On the Huffman and Alphabetic Tree Problem with General Cost Functions

Hiroshi Fujiwara* Tobias Jacobs†

Abstract

We address generalized versions of the Huffman and Alphabetic Tree Problem where the cost caused by each individual leaf i, instead of being linear, depends on its depth in the tree by an arbitrary function. The objective is to minimize either the total cost or the maximum cost among all leaves. We review and extend the known results in this direction and devise a number of new algorithms and hardness proofs.

It turns out that the Dynamic Programming approach for the Alphabetic Tree Problem can be extended to arbitrary cost functions, resulting in a time $O(n^4)$ optimal algorithm using space $O(n^3)$. We identify classes of cost functions where the well-known trick to reduce the runtime by a factor of n via a "monotonicity" property can be applied.

For the generalized Huffman Tree Problem we show that even the k-ary version can be solved by a generalized version of the Coin Collector Algorithm of Larmore and Hirschberg [LH90] when the cost functions are nondecreasing and convex. Furthermore, we give an $O(n^2 \log n)$ algorithm for the worst case minimization variants of both the Huffman and Alphabetic Tree Problem with nondecreasing cost functions.

Investigating the limits of computational tractability, we show 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. The alphabetic version becomes NP-hard when the leaf costs are interdependent.

1 Introduction

Computing minimum cost binary trees is a classic 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 the leaves 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. The two most important applications of optimal (alphabetic or non-alphabetic) trees

*This work was supported by KAKENHI (19700015, 23700014, and 23500014).

†This work was supported by a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD).

[0]A preliminary version of this paper has appeared in the proceedings of the 18th Annual European Symposium on Algorithms, 2010. The final publication is available at www.springerlink.com.

are coding and the construction of search indices. In the context of coding, each ℓ_i represents a letter from the source alphabet, and from a binary tree one obtains a binary encoding by labeling the left and right outgoing edge of each internal tree node with a 0 and 1, respectively, so the path from the tree root to ℓ_i becomes associated with a unique bit string. As one typically wants to minimize the expected length of encoded texts, it is natural to give each leaf a weight proportional to its expected frequency and solve the corresponding Huffman or Alphabetic Tree Problem. In the second main application, the construction of search indices, the optimization criterion is less obvious. It might be reasonable to minimize the expected search depth here as well, but keeping the longest possible search path as short as possible is another reasonable goal. A tradeoff between those two extremes would be to penalize outliers superlinearly, e.g. by using a quadratic cost function. This cost function could also be combined with a bound on the maximum search depth any leaf is allowed to have. Another scenario with nonlinear cost functions occurs when the top part of the search index is stored in main memory and the bottom part is stored on a disk.

Motivated by those kind of applications, we focus on a model where 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 $g : \{1, \ldots, n\} \to \text{leaves}(T)$ such that $\sum_{i=1}^n f_i(\text{depth}(g(i), T))$ is minimized, where $\text{depth}(g(i), T)$ is the distance (number of arcs) between the root of T and the leaf $g(i)$.
>
> **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, and we also consider the Δ-ary tree versions of these problems, denoted e.g. **Δ-GAT**.

The runtime of any reasonable algorithm for these problems clearly depends on the time required to evaluate the functions f_i. In this article we assume that evaluating the cost functions is done by an oracle in constant time. It is however not hard to recalculate the runtime of the algorithms taking non-constant evaluation time into account.

Throughout the paper we assume that n is the number of leaves, rather than the size of the input. Our aim is to compare the runtime over various settings, including the classical Huffman and Alphabetic Tree Problems, based on the same criterion. We remark that encoding the n cost functions might require $\Theta(n^2)$ space in the extreme case where the input is simply given by all values of f_i. In this case the runtime of a $t(n)$ time algorithm with regard to input size m is $t(m^{1/2})$. However, we believe that in most applications the cost functions are determined by a constant number of parameters and can therefore be encoded using only constant space.

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. An alternative but similar time $O(n \log n)$ algorithm has been given by Garsia and Wachs [GW77] (see [KLR97] for an instructive correctness proof of both algorithms).

Up to today it is unknown whether the Alphabetic Tree Problem can be solved in time $o(n \log n)$. For certain classes of algorithms $\Theta(n \log n)$ was shown to be a lower bound for the runtime [KM93]. Linear time algorithms have been given for the case of weights differing only by a constant factor [KM93, Hu73], exponentially separated weights [KM93], and weights from domains sortable in linear time [HLM05].

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 $\text{cost}(T) = \sum_i w_i t^{d_i}$ for any constant $t \geq 1$. 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.

A well studied special case of nonlinear cost functions are the height limited versions of the Huffman and Alphabetic Tree Problem. Here the objective is to compute an optimal tree subject to the constraint that each leaf must be reachable from the tree root in a given number of at most L steps, which can be reinterpreted as the cost being infinite whenever the depth of a leaf becomes longer than L. For the non-alphabetic (Huffman) version, an $O(n^2 \log n)$ time algorithm has been given in [Gar74], which has been improved to $O(nL)$ by Larmore [LH90] using the so called packet merge approach. In the same paper a version for the alphabetic tree problem is also given, but its correctness has been proved in a later article [LP94]. That alphabetic version has a runtime of $O(nL \log n)$, which improves upon the previous $O(n^3 \log n)$, $O(n^2 \log n)$, and $O(n^{3/2} L \log^{1/2} n)$ time methods for the problem that were proposed in [Gar74], [Ita76, Wes76], and [Lar87], respectively. The proof given in [LP94] also shows the correctness of the alphabetic packet merge algorithm in a much more general setting, namely, GAT with nondecreasing and convex cost functions.

Nonlinear cost functions can be regarded as a tradeoff between worst and average case optimization. In a recent paper [BD10], Bose and Douïeb study the relation between the worst and average case search tree problem. An interesting result with respect to the alphabetic tree problem is that any k-ary alphabetic tree can be restructured such that the new tree has height $\log_k n + 1$, and during the restructuring process the level of most one leaf increases by two, all other leaves drop by at most one level, and one quarter of all leaves do not drop at all.

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, JCLM10]. In the even more general *Binary Identification Problem*, the input is a number of subsets $S_1, \ldots, S_m \subset [1, n]$ 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 investigate the generalized Huffman and Alphabetic Tree Problem, including the Δ-ary and worst case minimization variants. We show that in many cases existing algorithms can be modified to solve more general versions, and we devise alternative solution methods. We also present hardness results and give counterexamples demonstrating the limits of applicability of many algorithms. See Table 1 for a summary of the results.

problem	class of cost functions		time complexity
GAT	subtree optimality and monotonicity	$w_i d_i$	$O(n^3)$ [GM59], $O(n^2)$ [Knu71], $O(n \log n)$ [HKT79]
		$w_i t^{d_i}$ $(t \geq 1)$	$O(n \log n)$ [HKT79]
		general	$O(n^2)$ [this paper]
	subtree optimality		$O(n^3)$ [this paper]
	monotonicity	nondecreasing and convex	$O(n^2 \log n)$ [LP94]
		general	$O(n^3)$ [this paper]
	general		$O(n^4)$ [this paper]
GHT	$w_i d_i$		$O(n \log n)$ [Huf52]
	nondecreasing and convex		$O(n^2 \log n)$ [this paper]
	general		NP-hard [this paper]
max-GAT	nondecreasing		$O(n^2 \log n)$ [this paper]
	general		$O(n^4)$ [this paper]
max-GHT	nondecreasing		$O(n^2 \log n)$ [this paper]
	general		NP-hard [this paper]

Table 1: Summary of previous and new results.

We start in Section 2 with the Dynamic Programming approach for the Alphabetic Tree Problem. It turns out that there is an extension to the $O(n^3)$-time algorithm by Gilbert and Moore [GM59] that optimally solves problem GAT, including max-GAT and the Δ-ary versions. This comes at the price of an extra factor of $n \cdot \Delta$ in the runtime and n in the memory requirements. There are two basic methods to reduce the resource requirements by factors of n again, each method being applicable for a certain subclass of cost functions. These classes are overlapping, and in case of the classic linear cost model they lead to the time and space $O(n^2)$ algorithm by Knuth [Knu71].

In Section 3 we turn our attention towards the special class of nondecreasing and convex cost functions. Under this assumption the (binary) Alphabetic Tree Problem is known to be efficiently solved by an algorithm by Larmore and Przytycka [LP94]. We give an extension of the algorithm proposed in [LH90] for the height limited Huffman problem that works for Δ-GHT with nondecreasing and convex costs. Then, in Section 4 we investigate Δ-max-GAT and Δ-max-GHT under the assumption that the cost functions are nondecreasing. Both problems turn out to be solvable in time $O(n^2 \log n)$ using a new binary search approach.

In Section 5 we present a number of hardness results that hold under the $P \neq NP$ assumption. While all versions of GAT are solvable in polynomial time via the Dynamic Programming method from Section 2, both GHT and max-GHT are shown to be inapproximable when the cost functions are unrestricted. In the case of GAT, a further generalization turns out to be computationally intractable, namely, when the leaves on each tree level contribute to the cost via a set function $2^{\{\ell_1,\ldots,\ell_n\}} \to \mathbb{R}$. The hardness is established via reduction from a submodular optimization problem. Section 6 concludes the paper.

Although the maximum depth of any full binary tree with n leaves clearly is $n-1$, we assume for simplicity throughout the paper a maximum depth of n. This assumption does not change the asymptotics of the runtime and space analyses.

2 The Dynamic Programming approach

In this section we discuss Dynamic Programming algorithms for the generalized Alphabetic Tree Problem. Our starting point is a recapitulation of the Gilbert-Moore algorithm for the classic linear cost version. All other algorithms in this section are generalizations of that method, often combined with speed-up techniques. One of those techniques is the exploitation of the well-known *monotonicity* property, and we will see for which problem versions that property holds.

2.1 The Gilbert-Moore algorithm and subtree optimality

The algorithm by Gilbert and Moore uses Dynamic Programming to solve the classic Alphabetic Tree Problem with $f_i(x) = w_i x$ for $i = 1, \ldots, n$. The basic idea is to exploit the fact that any optimal alphabetic tree T for leaves $\ell_1, \ldots, \ell_n$ can be decomposed into the two subtrees T_1 and T_2, where T_1 is an optimal alphabetic tree for $\ell_1, \ldots, \ell_j$ for some $j < n$ and T_2 is optimal for $\ell_{j+1}, \ldots, \ell_n$. It holds that $\text{cost}(T) = \text{cost}(T_1) + \text{cost}(T_2) + \sum_{k=1}^{n} w_k$, because the level of each leaf in T is by exactly one deeper than it is in T_1 or T_2.

Subproblems of the DP algorithm 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 $\ell_l, \ldots, \ell_r$. The value $\tilde{c}(l, r)$ of an optimal solution to that subproblem is calculated recursively as

$$\tilde{c}(l, r) = \min_{l \leq i < r} \left(\sum_{k=l}^{r} w_k + \tilde{c}(l, i) + \tilde{c}(i+1, r) \right),$$

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 terminals into optimal trees for larger subsequences.

The reason why the algorithm is correct is because an optimal tree for $f_1, \ldots, f_n$ is always the combination of optimal trees for the leaf sequences $f_1, \ldots, f_i$ and $f_{i+1}, \ldots, f_n$, for some $i \in \{1, \ldots, n\}$. If T' is the optimal tree for $f_1, \ldots, f_i$, the depth of each terminal in it is by one smaller than it is in T. In order to guarantee that the structure of T' is also the optimal structure as a part of T, it is sufficient that the costs are related via a monotone function. In general, if for each $1 \leq l \leq r \leq n$ there is a monotone function $h_{lr} : \mathbb{R} \to \mathbb{R}$ with

$$\sum_{l \leq i \leq r} f_i(\text{depth}(\ell_i, T) + 1) = h_{lr} \left(\sum_{l \leq i \leq r} f_i(\text{depth}(\ell_i, T)) \right),$$

then the algorithm by Gilbert and Moore is applicable. When the above formula is satisfied, we say that the cost function is *subtree optimal*. In the worst-case minimization problem variant the sums in the formula must be replaced with max.

Note that in the linear cost model, h_{lr} is calculated by adding the total weight of $\ell_l, \ldots, \ell_r$ to the cost. In the exponential cost model investigated in [HKT79] where $\text{cost}(T) = \sum_i w_i t^{d_i}$ for a constant t, we have $h_{lr} : x \mapsto tx$ for each l, r, and also the maximization variant $\text{cost}(T) = \max_i w_i t^{d_i}$ addressed in that paper satisfies subtree optimality with the same function h_{lr}.

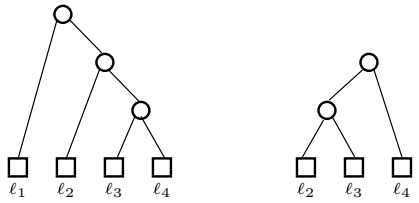


Figure 1: Counterexample to subtree optimality. Assume that the weights of $\ell_1, \ell_2, \ell_3, \ell_4$ are $4, 1, 1, 2$ and the linear cost function is used, with the exception that the cost of ℓ_2 in depth greater than 2 makes a jump to 10. Then the subtree containing ℓ_2, ℓ_3, ℓ_4 of the optimal tree on the left hand side differs from the optimal tree for that subsequence shown on the right hand side.

The general recursive formula for the calculation of $\tilde{c}(l, r)$ becomes

$$\tilde{c}(l, r) = \min_{l \le i < r} h_{lr}(\tilde{c}(l, i) + \tilde{c}(i + 1, r)),$$

and in the worst case problem variant one has to use

$$\tilde{c}(l, r) = \min_{l \le i < r} h_{lr}(\max\{\tilde{c}(l, i), \tilde{c}(i + 1, r)\}).$$

2.2 DP algorithm for GAT

The simple counterexample in Figure 1 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.

In order to handle an instance that does not satisfy subtree optimality, we introduce the idea of an *offset* as follows, which has not appeared in previous work. 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

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

The optimality of T implies that the structure of say T_1 must be such that the term $\sum_{1 \le j \le i} f_j(\text{depth}(\ell_j, T_1) + 1)$ 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_j's, and so on.

We make this offset an additional parameter of 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 $\sum_{i=l}^{r} f_i(\text{depth}(\ell_i, T') + k)$. 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

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

For solving max-GAT instead of GAT, all we have to do is replace the sum with a maximization symbol, obtaining

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

In both cases 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 $O(n^4)$.

Equation 1 shows that for solving some subproblem whose third parameter is k, it is sufficient to know the solution values of all subproblems with offset $k+1$. Therefore we can first solve all subproblems with offset n, then continue with offset $n-1$, until we reach offset 0. After having computed all solutions with offset k, solutions with offset $k-1$ can be discarded. This way, we never store solution values to more than $\Theta(n^2)$ different subproblems at once, i.e., we can compute the cost of the optimal solution to the initial problem in quadratic space. However, for determining the alphabetic tree achieving that optimal cost, we need to explicitly store the corresponding tree together with each solution value, which adds another factor of n to the space requirements. Alternatively, the optimal tree can be reconstructed from the complete $\Theta(n^3)$ size Dynamic Programming table. In both cases the memory requirements are $\Theta(n^3)$.

2.3 Monotonicity

The property of *monotonicity* was proven by Knuth [Knu71] to hold for the classic Alphabetic Tree Problem. It roughly states that the root of an optimal alphabetic tree can only move left when the terminal sequence 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 monotonicity 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.

Monotonicity yields an improved algorithm for the GAT problem. This algorithm computes optimal solutions to the subproblems in the following order: for $k = 0, \ldots, n$, for $a = 0, \ldots, n-1$, 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 k and a. 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. The monotonicity property implies that $i[j, j+a-1, k] \leq i[j, j+a, k] \leq i[j+1, j+a, k]$. 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 $i[j, j+a, k]$, $j = 1, \ldots, n-a$, 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 subtree optimality and monotonicity hold, 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 2.

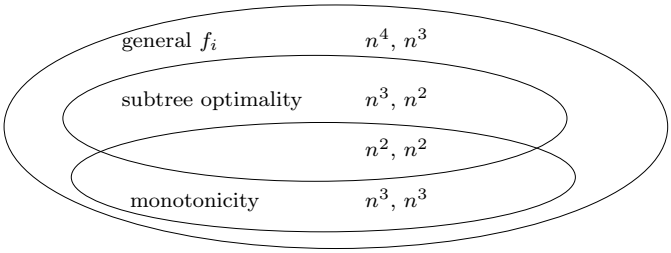
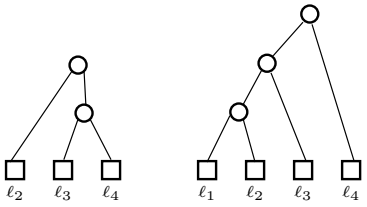


Figure 2: Runtime and space requirements of the DP approach for GAT. Whenever the space requirements are n^3 they can be reduced to n^2 at the price of only determining the optimal solution's cost.



	ℓ_1	ℓ_2	ℓ_3	ℓ_4
depth 1	0	5	1	1
depth 2	0	10	2	2
depth 3	0	10	3	3

Figure 3: A GAT instance not satisfying the monotonicity property. The depicted trees are optimal for their respective terminal sequences.

2.4 Cost functions satisfying monotonicity

In the preceding subsection we have seen that there are two different subclasses of cost functions that allow to speed up the Dynamic Programming algorithm. In Figure 1 it is demonstrated that not all cost functions are subtree optimal. Note that all cost functions in that example are nondecreasing and convex, which means that the monotonicity property holds for that instance (see discussion below). Cost functions satisfying monotonicity also form a true subclass of all cost functions, as shown in Figure 2.

Classes of cost functions satisfying subtree optimality have been given in Section 2.1. We now discuss cost functions satisfying the monotonicity property. For max-GAT, monotonicity is guaranteed by any nondecreasing cost function, which is stated in Theorem 2. Concerning GAT, the appropriate function class is more restricted, as demonstrated in Figure 3. In addition to being nondecreasing, we also need convexity here, i.e. $0 \leq f_i(x) - f_i(x-1) \leq f_i(x+1) - f_i(x)$ for each i and $x \geq 1$. The monotonicity property for GAT is due to a slightly more general result by Larmore and Przytycka [LP94].

Theorem 1 (Proved in [LP94], Section 8)**.** *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 the monotonicity property.*

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 the monotonicity property.*

The proof of Theorem 2 is based on a simple observation about the cost $\tilde{c}(l, r, k)$ of the optimal solution to subproblem (l, r, k), whose correctness 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.

Observation 1. *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). Assume that for some $j > i$ there exists an optimal solution T' 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 $(l-1, r, k)$ subject to the constraint that the root must be 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. $\text{cost}(T'') = \tilde{c}(l-1, i, k+1) \geq \tilde{c}(i+1, r, k+1)$. Multiple application of Observation 1 gives

$$\text{cost}(T') \geq \tilde{c}(l-1, j, k+1) \geq \tilde{c}(l-1, i, k+1) = \text{cost}(T'').$$

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

2.5 Multi-ary GAT

We finally examine to which extent the results from this section can be generalized to the Δ-ary Alphabetic Tree Problem.

Assume that for some offset value k we have already computed the solutions to all subproblems $(l', r', k+1)$. For solving some subproblem (l, r, k), we have to enumerate all possibilities to divide the $\ell_l, \ldots, \ell_r$ into Δ subsequences. This enumeration itself can be efficiently implemented by Dynamic Programming. For even Δ, the optimal division into Δ subsequences is equal to the best division into 2 subsequences, each of which is then recursively divided into $\Delta/2$ subsequences. For odd Δ, we first enumerate all possibilities for where the first subsequence ends. To integrate this strategy into the algorithm, we introduce a fourth parameter to the subproblem description: subproblem (l, r, k, δ) now asks for a Δ-ary alphabetic tree for $\ell_l, \ldots, \ell_r$ that is optimal under offset k and the restriction that the root has at most δ children. The recursive formula for the cost of the optimal solution now is as follows:

$$\tilde{c}(l, r, k, \delta) = \begin{cases} f_l(k) & \text{if } r = l \\ \tilde{c}(l, r, k+1, \Delta) & \delta = 1 \\ \min_{i=l}^{r-1}\{\tilde{c}(l, i, k, 1) + \tilde{c}(i+1, r, k, \delta-1)\} & \delta \neq 1 \text{ is odd} \\ \min_{i=l}^{r-1}\{\tilde{c}(l, i, k, \delta/2) + \tilde{c}(i+1, r, k, \delta/2)\} & \delta \text{ is even.} \end{cases} \tag{2}$$

The original problem is described by configuration $(1, n, 0, \Delta)$. At most $2\log\Delta$ different values of δ do actually appear during the computation, namely, a subset of $\{\Delta, \Delta-1, \lfloor\Delta/2\rfloor, \lfloor\Delta/2\rfloor - 1, \lfloor\Delta/4\rfloor, \ldots, 1\}$. For example, for $\Delta = 14$ the appearing values of δ are $14, 7, 6, 3, 2, 1$.

Furthermore, the extra space requirements are within a constant factor if we do the computations in the right order and discard all values not needed anymore. The order is as follows: for decreasing k, for increasing δ (but restricted to the $O(2\log\Delta)$ relevant values), for all l, r compute $\tilde{c}(l, r, k, \delta)$. Whenever we have computed some value of $\tilde{c}(l, r, k, \delta)$ for $\delta \notin \{1, \Delta\}$, we are only going to need that value for the computation of either $\tilde{c}(\cdot, \cdot, k, 2\delta)$ or $\tilde{c}(\cdot, \cdot, k, \delta+1)$, and after the latter values have been computed we can discard the $\tilde{c}(l, r, k, \delta)$.

We obtain overall runtime of $O(n^4 \log\Delta)$ and the space requirements remain $O(n^3)$ (or $O(n^2)$ if we are only interested in the cost of the optimal solution).

If subtree optimality holds, the resource requirements are by a factor of n lower like in the binary case, and the definition of subtree optimality is independent from Δ. In contrast, even the multi-ary version of the classic alphabetic tree problem with linear costs does not satisfy the monotonicity property, counterexamples can be found in [Got81].

For solving the Δ-ary version of max-GAT, we just have to replace all cost sums by maximization symbols in Formula 2. Furthermore, for the class of nondecreasing cost functions the proof of Theorem 2 directly generalizes, i.e. monotonicity is satisfied and thus the Dynamic Programming algorithm takes time $O(n^3 \log \Delta)$ and space $O(n^3)$ (or space $O(n^2)$ for the minimum cost value).

3 Packet-merge algorithms

As mentioned in the previous section, it has been shown by Larmore and Przytycka in [LP94] that problem GAT satisfies the monotonicity property when the cost functions are nondecreasing and convex. Therefore, the Dynamic Programming approach can be made to run in time $O(n^3)$. However, instead of proposing Dynamic Programming, the authors use monotonicity as an argument in the correctness proof of their so-called Packet-Merge algorithm. The latter algorithm runs in time $O(n^2 \log n)$ and therefore outperforms the Dynamic Programming method. Although the algorithm itself is very simple, it turns out that its correctness is very difficult to prove. Also, as the monotonicity property only holds for the case of binary trees, it is not likely that there is a version for constructing multi-ary alphabetic trees.

3.1 Multi-ary Huffman trees

There is a simpler version of the Packet-Merge algorithm for the construction of height-limited Huffman trees that appeared in [LH90]. This version also is much easier to analyze. In the following we show a two-fold generalization of it, namely, to multi-ary Huffman trees with arbitrary nondecreasing and convex cost functions.

We assume here that $f_i(0) = 0$ for each cost function f_i. This assumption is without loss of generality, because otherwise we can obtain an equivalent problem instance $g_1, \ldots, g_n$ by defining $g_i(j) = f_i(j) - f_i(0)$. The transformation preserves nondecreasingness and convexity of the cost functions, and any optimal solution for the transformed instance is also optimal for the original one.

Furthermore, we assume that $n \geq 2$ and that $n - 1$ is a multiple of $\Delta - 1$. To see why the latter assumption also comes without loss of generality, let I be a problem instance not satisfying that assumption and let I' be I enhanced by the minimum number x of nodes such that $n + x - 1$ becomes a multiple of $\Delta - 1$. The cost function of the additional nodes is the constant zero function. Any solution to I' can be transformed into a solution to I having the same cost simply by removing those x terminals. Conversely, for any full Δ-ary tree with m leaves it holds that $m - 1$ is a multiple of $\Delta - 1$, and therefore no solution to I can be a full Δ-ary tree, i.e., there always is some internal node v having less than Δ children. One can append a subtree containing the x extra nodes to v, obtaining a solution to I' having the same cost.

We are given the terminals $\ell_1, \ldots, \ell_n$, and we want to construct a Δ-ary Huffman tree for them. For positive integers $d_1, \ldots, d_n$, Kraft's Inequality [Kra49] tells us that there is a tree with each ℓ_i having depth d_i or less if and only if $\sum_{i=1}^{n} \Delta^{-d_i} \leq 1$. This sufficient and necessary condition is equivalent to

$$\sum_{i=1}^{n} (1 - \Delta^{-d_i}) \geq n - 1.$$

In the formula, each leaf having depth d_i contributes an amount of $1 - \Delta^{-d_i}$ to the sum. We distribute that contribution among the levels from 1 to d_i, such that ℓ_i contributes for each tree

level $1 \leq j \leq d_i$ an amount of $\frac{\Delta-1}{\Delta^j}$; note that $\sum_{j=1}^{d_i} \frac{\Delta-1}{\Delta^j} = 1 - \Delta^{-d_i}$. This contribution is what will be defined as the *width* of the corresponding packet in the following.

We introduce a *packet* p_{ij} for each terminal ℓ_i and each level j, i.e., the total number of packets is n^2. The *width* of a packet p_{ij} only depends on its level j, as mentioned above it is defined as $w_j = \frac{\Delta-1}{\Delta^j}$. Furthermore, each packet has a *cost* c_{ij}, which is defined as the cost difference between terminal i at level $j-1$ and the same terminal at level j, so $c_{ij} = f_i(j) - f_i(j-1)$. Note that there are no packets defined for the root level 0, and for any i and j, the costs of packets $p_{i1}, \ldots, p_{ij}$ sum up to $f_i(j)$.

Any Huffman tree can be represented by a collection of packets as follows: For each $i = 1, \ldots, n$ choose packets $p_{i1}, \ldots, p_{id_i}$, where d_i is the depth of leaf ℓ_i in the Huffman tree. By the above considerations of Kraft's Inequality, the total width of the packet collection is at least $n-1$. The above assumption of $n \geq 2$ has the effect that any leaf ℓ_i in any Huffman tree is represented by at least the packet p_i1. By the definition of the packets' costs, the total cost of the packet collection equals the cost of the Huffman tree.

As an example, consider an instance of the 3-ary alphabetic tree problem ($\Delta = 3$) with 5 weighted terminals ($n = 5$), where the cost of terminal ℓ_i being in level j is $w_i \cdot j^2$. There are $5^2 = 25$ packets, one for each terminal $\ell_1, \ldots, \ell_5$ and each level $1, \ldots, 5$. For instance, packet p_{i2} corresponds to level 2, and its width is $\frac{2}{3^2} = \frac{2}{9}$. Its cost is the difference between the level 1 and the level 2 cost of ℓ_i , i.e., $w_i \cdot 4 - w_i \cdot 1 = 3w_i$. Let T be the 3-ary tree where ℓ_1, ℓ_2 are on level 1 and ℓ_3, ℓ_4, ℓ_5 are on level 2. This tree corresponds to the packet subset $\{p_{11}, p_{21}, p_{31}, p_{32}, p_{41}, p_{42}, p_{51}, p_{52}\}$. The three level 2 packets have a total width of $\frac{2}{3}$, and there are five level 1 packets each having a width of $\frac{2}{3}$. The total width of T is $\frac{2}{3} + 5 \cdot \frac{2}{3} = 4 = n - 1$.

In general, the monotonicity of the cost function effectuates that each packet has a positive cost, and the convexity implies that the cost of p_{ij} is nondecreasing in j, that is, $c_{ij} \geq c_{ij'}$ for $j > j'$. Any Huffman tree can be represented as a packet collection having total width at least $n-1$, but not every such collection represents a tree. In order for a packet collection to represent a tree, one needs in addition that for each i there is a j such that all packets $p_{i1}, \ldots, p_{ij}$ are members of the collection and no packet p_{ik} with $k > j$ belongs to it. We will see shortly that the monotonicity of the packet cost guarantees that the latter constraint is automatically satisfied some cost-optimal set of packets.

We define a collection of packets to be optimal if it has minimum total cost under the constraint that its total width is at least $n-1$.

Lemma 1. *Let U be the set of packets obtained from an instance of the Huffman Tree Problem with convex cost functions by the procedure described above. From any optimal collection M of packets from U an optimal Huffman Tree T can be computed in time $O(n^2)$. Furthermore, the cost of T equals the cost of M.*

Proof. Let M be an optimal collection of packets as described in the lemma. Let i and j be such that packet p_{ij} is not a member of M, but some p_{ik} ($k > j$) is. If there are no such indices, the packet collection already represents a Huffman tree having the same cost. Otherwise, p_{ik} can be replaced with p_{ij} in M. By this operation, the total width of M increases, and due to the convexity of the cost functions the total cost of M cannot increase. As the cost cannot decrease as well due to the optimality assumption, it must remain the same. After a finite number of applications of this operation, M represents a Huffman tree having the same cost as the original packet collection M.

This Huffman tree must be optimal, because any better Huffman tree T^* could be transformed into a cheaper packet collection: let the terminals $\ell_1, \ldots, \ell_n$ have depth $d_1, \ldots, d_n$, select each packet p_{ij} with $j \leq d_i$. The cost of this collection is equal to the cost of T^*, and the width is above the threshold $n-1$, as shown by the above discussion using Kraft's Inequality.

To see that the transformation of M can be done in time $O(n^2)$, consider the following implementation: For each $i = 1, \ldots, n$, count $y_i := |\{j \mid p_{ij} \in M\}|$ and then replace the subset $\{p_{ij} \in M\}$ with $\{p_{i1}, \ldots, p_{iy_i}\}$. □

Lemma 1 reduces the Huffman problem to the problem of determining an optimal packet collection. Two more structural insights about solutions to the latter problem will lead to an efficient algorithm for it. Here it becomes crucial that $n-1$ is a multiple of $\Delta - 1$.

We remark that the following lemmas do not require the monotonicity of the packet costs that has been the central argument in the proof of the preceding lemma. This means that optimal packet collections can be computed efficiently, as we will see below, even without this monotonicity. However, only monotonous packet costs (or, equivalently, convex cost functions for the Huffman tree problem) guarantee the close relationship between optimal packet collections and optimal Huffman trees shown above.

Lemma 2. *Let U be a set of packets each having a width $w_j = \frac{\Delta-1}{\Delta^j}$ for some $j > 1$ and a cost, and let $n-1$ be a multiple of $\Delta - 1$. There is an optimal collection M of packets from U having the following property:*

If m is the largest index such that M contains some width w_m packet and $m > 1$, then the number k of such packets in M is $k = a \cdot \Delta$ for some integer a.

Proof. Let M be an optimal collection of packets from U such that among all optimal packet collections it has minimum cardinality.

We first show that the total width of M must be *exactly* $n-1$. As $n-1$ is a multiple of $\Delta - 1$, $n-1$ also is a multiple of any packet width $w_j = \frac{\Delta-1}{\Delta^j}$. In particular, $n-1$ is a multiple of w_m. Furthermore, the width of any packet wider than w_m in M is a multiple of w_m, and therefore the total width of M must be a multiple of w_m. Thus, the difference between $n-1$ and the total width of M must be either zero or at least w_m. In the latter case we can drop some width w_m packet from M and obtain a feasible packet collection at least as cheap as M and of smaller cardinality, which contradicts the minimality assumption.

Let M' be the packet collection obtained by removing all width w_m packets from M. By the structure of the packet widths, the total width of M' is a multiple of w_{m-1}. As also $n-1$, the total width of M, is a multiple of w_{m-1}, the width difference between M and M' must be multiple of w_{m-1} as well. This width difference is exactly the total width of all width w_m packets in M. As k is the number of these packets, it holds that $k\frac{\Delta-1}{\Delta^m} = a\frac{\Delta-1}{\Delta^{m-1}}$ for some integer a, which implies that $k = \Delta \cdot a$. □

Lemma 3. *Let U be a set of packets each having a width $w_j = \frac{\Delta-1}{\Delta^j}$ for some $j > 1$ and a cost, and let $n-1$ be a multiple of $\Delta - 1$. Let $m > 1$ be the largest index such that there are at least Δ packets of width w_m in U, and let P be the group consisting of the Δ width w_m packets having the smallest cost (breaking ties arbitrarily).*

There is an optimal collection of packets from U that either contains no or all packets from P.

Proof. Let M be an optimal collection of packets from U. Due to Lemma 2, M does not contain any packet of width smaller than w_m. From Lemma 2 also follows that M either contains zero or at least Δ width w_m packets. In the further case, we are done. Otherwise we can remove the Δ minimum cost packets of width w_m from M and insert the packets from P instead, which cannot increase the cost of M due to the definition of P. We obtain an optimal packet collection having the desired property. □

In the first round, the Packet-Merge algorithm greedily selects Δ width w_n packets having smallest cost and groups them into a meta packet P, breaking ties arbitrarily. The lemma

just proven implies that the solution space can be restricted to packet collections that either completely contain P or contain no single member of P. P has a total width of w_{n-1}, so from now on it can as well be treated as a single width w_{n-1} packet whose cost is $\sum_{p_{in} \in P} c_{in}$.

This grouping procedure is repeated with the next Δ minimum cost width w_n packets, until we have less than Δ of such packets left. Those leftover packets can be dropped due to Lemma 3, which marks the end of the first round. We now have an instance of the packet selection problem where the smallest packet width is w_{n-1}. We continue with the next round, and after $n-1$ rounds we arrive at an instance which consists only of width $w_1 = \frac{\Delta-1}{\Delta}$ packets. The $\frac{(n-1)\Delta}{\Delta-1}$ of them having the smallest cost give an optimal solution; note that the latter expression is an integer due to the assumption that $\Delta - 1$ is a multiple of $n-1$.

For the runtime and space analysis, we first reason about the maximum number of meta packets that are generated during one round, when the width w_m packets are eliminated for some $m \in \{n, n-1, \ldots, 2\}$. We claim that this number is at most n. By induction we can assume that at the beginning of the round the total number of width w_m packets is at most $2n$. As each new width w_{m-1} meta packet consists of $\Delta \geq 2$ such packets, the claim follows.

In the round where the width w_m packets are eliminated, assume that we are given the meta packets that have been generated in the previous round. We generate the n additional non-meta width w_m packets from the cost functions, then sort the set of all width w_m packets by cost, and finally generate the width w_{m-1} meta packets by grouping. The runtime is dominated by the sorting time $O(n \log n)$. As there are n rounds, the overall runtime is $O(n^2 \log n)$. The space requirements are $O(n^2)$ if we keep all information about which packet is contained in which meta packet. This is not necessary if we are rather interested in the cost of an optimal solution than in its structure; then we can discard all information about a (meta-) packet once it has become part of a meta packet, and the space requirements are only linear.

4 Worst case optimization

In this section we give an algorithm that computes optimal solutions to problem instances of both max-GHT and max-GAT when the cost functions are nondecreasing.

The approach is based on the simple observation that there are only n^2 possible values for the worst case cost of a tree, namely, $f_i(j)$ for each $1 \leq i, j \leq n$. In a preprocessing step, our algorithm sorts those n^2 cost values. Then it performs binary search to determine the smallest possible cost achievable by a feasible solution.

The feasibility test asks whether there is a solution tree whose cost is not larger than some given rational number c. For each terminal ℓ_i, the upper bound c induces an upper bound $d_i := \max\{j \mid f_i(j) \leq c\}$ on the tree depth of the terminal, which can be found in time $n \log n$, again using binary search. After having determined $d_1, \ldots, d_n$ we need to check whether there is a solution where these depth bounds are satisfied. The method of doing this depends on whether we want to solve max-GHT or max-GAT, but in both cases it takes linear time and space, as shown below. The overall runtime for determining the optimal solution cost is therefore $O(n^2 \log n)$, while the space requirements are linear. For constructing the corresponding solution, we need a method to construct a tree from the optimal depth sequence $d_1, \ldots, d_n$. This is also accomplishable in linear space, and the runtime is in $O(n^2 \log n)$, as we show below.

The approach can be straightforwardly generalized to solve the Δ-ary problem versions within the same asymptotic runtime and space bounds.

4.1 Huffman trees

In the case of max-GHT, we simply apply Kraft's Inequality as the feasibility test, i.e. we check whether

$$\sum_{i=1}^{n} 2^{-d_i} \leq 1.$$

Once the optimal solution cost is found, one method to construct the solution tree is to sort the terminals by nondecreasing level and then apply the method given below for constructing an alphabetic tree from the depth sequence. This works out because for any tree T there is a tree T' where the leaf levels from left to right are nonincreasing, and each leaf has the same level in T and T'.

4.2 Alphabetic trees

There is an equivalent formula to Kraft's Inequality, discovered by Yeung [Yeu91], for determining whether there is an alphabetic tree with the terminals in depth $d_1, \ldots, d_n$ or less. Given $d_1, \ldots, d_n$, define

$$h_0 = 0 \text{ and } h_i = (\lceil h_{i-1} \cdot 2^{d_i} \rceil + 1) \cdot 2^{-d_i}, i = 1, \ldots, n.$$

There is an alphabetic tree (possibly including internal nodes having only one child) where the terminals have depth $d_1, \ldots, d_n$ if and only if $h_n \leq 1$.

In the following we give a proof of Yeung's formula that induces an efficient method to construct a corresponding alphabetic tree whenever the condition for its existence is satisfied. Note that when we remove the $\lceil \cdot \rceil$-operator from the above formula we obtain exactly the left side Kraft's Inequality. This implies the *only if* part, because h_n is never smaller than this left side.

For proving the *if*-part, we associate each node position in an alphabetic tree with a rational number from the semi-open interval $(0, 1]$ and show how to construct an alphabetic tree where each terminal ℓ_i is at a position associated with h_i. Let v be a tree node at level d, and let $b_1, \ldots, b_d \in \{0, 1\}^d$ be a bit vector such that for $j = 1, \ldots, d$ the component b_j is 1 if and only if the level j node on the path from the tree root to v is the left child of its parent. Note that this definition deviates from the standard notion of the left child being associated with 0. The rational number $q(v) \in (0, 1]$ associated with v is defined as

$$q(v) := 1 - (0.b_1 \cdots b_d)_2 = 1 - \sum_{j=1}^{d} b_j 2^{-j}.$$

In particular, the tree root and each other node on the rightmost path of a full binary tree is associated with 1. See Figure 4 for an example.

Assuming that Yeung's formula is satisfied, we show how to construct an alphabetic tree with $q(\ell_i) = h_i$ for $i = 1, \ldots, n$ from left to right, with the terminals being at level $d_1, \ldots, d_n$. First, create a left path (i.e., a path where each node is the left child of its parent) from the tree root to ℓ_1 in level d_1. We obtain that $q(\ell_1) = 1 - \sum_{j=1}^{d_1} 2^{-j} = 2^{-d_1} = h_1$.

For $i = 2, \ldots, n$, assume that the partial tree for $\ell_1, \ldots, \ell_{i-1}$ has already been constructed, in particular $q(\ell_{i-1}) = h_{i-1}$ and ℓ_{i-1} is in level d_{i-1}. Let also be $b_1, \ldots, b_{d_{i-1}}$ be the bit vector corresponding to the path from the root to ℓ_{i-1}. That vector must contain at least one 1, because otherwise we would have $q(\ell_{i-1}) = h_{i-1} = 1$ and as $h_i > h_{i-1}$ the inequality $h_n \leq 1$ could not be satisfied. Let j be the largest index with $b_j = 1$, and let v_j be the corresponding tree node. We know that node v_j is the left child of its parent v_{j-1} (observe that this parent

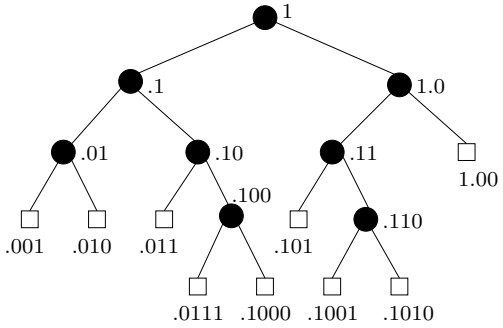


Figure 4: A binary tree and the binary representation of $q(v)$ at each node v. Note that $q(v)$ remains constant as long as we traverse a right path, and it becomes by 2^{-d} smaller whenever we traverse an edge leading to a left child at level d.

could be the tree root), and from the definition of q we have that $q(v_j) = q(\ell_{i-1})$. We now distinguish between two cases.

Case 1: $d_i \geq j$, i.e., the level of ℓ_i is deeper than the level of v_{j-1}. This implies that $h_{i-1} = q(\ell_{i-1}) = q(v_j) = 1 - \sum_{j'=1}^{j} b_{j'} 2^{-j'}$ is a multiple of 2^{-d_i}, and we have $h_i = h_{i-1} + 2^{-d_i}$. We create a new left path of length $j - d_i$ whose last node is ℓ_i, and this path is appended as the right subtree to v_{j-1}. Now the characteristic bit vector of the overall path to ℓ_i is $b_1, \ldots, b_{j-1}, 0, 1, \ldots, 1$, and we obtain that $q(\ell_i) = q(v_j) + 2^j - \sum_{j'=j+1}^{d_i} 2^{-j'} = h_{i-1} + 2^{-d_i} = h_i$. As ℓ_{i-1} is in the left subtree under v_{j-1} and ℓ_i is in the right subtree under v_{j-1}, our placement also satisfies the alphabetic restriction.

Case 2: $d_i < j$, i.e., ℓ_i is on a tree level less deep than v_j. On the path from the tree root to v_j, consider the node v_{d_i} that has level d_i. The characteristic bit vector of the path from the root to v_{d_i} is $b_1, \ldots, b_{d_i}$, so $q(v_{d_i}) = 1 - \sum_{j'=1}^{d_i} b_{j'} 2^{-j'}$ is a multiple of 2^{-d_i}, and at the same time $q(v_{d_i})$ is by no more than 2^{-d_i} larger than $q(v_j) = q(\ell_{i-1}) = h_{i-1}$. Therefore $\lceil h_{i-1} 2^{d_i} \rceil = q(v_{d_i}) 2^{d_i}$, which implies that $q(v_{d_i}) = \lceil h_{i-1} 2^{d_i} \rceil 2^{-d_i}$. We now do the same construction as in Case 1, but here we place ℓ_i in relation to v_{d_i} instead of ℓ_{i-1}, resulting in $q(\ell_i) = q(v_{d_i}) + 2^{-d_i} = \lceil h_{i-1} 2^{d_i} \rceil 2^{-d_i} + 2^{-d_i} = h_i$. The resulting placement of ℓ_i in the tree is right of v_{d_i}, and as the latter node is an ancestor of ℓ_{i-1} the alphabetic restriction is satisfied between ℓ_{i-1} and ℓ_i.

The above argumentation shows the correctness of Yeung's inequality, and it describes a way to construct a corresponding tree T' where the terminals are exactly in depth $d_1, \ldots, d_n$. This tree T' possibly contains unary nodes with only one child. We obtain the full binary tree T from T' by deleting those unary nodes, it is trivial to see that this only decreases the depth of any terminal.

In order to make the construction efficient, we need to solve the issue that T' might be large due to a great number of unary nodes. The idea is to represent T' by a full binary edge-weighted tree, where a weight k edge represents a path of length k. The construction of T' can be completely done within this representation. The transition from T' to T now becomes particularly easy, because we just have to remove the edge weights.

The space requirement of the representation of T' is $O(n \log n)$ as any full binary tree has $O(n)$ edges and each of them represents a path of length at most n. However, we can immediately forget the weight of any edge as soon as it is not contained by the path leading to the actual ℓ_i. The latter path has a (weighted) length of at most n, so the whole construction can be done within linear space.

5 Hardness results

In the previous section we have seen that both the Huffman and Alphabetic Tree Problems admit polynomial time algorithms for rather general classes of cost functions. This section is devoted to the limits of computational tractability regarding these problems. It turns out that the Huffman Problem becomes NP-hard if we drop that constraint that the cost function must be monotonic. This holds for both GHT and max-GHT. As for GAT and max-GAT, the existence of a polynomial time algorithm for any cost function has been shown in Section 2.2. We prove in this section that a further generalization makes the problem NP-hard.

5.1 Complexity of GHT

The computational complexity of 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 consisting of $2k + 3m$ leaves. We consider the solution tree for instance I to be partitioned into 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 $i = 1, 2, \ldots$ and $q \in \{1, 2, 3\}$. The tree root level 0 is not considered to be in one of the layers. In other words, the infinite sequence of integers $(1, 2, 3, 4, 5, \ldots)$ equals the sequence $(l_1^1, l_2^1, l_3^1, l_1^2, l_2^2, \ldots)$. Only the layers $i = 1, \ldots, m$, each of which corresponds to a 3-element subset, will play a role in our proof.

In instance I there are three different types of cost functions. Each of the $2k + 3m$ cost functions formally defined below is associated with exactly one leaf in instance I. Intuitively, the f-type functions will force the optimal tree to have a path to level $3m - 1$. This path has m "holes", one for each subset in D. There also is an h-type function for each element of C, which is defined such that the respective leaf fits into the holes corresponding to the subsets the element of C appears in. In addition, each hole has room for no more than three h-type leaves. The third category of cost functions, the g-type functions, has the purpose to "fill" the holes not populated by h-type leaves.

Firstly, 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.

Secondly, we introduce $m - k$ functions $g_1, \ldots, g_{m-k}$. These 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 4. *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 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.

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. Similarly, we can assume that v_2^m is the parent of the leaves associated with f_2^m and f_3^m, since we can interchange children between the parent of the leaf associated with f_2^m and that of f_3^m.

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 g-type and h-type functions is exactly the set of leaves being in those subtrees. The root of T_i is at level l_1^i. 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. Therefore, the three h-type cost functions in T_i must be exactly the cost functions corresponding to the elements of D_i.

As the number of subtrees T_i of this kind is k, and each h-type function can only occur in one of them, the corresponding selection of D_i's 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.*

5.2 Set function alphabetic tree problem

We prove NP-hardness of a further extension of GAT. Let $g_1, \ldots, g_n : 2^L \to \mathbb{R}_0^+$ be real-valued set functions on the set $L = \{\ell_1, \ldots, \ell_n\}$ of leaves. Each g_i contributes $g_i(S_i(T))$ to the cost, where $S_i(T)$ denotes the subset of leaves that are located at depth i in the alphabetic tree T. We hereafter assume that the value of a set function is provided as an oracle. The extended problem is described as follows.

> **Set Function Alphabetic Tree Problem, SFAT.** Given a sequence of leaves $\ell_1, \ldots, \ell_n$ and n set functions $g_1, \ldots, g_n : 2^{\{\ell_1, \ldots, \ell_n\}} \to \mathbb{R}_0^+$, the objective of SFAT is to determine a binary tree T whose leaves in left-to-right order are $\ell_1, \ldots, \ell_n$, such that $\sum_{i=1}^n g_i(S_i(T))$ is minimized.

One can easily see that if each of the set functions is modular, that is, its value is the sum of the contributions of each leaf, then the problem is equivalent to GAT. Contrary to the polynomial-time complexity of GAT, SFAT turns out to be NP-hard, even if the set functions

are all submodular. We will show that a special case of the Multiway Partition Problem [ZNI05] is reducible to SFAT.

> **Multiway Partition Problem, MP.** Given a submodular system (V, f), a target set $U \subseteq V$, and an integer k with $2 \leq k \leq |U|$, minimize $\sum_{i=1}^{k} f(V_i)$ subject to $\bigcup_{i=1}^{k} V_i = V$, $V_i \cap V_j = \emptyset$ for $i \neq j$, and $V_i \cap U \neq \emptyset$ for $1 \leq i \leq k$.

In what follows we give a construction of an instance of SFAT from a given instance of MP with $k = |U|$. MP is known to be NP-hard even with $k = |U|$ (cf. [ZNI05]).

Let $m := |V|$, $V =: \{v_1, \ldots, v_m\}$, and $U =: \{u_1, \ldots, u_k\}$. Each element of V will be associated with a leaf in the SFAT instance we are about to construct. For simplicity of notation we do not distinguish between elements of V and the corresponding leaves. The cost functions will be such that each element u_i of U will be forced to be on a certain tree level denoted λ_i. These $|U|$ levels are pairwise distinct. Furthermore, the cost functions will enforce that the elements of $V \setminus U$ are only allowed to be on the levels $\{\lambda_1, \ldots, \lambda_k\}$. Therefore, the distribution of the elements of $V \setminus U$ among these m levels represents a feasible solution to the MP instance. We also need that the alphabetic tree has the same cost as the corresponding partition. This is achieved by defining the cost function of λ_i as

$$g_{\lambda_i}(S) = \begin{cases} f(S \setminus D) & \text{if } u_i \in S, \\ \infty & \text{otherwise,} \end{cases}$$

where D stands for the set of dummy leaves defined below. It is not hard to see that the submodularity of f implies the submodularity of the level cost functions. To avoid that leaves from V appear on other levels than $\lambda_1, \ldots, \lambda_k$, the cost functions g_j with $j \notin \{\lambda_1, \ldots, \lambda_k\}$ are defined to be zero for the empty set and infinity otherwise.

We have already achieved that each alphabetic tree with finite cost represents an MP solution having the same cost. For making the reduction complete, we need to ensure that the level of each element of V in an alphabetic tree can be chosen freely among $\lambda_1, \ldots, \lambda_k$. We are going to use a certain number of dummy leaves. The dummy leaves do not influence the level set functions, i.e., adding or removing dummy leaves never changes a function value. This clearly preserves submodularity.

Assume that we have a leaf sequence $q_i := v_i, z_1, \ldots, z_k$, where all leaves but v_i are dummy leaves. For each $j = 1, \ldots, k$ there exists an alphabetic tree $T^{v_i:j}$ for the sequence where v_i is on level j. The tree can be implemented as follows: The path to v_i is a pure left path, and for $h = 1, \ldots, j-1$ the right child of the level h node on that path is leaf z_{j-h}. The level 0 node of the path is the tree root, and its right subtree is some alphabetic tree for $z_j, \ldots, z_k$.

Let T_{stub} be the smallest complete binary tree with $m' \geq m$ leaves. In other words, the depth of T_{stub} is $d := \lceil \log_2 m \rceil$, and it has $m' := 2^d$ leaves. We now determine the levels associated with $u_i \in U$ as

$$\lambda_i := d + i, \quad i = 1, \ldots, k.$$

With this definition we now have completely determined the level cost functions. The leaf sequence of the SFAT instance is constructed as

$$q_1, \ldots, q_m, y_{m+1}, \ldots, y_{m'},$$

with q_i being the sequences defined as above and dummy leaves $y_{m+1}, \ldots, y_{m'}$. We achieve that any feasible solution to the MP instance can indeed be represented as an alphabetic tree, because the level of each $v_i \in V$ can be freely chosen among $\lambda_1, \ldots, \lambda_m$: Simply replace the ith leftmost leaf of T_{stub} with $T^{v_i:j}$, where λ_j is the desired level of v_i. The remaining $m' - m$

rightmost leaves of T_{stub} are replaced with $y_{m+1}, \ldots, y_{m'}$, and we obtain a feasible solution to the SFAT instance.

We have proven the following theorem.

Theorem 4. *SFAT is NP-hard, even if the set functions are all submodular.*

6 Conclusion

In this work we have considered the Huffman and Alphabetic Tree Problem with general cost functions. It has turned out that the most general problem versions are NP-hard, but optimal solutions can be computed in polynomial time under realistic assumptions about the cost functions. One interesting open problem that remains is the computational complexity of the Huffman tree problem with nondecreasing but not necessarily convex cost functions.

As for the problem cases that have turned out to be computationally tractable, a natural question for future research is about tight bounds for the runtime. To our knowledge the only known lower runtime bounds are a reduction to the classic Huffman problem from sorting, and the $\Omega(n \log n)$ lower bound for the classic Alphabetic Tree Problem under a certain model of computation [KM93].

Acknowledgment: We wish to thank Sebastian Kamprath for a helpful comment regarding the presentation of Section 2.1.

References

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

[Bae10] M. B. Baer. Alphabetic coding with exponential costs. *Information Processing Letters*, 110(4):139–142, 2010.

[BD10] P. Bose and K. Douïeb. Should static search trees ever be unbalanced? In *Proc. ISAAC '10*, volume 6506 of *LNCS*, pages 109–120. Springer, 2010.

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

[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. Springer, 2009.

[Gar74] M. R. Garey. Optimal binary search trees with restricted maximal depth. *SIAM Journal on Computing*, 3(2):101–110, 1974.

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

[Got81] L. Gotlieb. Optimal multi-way search trees. *SIAM Journal on Computing*, 10(3):422–433, 1981.

[GW77] A. M. Garsia and M. L. Wachs. A new algorithm for minimum cost binary trees. *SIAM Journal on Computing*, 6(4):622–642, 1977.

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

[HLM05] T. C. Hu, L. L. Larmore, and J. D. Morgenthaler. Optimal integer alphabetic trees in linear time. In *Proc. ESA '05*, volume 3669 of *LNCS*, pages 226–237. Springer, 2005.

[HT71] T. C. Hu and A. C. Tucker. Optimal Computer Search Trees and Variable-Length Alphabetical Codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[Hu73] T. C. Hu. A new proof of the T-C algorithm. *SIAM Journal on Applied Mathematics*, 25(1):83–94, 1973.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. the Institute of Radio Engineers*, volume 40, pages 1098–1101, 1952.

[Ita76] A. Itai. Optimal alphabetic trees. *SIAM Journal on Computing*, 5(1):9–18, 1976.

[JCLM10] T. Jacobs, F. Cicalese, E. Laber, and M. Molinaro. On the complexity of searching in trees: average-case minimization. In *Proc. ICALP '10*, volume 6198 of *LNCS*, pages 527–539. Springer, 2010.

[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.

[KLR97] M. Karpinski, L. L. Larmore, and W. Rytter. Correctness of constructing optimal alphabetic trees revisited. *Theoretical Computing Science*, 180:309–324, 1997.

[KM93] 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 Informatica*, 1:14–25, 1971.

[Kra49] L. G. Kraft. A device for quantizing, grouping, and coding amplitude modulated pulses. Master's thesis, Massachusetts Institute of Technology, 1949.

[Lar87] L. L. Larmore. Height restricted optimal binary trees. *SIAM Journal on Computing*, 16(6):1115–1123, 1987.

[LH90] L. L. Larmore and D. S. Hirschberg. Length-limited coding. In *Proc. SODA '90*, pages 310–318, 1990.

[LP94] L. L. Larmore and T. M. Przytycka. A fast algorithm for optimum height-limited alphabetic binary trees. *SIAM Journal on Computing*, 23:1283–1312, 1994.

[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.

[Wes76] R. L. Wessner. Optimal alphabetic search trees with restricted maximal height. *Information Processing Letters*, 4(4):90–94, 1976.

[Yeu91] R. W. Yeung. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37(3):564–572, 1991.

[ZNI05] L. Zhao, H. Nagamochi, and T. Ibaraki. Greedy splitting algorithms for approximating multiway partition problems. *Mathematical Programming*, 102:167–183, 2005.