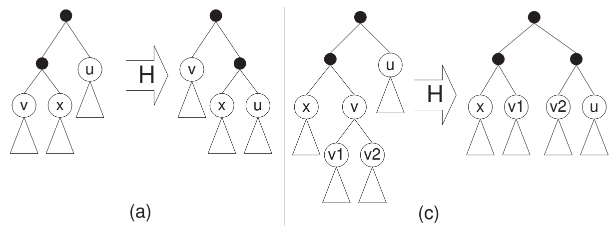

Figure 8: The H transformation. Cases (a) and (c) are presented above. Case (b) is symmetric to case (a), and case (d) is symmetric to case (c)

We obtain that the dominating step of the whole algorithm is step 6, and that the whole algorithm takes $O(kn^2)$ time. ■

5 Maximum depth of alphabetic tree

In this section we show a bound on the depth of a minimal-cost alphabetic tree in terms of $s = \frac{W_{sum}}{W_{min}}$ where $W_{sum} = \sum_{w \in W} w$ and $W_{min} = \min_{w \in W} w$. Katona and Nemetz [12] showed a bound on the depth of a Huffman tree in terms of s. We show that the same bound holds for minimal-cost alphabetic trees. We also show an example that achieves this bound, and thus this bound is tight. The bound we show is:

$$\left\lfloor \frac{1}{\log \phi} \log s + (2 - \frac{1}{\log \phi}) \right\rfloor \approx \lfloor 1.44 \log s + 0.56 \rfloor$$

where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio.

Let T be an alphabetic tree, and let v be a node in T. We denote by $p(v)$ the parent of v, and by $uncle(v)$ the sibling of $p(v)$. The *weight* of an internal node v in T is the sum of the weights of the leaves in the subtree rooted by v. We denote the weight of v by $w(v)$.

Lemma 5.1 *Let T be a minimum-cost alphabetic tree, and let v be an internal node at depth at least 2 in T, then $w(v) \leq w(uncle(v))$.*

Proof. There are four cases regarding whether v and $p(v)$ are left or right children.
(a) Both v and $p(v)$ are left children.
(b) Both v and $p(v)$ are right children.
(c) v is right child and $p(v)$ is left child.
(d) v is left child and $p(v)$ is right child.
Figure 8 shows a transformation H for each of those cases that decreases by one the depth of all the leaves in the subtree rooted at v, and increases by one the depth of all the leaves at the subtree rooted at $uncle(v)$. The transformation H does not change the depth of any other leaf, and it does not change the order of the leaves. If we assume by contradiction that $w(v) > w(uncle(v))$ we can get a tree $T' = H(T)$ with cost less than T. This contradicts the minimality of the cost of T. ■

We denote by F_i where $0 \leq i$, the i-th element of the Fibonacci sequence, i.e. $F_0 = 1$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$ for $i > 1$. The following lemma is proved by induction using Lemma 5.1.

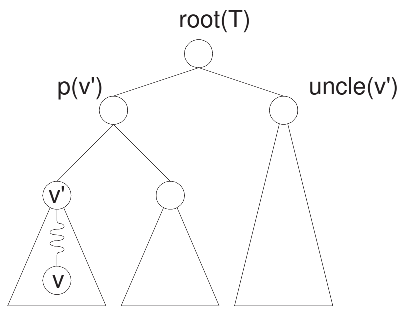

Figure 9: The tree used in the proof of Lemma 5.2.

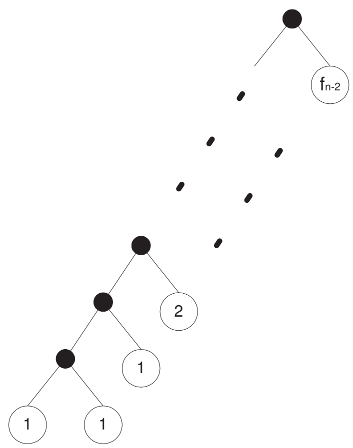

Figure 10: Maximum depth minimum-cost tree for given $\frac{W_{sum}}{W_{min}}$.

Lemma 5.2 *Let T be a minimal-cost alphabetic tree, let v be an internal node at depth $1 \leq h$ in T, then $w(root(T)) \geq F_h w(v)$.*

Proof. We prove the lemma by induction on h. The case when $h = 1$ is trivial since $F_1 = 1$. The case when $h = 2$ is also simple since $w(root(T)) \geq w(v) + w(uncle(v)) \geq 2w(v)$. Now we assume the lemma is true for $h < k$ for some $3 \leq k$ and prove it for $h = k$.

Let v be a node at depth k. Let v' be the ancestor of v at depth 2 (See Figure 9). v' exists since the depth of v is greater or equal to 3. The subtree rooted at v' is a minimal-cost alphabetic tree, and v is a node of depth $k-2$ at that subtree. Thus by induction hypothesis $w(v') \geq F_{k-2}w(v)$. Now, lets look at the subtree rooted at $p(v')$. This subtree is also a minimal-cost alphabetic tree, and the depth of v in that subtree is $k-1$. By induction hypothesis, $w(p(v')) \geq F_{k-1}w(v)$. Now, the lemma is true since $w(root(T)) = w(p(v')) + w(uncle(v'))$, and using Lemma 5.1 we obtain that $w(root(T)) \geq w(p(v')) + w(v') \geq F_{k-1}w(v) + F_{k-2}w(v) = F_k w(v) = F_h w(v)$. ■

The main result of this section is stated by the following theorem.

Theorem 5.1 *Let T be minimal-cost alphabetic tree over the weight sequence $W = (w_1, w_2, \ldots, w_n)$, and let $s = \frac{W_{sum}}{W_{min}}$ then the depth of T, is bounded by the following:*

$$depth(T) = h \leq \left\lfloor \frac{1}{\log \phi}(\log s - 1) \right\rfloor + 2 \approx \lfloor 1.44 \log s + 0.56 \rfloor$$

where ϕ is the golden ratio.

Proof. Let h be the depth of T, and let w be the minimum-weight leaf of depth h in T. Let v be the parent of w. By Lemma 5.2 we know that $W_{sum} \geq F_{h-1}w(v)$. Since $w(v) \geq 2W_{min}$, we get $W_{sum} \geq 2F_{h-1}W_{min}$ and since it is well known that $F_k \geq \phi^{k-1}$, we get $W_{sum} \geq 2\phi^{h-2}W_{min}$. By rearranging this inequality and taking logs we get that $\log \frac{W_{sum}}{W_{min}} \geq (h-2)\log\phi + 1$ from which the lemma follows. ■

Now we show that our bound is tight by giving an example that reaches it. This is in fact the same example given by [12] for Huffman trees. Let $W = (1, 1, 1, 2, 3, 5, \ldots, F_{n-2})$ for some $n \geq 3$. For that sequence we have $\frac{W_{sum}}{W_{min}} = F_n$. The minimal-cost alphabetic tree for that sequence is a long thin branch of depth $n-1$, shown in Figure 10. It is not hard to verify that $\left\lfloor \frac{1}{\log\phi}(\log F_n - 1)\right\rfloor + 2 = n - 1 = h$.

6 Concluding remarks

A partial alphabetic tree over a set of weights W that is partitioned into blocks $B_1, \ldots, B_k$ is a binary tree with the weights in W residing in its leaves such that for every $w \in B_k$ and $w' \in B_{k'}$, for $k < k'$, w is in a leaf which is to the left of the leaf of w'.

We describe a pseudo polynomial algorithm that finds an optimal partial alphabetic tree for a given set of weights partitioned into blocks. The tree is optimal in the sense that it minimizes the average path length over all partial alphabetic trees with weights W. Our technique is general and applies to several other objective functions as well.

We know of several special cases of the PAT problem for which a polynomial time algorithm exists. Two such special cases are the classical Huffman problem and the classical optimal alphabetic tree problem. Other special cases appear in Section 2.

Perhaps the most intriguing problem left open by our work is whether a strongly polynomial time algorithm exists for the partial alphabetic tree problem.

The partial alphabetic tree problem generalizes nicely to arbitrary partial orders as follows. Let $W = \{w_1, \ldots, w_n\}$ be a multiset of nonnegative weights, and let $\prec$ be a partial order defined on W. Let $\mathcal{T}$ be the set of binary trees each having the weights in W as leaves such that if $w_i \prec w_j$ then w_i precedes w_j in an inorder traversal of the tree. The problem is to find a tree $T \in \mathcal{T}$ that minimizes $\sum_{i=1}^{n} w_i d(w_i)$, where $d(w_i)$ is the depth of w_i in T. The complexity status of this problem is also open.

References

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In 12^{th} *Annual ACM Symposium on Discrete Algorithms (SODA)*, pages 547–556, 2001.

[2] Julia Abrahams. Code and parse trees for lossless source encoding. *Communications in Information and Systems (CIS)*, 1(2):113–146, April 2001.

[3] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2002.

[4] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–264. ACM Press, 2002.

[5] Arye Barkan and Haim Kaplan. Partial alphabetic trees. In Rolf Mohring and Rajeev Raman, editors, *Algorithms - ESA 2002*, volume 2461, pages 114–125. Springer, September 2002.

[6] A. M. Garsia and M. L. Wachs. A new algorithm for minimum cost binary trees. *SIAM J. Comput*, 6(4):622–642, Dec 1977.

[7] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38:933–967, July 1959.

[8] T. C. Hu and A. C. Tucker. Optimum computer search trees. *SIAM J. Appl. Math.*, 21:514–532, 1971.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc IRE, 40*, pages 1098–1101, 1952.

[10] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2002.

[11] Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter. Correctness of constructing optimal alphabetic trees revisited. *Theoretical Computer Science*, 180(1-2):309–324, June 1997.

[12] G. O. H. Katona and T. O. H. Nemetz. Huffman codes and self information. *IEEE Trans. on Information Theory*, IT-22(3):337–340, May 1976.

[13] D. E. Knuth. Optimum binary search trees. *Acta Infomatica*, 1:14–25, 1971.

[14] M. Thorup and U. Zwick. Compact routing schemes. In 13^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.