On the Huffman and Alphabetic Tree Problem with General Cost Functions[1]

Hiroshi Fujiwara[2]
Toyohashi University of Technology
h-fujiwara@cs.tut.ac.jp

Tobias Jacobs[3]
National Institute of Informatics
jacobs@nii.ac.jp

Abstract

We study a wide generalization of two classical problems, the Huffman Tree and Alphabetic Tree Problem. We assume that the cost caused by the ith leaf is $f_i(d_i)$, where d_i is its depth in the tree under consideration, and $f_i : \mathbb{N}_0 \to \mathbb{R}_0^+$ is an arbitrary function. All solution methods known for the classical cases fail to compute the optimum here.

For the generalized Alphabetic Tree Problem, we give a dynamic programming algorithm solving it in time $O(n^4)$, using space $O(n^3)$. Furthermore, we show that the runtime can be reduced to $O(n^3)$ if the cost functions are nondecreasing and convex. The improved algorithm can also be used in the setting where the cost functions are nondecreasing and the objective function is the maximum leaf cost.

We also prove that the Huffman Tree Problem in its full generality is inapproximable unless P=NP, no matter if the objective function is the sum of leaf costs or their maximum. For the latter problem, we show that the case where the cost functions are nondecreasing admits a polynomial time algorithm.

1 Introduction

Computing minimum cost binary trees is a classical combinatorial problem having applications in various areas of informatics. Given a set $\{\ell_1, , \ldots, \ell_n\}$ of leaves having weights $\{w_1, \ldots, w_n\}$, the famous Huffman Tree Problem describes the task to compute a binary tree T with leaf set $\{\ell_1, , \ldots, \ell_n\}$, such that the weighted total distance between the tree root and a leaf is minimized. The Alphabetic Tree Problem differs from the Huffman Problem by the additional constraint that the left-to-right order of the leaves in the solution tree must be exactly $\ell_1, , \ldots \ell_n$. The objective function of both versions is given as

$$\sum_{i=1}^{n} w_i \cdot \mathrm{dist}(\mathrm{root}(T), \ell_i) \ .$$

In the above model it is assumed that the access cost of a leaf is proportional to its depth in the tree. In this work we investigate a generalized problem: we assume that the cost of leaf ℓ_i having distance d_i from the root is determined by $f_i(d_i)$, where $f_i : \mathbb{N}_0 \to \mathbb{R}_0^+$ is an arbitrary function. Instances of our generalized problem are determined by the n cost functions $f_1, \ldots, f_n$, one for each leaf. The corresponding optimization problems can be formulated as follows.

> **General Cost Huffman Tree Problem, GHT** Given n arbitrary functions $f_1, \ldots, f_n : \mathbb{N}_0 \to \mathbb{R}_0^+$, the objective of GHT is to determine a binary tree T having n leaves and a bijection $g : \{1, \ldots, n\} \to \mathrm{leaves}(T)$ such that $\sum_{i=1}^{n} f_i(\mathrm{depth}(g(i), T))$ is minimized, where $\mathrm{depth}(g(i), T)$ is the distance between the root of T and the leaf $g(i)$.

[1]The original publication is available at www.springerlink.com.
[2]This work was supported by KAKENHI (19700015).
[3]This work was supported by a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD).

General Cost Alphabetic Tree Problem, GAT Given n arbitrary functions $f_1, \ldots, f_n : \mathbb{N}_0 \to \mathbb{R}_0^+$, the objective of GAT is to determine a binary tree T whose leaves in left-to-right order are $\ell_1, \ldots, \ell_n$, such that $\sum_{i=1}^{n} f_i(\text{depth}(\ell_i, T))$ is minimized.

We also investigate problems **max-GHT** and **max-GAT**, where the objective function is $\max_{i=1}^{n} f_i(\text{depth}(g(i), T))$ and $\max_{i=1}^{n} f_i(\text{depth}(\ell_i, T))$, respectively.

Related Work

The Huffman Tree Problem is named after D. A. Huffman [Huf52], who gave an algorithm solving it in time $O(n \log n)$, which is the fastest possible. For the Alphabetic Tree Problem, the first polynomial time algorithm was a dynamic programming approach having a runtime of $O(n^3)$, proposed by Gilbert and Moore [GM59]. Knuth [Knu71] identified a property of optimal alphabetic trees that admits a speedup of the DP algorithm. The resulting runtime is $O(n^2)$. The $O(n \log n)$ time method discovered by Hu and Tucker [HT71] uses a bottom-up approach which resembles the Huffman-Algorithm.

Up to today it is unknown whether the Alphabetic Tree Problem can be solved in time $o(n \log n)$. However, for certain classes of special weight assignments, linear time algorithms were given [KM93, Hu73], and for certain classes of algorithms $\Theta(n \log n)$ was shown to be a lower bound for the runtime [KM93]. Flajolet and Prodinger [FP87] gave an asymptotic estimate of the number of feasible solutions to the Alphabetic Tree Problem.

Efforts to solve the Alphabetic Tree Problem with non-linear costs were made by Hu et al. in [HKT79]. The authors identified a class of cost functions where the Hu-Tucker Algorithm is applicable, including power summations of the type $\text{cost}(T) = w_i t^{d_i}$ for any $t \geq 1$. Baer [Bae10] recently showed that for $t < 1$ neither the Hu-Tucker algorithm nor the approach of Knuth leads to optimal solutions, and he proposes to use the $O(n^3)$ algorithm by Gilbert and Moore instead.

Another direction of generalization is to impose additional constraints on the structure of the output tree. The Alphabetic Tree Problem can be interpreted as the task to determine an optimal binary search strategy for a totally ordered set. A considerable number of papers about search in partially ordered sets have been published, see e.g. [CDKL04, MOW08, CJLM10]. In the even more general *Binary Identification Problem*, the input is a number of subsets $S_1, \ldots, S_m$ of leaves, and the goal is to compute a minimum cost search strategy using queries of the type "is leaf ℓ_i in set S_j or not?". Recent results from this area can be found in e.g. [CPR$^+$07, AH08, CPRS09].

Our Results

We contribute a number of insights concerning the General Cost Alphabetic Tree (GAT) and Huffman Tree Problem (GHT), including the versions where the maximum leaf cost has to be minimized instead of the sum.

In Section 2 we show that an extension of the Gilbert-Moore Algorithm solves GAT in time $O(n^4)$ and space $O(n^3)$, regardless of the cost functions. We then define two properties of cost functions, *subtree optimality* and *structural continuity*. Both properties admit a speedup of the algorithm by the factor of n. The speedups are independent from each other, so problem instances whose cost functions satisfy both properties admit a $O(n^2)$ time optimal algorithm, which happens to be exactly the method proposed by Knuth [Knu71].

We prove that if the cost functions are nondecreasing and convex, then the property of structural continuity is satisfied. Therefore, this case of the Alphabetic Tree Problem can be solved in time $O(n^3)$. Furthermore, we show that for max-GAT instances to satisfy structural

continuity it even suffices that the cost functions are nondecreasing. These results can be found in Section 3.

In Section 4 we address the Huffman Tree Problem. We show for both GHT and max-GHT that it is NP-hard to decide whether an instance admits a cost zero solution. This means that those problems are much harder than their Alphabetic Tree counterparts: they are inapproximable unless P=NP. As a positive result, we show that max-GHT admits a polynomial time algorithm if the cost functions are nondecreasing. The computational tractability of standard GHT with nondecreasing cost functions remains an open problem.

2 Dynamic Programming Algorithm for GAT

This section begins with a recapitulation of the Gilbert-Moore algorithm for the classical Alphabetic Tree Problem. We believe that the property of *subtree optimality* can be well understood in the context of that algorithm, because this property is essentially required for its correctness. Subsequently, we give an extended algorithm which solves GAT without requiring subtree optimality. The runtime and space requirements of that algorithm are however by a factor of n higher. We then introduce the property of *structural continuity* and show how it helps to make the algorithm more time-efficient again.

The Gilbert-Moore Algorithm and Subtree Optimality

The algorithm by Gilbert and Moore employs a dynamic programming approach to solve the classical problem with $f_i(x) = w_i x$ for $i = 1, \ldots, n$. Subproblems are determined by two integer parameters (l, r) with $1 \leq l \leq r \leq n$. A subproblem (l, r) asks about an optimal alphabetic tree for $f_l, \ldots, f_r$. The value $\tilde{c}(l, r)$ of an optimal solution to that subproblem is calculated recursively as $\min_{l \leq i < r}(\tilde{c}(l, i) + \tilde{c}(i + 1, r))$, and the optimal search tree is obtained by making the optimal solution tree to (l, j) and $(j+1, r)$ the left and right subtree of the root, respectively, where j is the value of i for which the minimum is reached in the above formula. In the basic case of $l = r$, the optimal tree only consists of one leaf.

There are $O(n^2)$ different subproblems, and each of them requires computation time $O(n)$, so the overall runtime is $O(n^3)$, while the space requirements are $O(n^2)$. The algorithm successively merges optimal alphabetic trees for subsequences of cost functions into optimal trees for larger subsequences. The reason why this works out is because an optimal tree for $f_1, \ldots, f_n$ is always the combination of optimal trees for $f_1, \ldots, f_i$ and $f_{i+1}, \ldots, f_n$, for some $i \in \{1, \ldots, n\}$. This property is called *subtree optimality*. Note that by induction it follows that for any internal node v in an optimal alphabetic tree, the subtree under v is an optimal alphabetic tree for the sequence of leaves that are descendants of v.

Algorithm for GAT

A simple counterexample (see full paper) shows that GAT is not subtree-optimal in general, i.e. the left and right subtree under the root of an optimal alphabetic tree are not necessarily optimal alphabetic trees.

For some problem instance $(f_1, \ldots, f_n)$, assume that i is such that the leaves $\ell_1, \ldots, \ell_i$ and the leaves $\ell_{i+1}, \ldots, \ell_n$ are in the left and right subtree T_1 and T_2 under the root of an optimal

alphabetic tree T, respectively. We have that

$$\begin{aligned}\mathrm{cost}(T) &= \sum_{1\le j\le i} f_j(\mathrm{depth}(\ell_j, T)) + \sum_{i<j\le n} f_j(\mathrm{depth}(\ell_j, T)) \\ &= \sum_{1\le j\le i} f_j(\mathrm{depth}(\ell_j, T_1)+1) + \sum_{i<j\le n} f_j(\mathrm{depth}(\ell_j, T_2)+1)\ .\end{aligned}$$

The optimality of T implies that the structure of say T_1 must be such that the term $\sum_{1\le j\le i} f_i(\mathrm{depth}(\ell_i, T_1)+1)$ is minimized. This differs from the optimization term for problem instance $(f_1, \ldots, f_i)$ only by the offset of 1 that is added to each depth value. By the same argument, the two subtrees under the root of T_1 are optimal alphabetic trees with respect to the cost function where an offset of 2 is added to the depth values before applying the f_js, and so on.

This offset is added as an additional parameter to the description of subproblems, so subproblem (l, r, k) is to determine an alphabetic tree T' having leaves $\ell_l, \ldots, \ell_r$ so as to minimize $\sum_{i=l}^{r} f_i(\mathrm{depth}(\ell_i, T') + k)$. It can also be interpreted as the task to compute a tree T' which minimizes the sum of access costs under the assumption that the root of T' is appended to a path of length k.

The cost $\tilde{c}$ of an optimal solution to a subproblem is calculated as

$$\tilde{c}(l,r,k) = \begin{cases} f_r(k) & \text{if } l = r \\ \min_{i=l}^{r-1}\{\tilde{c}(l,i,k+1) + \tilde{c}(i+1,r,k+1)\} & \text{otherwise}\ . \end{cases} \tag{1}$$

The original problem instance I is given as subproblem $(1, n, 0)$. Each time a new subproblem with k incremented by one is generated, the difference between l and r decreases by at least one, which implies that k never grows larger than n. Consequently, we have no more than $O(n^3)$ different subproblems. Each subproblem requires an effort of $O(n)$, so the runtime of this algorithm is $O(n^4)$.

Structural Continuity

The property of *structural continuity* was proven by Knuth [Knu71] to hold for the classical Alphabetic Tree Problem. It roughly states that the root of an optimal alphabetic tree can only move left when the interval under consideration is extended to the left. For making this more precise, recall that the root of an alphabetic tree divides the sequence of leaves into a left and a right subsequence, $\ell_1, \ldots, \ell_i$ and $\ell_{i+1}, \ldots, \ell_n$. We say that i is the position of the root. Now assume that i is the position of the root of an optimal tree for subsequence $\ell_l, \ldots, \ell_r$. Then the property of structural continuity guarantees that there is an optimal alphabetic tree for subsequence $\ell_{l-1}, \ldots, \ell_r$ where the root is at a position smaller than or equal to i. Symmetrically, for $\ell_l, \ldots, \ell_{r+1}$, there is an optimal solution where the index of the root's position is not smaller than i.

Structural continuity yields an improved algorithm for the GAT problem. This algorithm computes optimal solutions to the subproblems in the following order: For $a = 0, \ldots, n-1$, for $k = 0, \ldots, n$, compute the solutions to all subproblems (l, r, k) with $r - l = a$. It is not hard to see that this order of computation guarantees that each solution to a subproblem is computed before it is needed during the computation of another subproblem's optimal solution.

We reason about the behavior of the algorithm during some fixed assignment of a and k. Denote the choice of i in Equation 1 as $i[l, r, k]$. For $j = 1, \ldots, n-a$, the optimal solution to $(j, j+a, k)$ is computed. Structural continuity implies that $i[j, j+a-1, k] \le i[j, j+a, k] \le i[j+1, j+a, k]$. The values of $i[j, j+a-1, k]$ and $i[j+1, j+a, k]$ have already been determined due to the order of computation. There are only $i[j+1, j+a, k] - i[j, j+a-1, k] + 1$ possible values for $i[j, j+a, k]$ to be considered by the algorithm. Summing them up for $i[j, j+a, k]$,

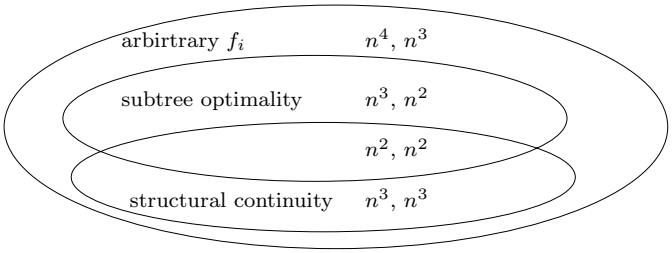

Figure 1: Runtime and space requirements of the DP approach for GAT.

$j = 1, \ldots, n - a$, results in a telescope sum which evaluates to $O(n)$. There are $O(n^2)$ different configurations of a and k, so the improved runtime is $O(n^3)$.

If both subset optimality and structural continuity holds, the Gilbert-Moore algorithm can modified in a similar manner to obtain runtime $O(n^2)$, details can be found in [Knu71]. The resource requirements of the dynamic programming approach are visualized in Figure 1.

3 Cost functions satisfying structural continuity

In this section we prove that structural continuity is satisfied by GAT instances whose cost functions are nondecreasing and convex. We also show that nondecreasing cost functions lead to structural continuity in the case of max-GAT.

Theorem 1. *Let $(f_1, \ldots, f_n)$ be an instance of GAT where the cost functions are nondecreasing and convex, i.e. $0 \leq f_i(x) - f_i(x-1) \leq f_i(x+1) - f_i(x)$ for each $i = 1, \ldots, n$ and $x \geq 1$. Then the instance satisfies structural continuity.*

For proving the theorem, we use an alternative characterization of the GAT problem. Imagine the infinite binary tree $\mathcal{T}$, where every node is an internal one. Given the functions $f_1, \ldots, f_n$, we seek to find a minimum cost injective assignment g from the index set $\{1, \ldots, n\}$ to nodes of $\mathcal{T}$, under the restriction that any downward path in $\mathcal{T}$ encounters at most one of these n indices. Furthermore, the assignment has to be alphabetical, i.e. for any two indices $i < j$, $g(i)$ and $g(j)$ must have a common ancestor w such that $g(i)$ is in the left and $g(j)$ is in the right subtree under w. The objective function to be minimized is $\sum_{i=1}^{n} f_i(\text{depth}(g(i), \mathcal{T}))$.

As the f_is are nondecreasing, it is not hard to verify that there always exists an optimal solution where any infinite downward path starting at the root of $\mathcal{T}$ contains a node some index is assigned to. Assignments satisfying that property are called *regular*. When g is regular, then, by turning the n nodes in $g(\{1, \ldots, n\})$ into leaves, one obtains a finite tree that has n nodes and is a solution to the GAT instance. Conversely, any solution for the GAT problem can be interpreted as a regular assignment g having the same cost.

The subproblems considered by the dynamic programming approach also have a natural interpretation in terms of $\mathcal{T}$. Subproblem (l, r, k) describes the task to find a minimum cost regular assignment of the index set $\{l, \ldots, r\}$ to $\mathcal{T}$ under the same restrictions as above, but with each f_i replaced with f_i^{+k}, where $f_i^{+k}(d) := f_i(d + k)$.

Theorem 1 claims that the root of the optimal tree never has to move right when the interval under consideration is extended to the left. In that proposition, the sequence of functions is interpreted to be fixed, while the tree structure is flexible and adapts to the subproblem under consideration. However, from the point of view just introduced, the tree $\mathcal{T}$ is fixed and the assignment g of the index set to nodes in $\mathcal{T}$ is the flexible part. In this context, Theorem 1 claims that indices never move left when the interval is extended to the left.

To make our statements more formal, we need to introduce some more notation. Let i be an index that is assigned to a node of $\mathcal{T}$ by g. Assume that g is modified. We say that i moves upwards/downwards, if its new position is an ancestor/descendant of its old position. We further say that i moves left/right, when its new position is left/right of its old position, where a node u is defined to be left (right) of node v, when there exists a node w in $\mathcal{T}$ such that u is in the left (right) and v is in the right (left) subtree under w. Any movement of i is either an upward, downward, left, or right movement. The proposition we are going to show is actually more general than Theorem 1. The complete proof of the following lemma will appear in the full version of this paper.

Lemma 1. *Let g be the assignment corresponding to an optimal solution to (l, r, k). Then there is a regular optimal solution g' to $(l-1, r, k)$ such that for any index $i = l, \ldots, r$, it either holds that $g'(i) = g(i)$, $g'(i)$ is right of $g(i)$, or $g'(i)$ is a descendant of $g(i)$.*

Proof (sketched). Assume for contradiction that there is some index i moving upwards or left, no matter which optimal regular solution g' is chosen. Let i the minimum index with that behavior in some optimal solution g'.

If $i = l$, a contradiction is caused by the fact that i is assigned to the leftmost path of $\mathcal{T}$ by g, but g' must assign index $i-1$ to a node on this path.

If $i > l$, first consider the case that index i moves upwards. We split the transformation from g into g' into the transformation from $g|\{l, \ldots, i-1\}$ into $g'|\{l-1, \ldots, i-1\}$, and the transformation from $g|\{i, \ldots, r\}$ into $g'|\{i, \ldots, r\}$. The further transformation only changes the positions of indices $< i$, and the latter only affects indices $\geq i$. As index $i-1$ moves neither left nor upwards, these two transformations take place in non-overlapping parts of $\mathcal{T}$ and can therefore be performed independently of each others. Only the transformation from $g|\{i, \ldots, r\}$ into $g'|\{i, \ldots, r\}$ causes indices to move left or downwards, and from the optimality of g and g' we conclude that the solution obtained from g by only performing the first transformation is as good as g'.

For analyzing the case where i moves left, observe that $i-1$ can only move downwards in order to make room for i. We define an alternative solution h' for $(l-1, r, k)$. Solution h' is constructed starting from g', by changing the positions of the indices $i, \ldots, r$ to the positions they are assigned to by g. After that, we can move index $i-1$ a certain number of steps upwards. We then show that h' is optimal for $(l-1, r, k)$, although no index moves down or left during the transformation from g into h'. □

Proof of Theorem 1. Let i be the position of the root in an alphabetic tree T for subproblem (l, r, k). The corresponding function g assigns indices $l, \ldots, i$ to the left subtree under the root of $\mathcal{T}$, and indices $i+1, \ldots, r$ are assigned to the right subtree. Conversely, the position of the root of T is exactly the largest index being assigned to the left subtree under the root of $\mathcal{T}$.

So when during the transformation from g to g' no index moves left, no index larger than i can move into the left subtree under the root of $\mathcal{T}$, and this means that the root of the alphabetic tree T' corresponding to g' cannot have a greater position than the root of T. Using this argumentation, Theorem 1 follows immediately from Lemma 1. □

Theorem 2. *Let $(f_1, \ldots, f_n)$ be an instance of max-GAT where the cost functions are nondecreasing, i.e. $f_i(x) \geq f_i(x-1)$ for each $i = 1, \ldots, n$ and $x > 1$. Then the instance satisfies structural continuity.*

This theorem is much simpler to prove than Theorem 1. The argumentation is based on a simple observation about the cost $\tilde{c}(l, r, k)$ of the optimal solution to subproblem (l, r, k). The correctness of the following lemma is established by the fact that any solution to $(l -$

$1, r, k)$ can easily be transformed into a cheaper solution to (l, r, k) when the cost functions are nondecreasing.

Lemma 2. *For any instance of max-GAT with nondecreasing cost functions and any subproblem (l, r, k) of it, it holds that $\tilde{c}(l, r, k) \leq \tilde{c}(l-1, r, k)$ and $\tilde{c}(l, r, k) \leq \tilde{c}(l, r+1, k)$.*

Proof of Theorem 2. Let i be the position of the root in an optimal solution T to subproblem (l, r, k). For some $j > i$, let T' be an optimal solution to subproblem $(l-1, r, k)$ with the root's position at j. We show that the tree T'', which is defined as the best solution to $(l-1, r, k)$ having the root at position i, is not more expensive than T'. The cost of T'' is given as $\text{cost}(T'') = \max\{\tilde{c}(l-1, i, k+1), \tilde{c}(i+1, r, k+1)\}$.

Let us first assume that the maximum in the formula for $\text{cost}(T'')$ is defined by the first term, i.e. $\tilde{c}(l-1, i, k+1) \geq \tilde{c}(i+1, r, k+1)$. Multiple application of Lemma 2 gives

$$\tilde{c}(l-1, j, k+1) \geq \tilde{c}(l-1, i, k+1) \geq \tilde{c}(i+1, r, k+1) \geq \tilde{c}(j+1, r, k+1) \ ,$$

and this implies that $\text{cost}(T') = \max\{\tilde{c}(l-1, j, k+1), \tilde{c}(j+1, r, k+1)\}$ is not smaller than $\text{cost}(T'')$.

Now we assume that the second term establishes the maximum, i.e. $\tilde{c}(l-1, i, k+1) \leq \tilde{c}(i+1, r, k+1)$. By Lemma 2, $\tilde{c}(l, i, k+1) \leq \tilde{c}(i+1, r, k+1)$, which means that $\tilde{c}(i+1, r, k+1)$ is the maximum in the cost term for T as well. In other words, $\text{cost}(T) = \text{cost}(T'')$. Lemma 2 states that no solution to $(l-1, r, k+1)$, including T', can be cheaper than T. Therefore, T'' is optimal. □

4 Huffman Tree Problem

In this section we address the General Cost Huffman Tree Problem (GHT). In the classical linear cost model, this problem can be reduced to the Alphabetic Tree Problem by sorting the nodes by weight. A simple counterexample (to appear in the full paper) demonstrates that this reduction does not work correctly in the case of general costs, even if the cost functions are monotonic and convex. We show that GHT in its full generality is inapproximable unless P=NP, which holds for both GHT and max-GHT. Subsequently, we prove that the latter problem admits a polynomial time algorithm if the cost functions are nonincreasing.

Complexity of GHT

The computational complexity of sum-GHT and max-GHT will both be settled by one reduction from the 3-Set Cover Problem, which is well-known to be NP-hard [Kar72].

> **Exact Cover by 3-Sets, X3C.** Given some set C with $|C| = 3k$, $k \in \mathbb{N}$, and a collection D of 3 element subsets of C, the problem X3C is to decide whether there is a sub-collection $D' \subseteq D$, such that each element of C occurs in exactly one member of D'.

Let (C, D) be an instance of X3C with $|D| = m$ and $|C| = n$. In the following, we show how to construct an equivalent instance I of GHT. We consider the solution tree for instance I to be partitioned into m layers, each consisting of three levels. In order to simplify notation, let $l_q^i = 3(i-1) + q$ denote the qth level of the ith layer for $i = 1, \ldots, m$ and $q = 1, 2, 3$. The tree root level 0 is not considered to be in one of the layers.

In instance I there are three different types of cost functions. First, there are functions f_2^i and f_3^i for $i = 1, \ldots, m$. For $i < m$, f_q^i is defined as $f_q^i(l_q^i) = 0$ and $f_q^i(x) = 1$ for any $x \neq l_q^i$. The mth pair is defined as $f_2^m(x) = f_3^m(x) = 0$ for $x = l_2^m$, and $f_2^m(x) = f_3^m(x) = 1$ otherwise. Note that there are no functions f_1^i.

Second, we introduce $m-k$ functions $g_1, \ldots, g_{m-k}$. Those functions are all identical, for $t = 1, \ldots, m-k$ they are defined as $g_t(l_1^i) = 0$ for $i = 1, \ldots, m$, and $g_t(x) = 1$ for all other values of x.

Finally, I contains n different functions $h_1, \ldots, h_n$, one for each element of $C = \{c_1, \ldots, c_n\}$. Let $D = \{D_1, \ldots, D_m\}$. For each $i = 1, \ldots, m$, select one element $d_i \in D_i$ arbitrarily. Now, for $j = 1, \ldots, n$, define

$$h_j(x) = \begin{cases} 0 & \text{if } x = l_2^i \text{ for some } i \text{ with } c_j = d_i \\ 0 & \text{if } x = l_3^i \text{for some } i \text{ with } c_j \in D_i \setminus \{d_i\} \\ 1 & \text{otherwise .} \end{cases}$$

Lemma 3. *There is a solution to instance I having cost 0 if and only if instance (C, D) admits an exact cover.*

Proof. "⇒" Assume that there is a solution T to instance I having cost 0. Then the leaf associated with function f_q^i must be on level l_q^i for $q = 1, 2$ and $i = 1, \ldots, m-1$, and the leaves associated with f_2^m and f_3^m must be on level l_2^m. Because of the leaves on level l_2^m, there must be a downward path $v_1^1, v_2^1, v_3^1, v_1^2, \ldots, v_2^m$ starting in the root v_1^1 of T, such that v_q^i is on level $l_q^i - 1$, and v_2^m is the parent of the leaves associated with f_2^m and f_3^m.

We can assume that in T, for $1 \leq i \leq m-1$, the internal nodes v_2^i and v_3^i are the parents of the leaves associated with f_2^i and f_3^i, respectively. If this property does not hold, then the tree T can be modified in order to satisfy it: simply interchange children between v_t^i and the parent of the leaf associated with f_t^i appropriately. This does not change the level of any leaf, so the cost of T remains zero.

Now consider the subtrees $T_1, \ldots, T_m$, where T_i is defined as the subtree under v_1^i which does not contain v_2^i. The set of leaves associated with the the g-type and h-type functions is exactly the set of leaves being in those subtrees. For $0 \leq i \leq m$, the unique h_j with $c_j = d_i$ is the only function besides f_2^i which evaluates to 0 for input l_2^i. Furthermore, only the g-type functions evaluate to 0 for input l_1^i. Therefore, if some T_i does not contain any g-type leaf, then it has at least three leaves which are associated with h-type functions.

As there are only $m-k$ functions of type g, k of the m subtrees T_i must contain at least three h-type leaves. As the total number of h-type leaves is $n = 3k$, each of those subtrees must contain exactly three of them.

Let T_i be such a subtree. Function h_j with $c_j = d_i$ is the only function besides f_2^i which evaluates to 0 for input l_2^i, and $h_{j'}, h_{j''}$ with $\{c_{j'}, c_{j''}\} = D_i \setminus \{d_i\}$ are the only two functions besides f_3^i evaluating to 0 for input l_3^i. T_i must contain exactly those three functions, because otherwise it would have more than three leaves.

As the number of subtrees T_i of this kind is m, and each h-type function can only occur in one of them, the corresponding selection of D_is must be an exact cover of U.

"⇐" If $D' \subset D$ is an exact cover, then one can construct a zero cost solution tree T from it which has the structure just described. □

We have given a reduction showing that it is NP-hard to decide whether a GHT instance admits a zero cost solution. This establishes the following theorem.

Theorem 3. *GHT and max-GHT are inapproximable unless P=NP.*

Max-GHT with Monotonic Costs

We give a polynomial time algorithm for the version of GHT where the cost functions are monotonic and the objective is to minimize the maximum cost caused by a leaf.

Let $(f_1, \ldots, f_n)$ be an instance of max-GHT. The cost of any solution is fully characterized by the *level assignment* function $d : \{1, \ldots, n\} \to \{0, \ldots, n\}$, which assigns the tree level of the corresponding leaves to indices of the cost functions (note that no leaf can have a depth greater than n). Given d, the cost of the corresponding tree can be calculated as $\text{cost}(d) = \max_i f_i(d(i))$.

Our approach is to compute an optimal level assignment function and then derive an optimal tree from it. Given d, let $\bar{d} : j \mapsto |\{d(i) = j\}|$ be the function which assigns to each tree level the number of leaves assigned to it by d.

Lemma 4. *A function $d : \{1, \ldots, n\} \to \{0, \ldots, n\}$ is the level assignment function of a binary tree having n leaves, if and only if*

$$\sum_{k=0}^{n} \bar{d}(k) 2^{n+1-k} = 2^{n+1} \ . \tag{2}$$

Furthermore, for any level assignment function satisfying that equation, a corresponding binary tree can be computed in polynomial time.

Proof. "⇒": Let d be the level function of a binary tree T having leaves $\ell_1, \ldots, \ell_n$. For $i = 1, \ldots, n$, replace ℓ_i with a full binary tree having depth

$$(n+1) - \text{depth}(\ell_i, T) = (n+1) - d(i) \ .$$

By this transformation, we obtain the full binary tree $\mathcal{T}_{n+1}$ having depth $n+1$. For $i = 1, \ldots, n$, the tree replacing ℓ_i has $2^{(n+1)-d(i)}$ leaves. The claim follows from the fact that $\mathcal{T}_{n+1}$ has 2^{n+1} leaves.

"⇐": From the level function d we construct a tree T level by level, keeping track of the total number $v(j)$ of nodes (internal nodes and leaves) on each level j. Our construction will maintain the invariant

$$\sum_{k=0}^{j-1} \bar{d}(k) 2^{n+1-k} + v(j) 2^{n+1-j} = 2^{n+1} \ .$$

If $n = 1$, then it must hold that $f(1) = 0$, so we can simply place the only leaf at the root of T. Otherwise, we place an internal node at the root of T at level 0. In both cases, the above invariant is established with respect to $j = 0$.

For $1 \leq j \leq n$, assume that we have already created level $0, \ldots, j-1$ of T and have established

$$\sum_{k=0}^{j-2} \bar{d}(k) 2^{n+1-k} + v(j-1) 2^{n+1-(j-1)} = 2^{n+1} \ .$$

We have $v(j-1) - d(j-1)$ internal nodes at level $j-1$, so the total number of nodes on level j is $v(j) = 2v(j-1) - 2d(j-1))$. This implies $v(j) 2^{n+1-j} = v(j-1) 2^{n+1-(j-1)} - d(j-1) 2^{n+1-(j-1)}$, so

$$\sum_{k=0}^{j-1} \bar{d}(k) 2^{n+1-k} + v(j) 2^{n+1-j} = \sum_{k=0}^{j-2} \bar{d}(k) 2^{n+1-k} + v(j-1) 2^{n+1-(j-1)} = 2^{n+1} \ ,$$

which establishes the invariant with respect to j. For showing that level j of the tree can be constructed, we need to prove that $\bar{d}(j) \leq v(j)$. This can be shown using the invariant: $2^{n+1-j} v(j) =$

$$2^{n+1} - \sum_{k=0}^{j-1} \bar{d}(k) 2^{n+1-k} \geq 2^{n+1} - \left(\sum_{k=0}^{j-1} \bar{d}(k) 2^{n+1-k} + \sum_{k=j+1}^{n} \bar{d}(k) 2^{n+1-k} \right)$$

$= 2^{n+1-j}\bar{d}(j)$, where both equalities are implied by Equation 2.

When j is the largest index with $\bar{d}(j) > 0$, the above formula gives $v(j) = d(j)$, so any node on the deepest level of T is a leaf. Thus, T is a feasible binary tree with n leaves. The construction of T clearly takes only polynomial time. □

Lemma 4 reduces max-GHT to the search for a minimum cost assignment d satisfying Equation 2. We further simplify this task by relaxing the Equation to an Inequation. The proof of the following lemma is deferred to the full paper.

Lemma 5. *Let $d : \{1, \ldots, n\} \to \{0, \ldots, n\}$ be a function satisfying*

$$\sum_{k=0}^{n} \bar{d}(k) 2^{n+1-k} \leq 2^{n+1} \ . \tag{3}$$

Then there is a feasible level assignment function d' with $d'(x) \leq d(x)$ for $1 \leq x \leq n$. Furthermore, d' can be determined from d in polynomial time.

Theorem 4. *The version of max-GHT where all cost functions are monotonic can be solved in polynomial time.*

Proof. We have reduced max-GHT to the problem of determining a minimum cost assignment d satisfying Inequation 3. This problem can be solved using binary search to determine the minimum cost of any feasible d. The search is performed over the set of all n^2 function values of $f_1, \ldots, f_n$, because the cost of any solution for our problem instance is one of those function values.

In each iteration of binary search, guess a value c for $\text{cost}(d)$. Recall that we have assumed the cost functions to be monotonic. So, for $i = 1, \ldots, n$, there is some x_i such that $f_i(x) \leq c$ for $x \leq x_i$ and $f_i(x) > c$ otherwise. Assign $d(i) = x_i$ for $i = 1, \ldots, n$ and check Inequation 3. If it is satisfied, then there is a solution tree to our problem instance having cost equal or less than c, which is computable in polynomial time, due to Lemma 5 and 4. Conversely, if Inequation 3 is not satisfied, then there is no solution tree to the instance having cost c or less. This is because the level function d' of such a tree could be obtained from d by decreasing the function values of a certain set of indices, and changing d in that manner can only further increase the left side of Inequation 3. □

References

[AH08] M. Adler and B. Heeringa. Approximating optimal binary decision trees. In *Proc. APPROX-RANDOM '08*, volume 5171 of *LNCS*, pages 1–9, 2008.

[Bae10] M. B. Baer. Alphabetic coding with exponential costs. *Inf. Process. Lett.*, 110(4):139–142, 2010.

[CDKL04] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. Laber. Searching in random partially ordered sets. *Theor. Comput. Sci.*, 321(1):41–57, 2004.

[CJLM10] F. Cicalese, T. Jacobs, E. Laber, and M. Molinaro. On the complexity of searching in trees: Average-case minimization. In *Proc. ICALP '10 (to appear)*, 2010.

[CPR+07] V. T. Chakaravarthy, V. Pandit, S. Roy, P. Awasthi, and M. K. Mohania. Decision trees for entity identification: approximation algorithms and hardness results. In *Proc. PODS '07*, pages 53–62, 2007.

[CPRS09] V. T. Chakaravarthy, V. Pandit, S. Roy, and Y. Sabharwal. Approximating decision trees with multiway branches. In *Proc. ICALP '09*, volume 5555 of *LNCS*, pages 210–221, 2009.

[FP87] P. Flajolet and H. Prodinger. Level number sequences for trees. *Discrete Mathematics*, 65(2):149–156, 1987.

[GM59] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Tech.*, 38:933–966, July 1959.

[HKT79] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM J. Appl. Math.*, 37(2):246–256, 1979.

[HT71] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM J. Appl. Math.*, 21(4):514–532, 1971.

[Hu73] T. C. Hu. A new proof of the T-C algorithm. *SIAM J. Appl. Math.*, 25(1):83–94, 1973.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[Kar72] R. M. Karp. *Reducibility among combinatorial problems. In R. E. Miller and T. W. Thatcher (eds.),* Complexity of Computer Computations. Plenum Press, New York, 1972.

[KM93] M. M. Klawe and B. Mumey. Upper and lower bounds on constructing alphabetic binary trees. In *Proc. SODA '93*, pages 185–193, 1993.

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.

[MOW08] S. Mozes, K. Onak, and O. Weimann. Finding an optimal tree searching strategy in linear time. In *Proc. SODA '08*, pages 1096–1105, 2008.