The Huffman Tree Problem with Unit Step Functions

Hiroshi Fujiwara Takuya Nakamura Toshihiro Fujito

Abstract

A binary tree is regarded as a prefix-free binary code, in which the weighted sum of the lengths of root-leaf paths is equal to the expected codeword length. Huffman's algorithm computes an optimal tree in $O(n \log n)$ time, where n is the number of leaves. The problem was later generalized by allowing each leaf to have its own function of its depth and setting the sum of the function values as the objective function. The generalized problem was proved to be NP-hard. In this paper we study the case where every function is a unit step function, that is, a function that takes a lower constant value if the depth does not exceed a threshold, and a higher constant value otherwise. We show that for this case, the problem can be solved in $O(n \log n)$ time, by reducing it to the Coin Collector's problem.

1 Introduction

In the *Huffman tree problem* [LH90] we are given a sequence of n weights, and asked to construct a binary tree of minimum weighted sum of the lengths of root-leaf paths. A solution stands for a prefix-free binary code of minimum expected codeword length for a given sequence of probabilities, in which each root-leaf path represents a codeword.

The *General Cost Huffman Tree Problem* [FJ14] is an extension of the Huffman tree problem. In this problem we are given a sequence of n functions, each of which is associated with a leaf and maps the depth of the leaf to a cost value. The goal is to construct a binary tree of minimum sum of the function values.

A typical application of the General Cost Huffman Tree Problem can be found in the structural design of a website. Suppose that we are going to design a website for an organization consisting of many groups. Each group has a preference to the arrangement of their page in the website. A group may strongly insist that their page should be accessed in one or two clicks from the front page. Another group may simply wish that their page is placed as closer to the front page as possible. By formulating such preferences as an instance and solving it, we can get a total optimal structure.

The original Huffman tree problem can be seen as a special case of the generalized problem where each function is linear, and can be solved in $O(n \log n)$ time by Huffman's algorithm [Huf52]. On the other hand, the General Cost Huffman Tree Problem is known to be NP-hard even if every function is a zero-one function, that is, a function whose image is the set $\{0, 1\}$ [FJ14]. It has been open whether the General Cost Huffman Tree Problem for zero-one functions with some good property can be solved in polynomial time.

1.1 Our Contribution

In this paper we give an affirmative answer to the question by presenting an algorithm for the case where every function is a non-decreasing zero-one function. More specifically, we establish

[0]This work was supported by KAKENHI (23700014, 23500014, and 26330010). The final publication is available at `http://doi.org/10.1587/transfun.E98.A.1189`.

Table 1: Complexity results for the General Cost Huffman Tree Problem.

class of cost functions	time complexity
linear	$O(n \log n)$ [Huf52]
unit step	$O(n \log n)$ [this paper]
non-decreasing convex	$O(n^2 \log n)$ [FJ14]
general	NP-hard [FJ14]

a stronger result: Our algorithm can deal with the case where each function is a *unit step function*, that is, a function that takes a lower constant value if the depth of the leaf does not exceed a threshold, and a higher constant value otherwise.

In the context of designing a website, this case corresponds to a situation that: Each group has an upper limit on the number of clicks needed to reach their page from the front page. And the penalty of breaking the limit is different between groups, which may represent the power balance among groups in the organization.

The main idea of our algorithm is to find a set of leaves that each take their lower constant values, by reduction to the Coin Collector's problem. The running time remains $O(n \log n)$ as the same as Huffman's algorithm for the original problem. Table 1 summarizes the previous and our results.

1.2 Related Work

We have already mentioned that Huffman's algorithm [Huf52] solves the original Huffman tree problem in $O(n \log n)$ time. Fujiwara and Jacobs [FJ14] gave an $O(n^2 \log n)$-time algorithm for the General Cost Huffman Tree Problem with non-decreasing convex functions. Their idea was to extend the $O(nL)$-time algorithm of Larmore and Hirschberg [LH90] that solves a version of the original Huffman tree problem with a maximum height of L. Unfortunately, as we will discuss in Section 4, it seems difficult to apply Fujiwara and Jacobs's technique to our case. In their paper [FJ14] it is also proved that the General Cost Huffman Tree Problem is NP-hard even for zero-one functions. Another version of the generalized tree problem in which the order of leaves is given as an additional input has also been intensively studied [LP94, FJ14].

2 General Cost Huffman Tree Problem with Unit Step Functions

Throughout this paper we assume that any binary tree has a *root* r even when it is not explicitly specified. Also, we discuss only undirected binary trees. For vertices v and w in a binary tree, we denote the path from v to w simply by the *path* v-w. The *distance* between the vertices v and w, denoted by $\mathrm{dist}(v, w)$, is the number of edges along the path v-w, which we sometimes call the *length* of the path v-w. A *leaf* in a binary tree is a vertex that is of degree one and is not the root. The *depth* of leaf l in a binary tree T is defined as $\mathrm{dist}(r, l)$, which we denote $\mathrm{depth}(l, T)$. A *full* binary tree is a binary tree such that all the non-leaf vertices have two children. We denote the set of vertices and the set of edges in a binary tree T by $V(T)$ and $E(T)$, respectively. Let $\mathbb{N}$ denote the set of positive integers.

The *General Cost Huffman Tree Problem*, called *GHT* hereafter, is formulated as below.

GHT

Input: A sequence of functions $f_1, f_2, \ldots, f_n : \mathbb{N} \to \mathbb{Q}$.

Output: A binary tree having leaves $l_1, l_2, \ldots, l_n$.

Objective: Minimize $f(T) := \sum_{i=1}^{n} f_i(\mathrm{depth}(l_i, T))$.

Note that if we apply

$$f_i(x) = w_i x$$

with $0 < w_i < 1$ for all $1 \leq i \leq n$, GHT is equivalent to the original Huffman tree problem.

In this paper we focus on functions defined as

$$f_i(x) = \begin{cases} a_i, & x \leq t_i; \\ b_i, & \text{otherwise} \end{cases}$$

with $t_i \in \mathbb{N}$ and $a_i, b_i (> a_i) \in \mathbb{Q}$ for all $1 \leq i \leq n$. We refer to such a function as a *unit step function*. Without loss of generality, we consider unit step functions that satisfy the two conditions as follows:

(a) Each function f_i takes value 0 or $r_i (> 0) \in \mathbb{Q}$.

(b) For each i, $t_i \leq n - 1$ holds.

The condition (a) would be immediately acceptable without any arguments; one can redefine the objective function by subtracting a constant of $\sum_{i=1}^{n} a_i$. This enables us to characterize the functions just by t_i's and r_i's. The reason why we can assume the condition (b) is clarified below. We thus deal with GHT with unit step functions in the following normalized form:

Problem $(\mathcal{P})$

Input: A sequence of tuples $I = ((t_1, r_1), (t_2, r_2), \ldots, (t_n, r_n)) \in (\mathbb{N} \times \mathbb{Q})^n$ with $t_i \leq n - 1$ and $r_i > 0$ for all $1 \leq i \leq n$.

Output: A binary tree T having leaves $l_1, l_2, \ldots, l_n$.

Objective: Minimize $f(T) := \sum_{i=1}^{n} f_i(\mathrm{depth}(l_i, T))$, where $f_i(x) = 0$ if $x \leq t_i$ and r_i otherwise.

Let $(\mathcal{P}')$ denote the problem $(\mathcal{P})$ without the condition (b). The validity of the condition (b) is achieved by repeatedly applying Lemma 2.

Lemma 1. *Let T be a non-full binary tree that is feasible to an instance of the problem $(\mathcal{P}')$. Suppose that T has leaves $l_1, l_2, \ldots, l_n$. Then, there exists a full binary tree T' with leaves $l'_1, l'_2, \ldots, l'_n$ such that $\mathrm{depth}(l'_i, T') \leq \mathrm{depth}(l_i, T)$ for each i, and $f(T') \leq f(T)$ hold true.*

Proof. Since T is not full, there is a non-root vertex v of degree two. Contract either of the two edges incident to v. The depth of every leaf that is a descendant of v decreases by one. Then, the contribution of such a leaf to the objective function decreases or remains the same. Repeat this while the tree is not full, and rename l_i to l'_i for each i. Finally we obtain a full binary tree T' such that $\mathrm{depth}(l'_i, T') \leq \mathrm{depth}(l_i, T)$ for each i, and $f(T') \leq f(T)$. □

Lemma 2. *Let $I = ((t_1, r_1), (t_2, r_2), \ldots, (t_n, r_n))$ be an instance of the problem $(\mathcal{P}')$ such that for some j, $t_j \geq n$. Also, let I' be the same instance as I except that $t_j = n - 1$. Suppose that a binary tree T_0 is optimal for the instance I'. Then, T_0 is optimal also for the instance I.*

Proof. We write the objective function with respect to the instance I as f, and that with respect to the instance I' as f'.

We first show $\mathrm{depth}(l_j, T_0) \leq n-1$. Assume that $\mathrm{depth}(l_j, T_0) \geq n$. Then, T_0 is not a full binary tree, since in a full binary tree with n leaves, any leaf has a depth no larger than $n-1$. By Lemma 1, there is a full binary tree T_0' that satisfies $\mathrm{depth}(l_i', T_0') \leq \mathrm{depth}(l_i, T_0)$ for each i, and $f'(T_0') \leq f'(T_0)$. Since $\mathrm{depth}(l_j', T_0') \leq n-1$ holds by the same reason above, we have $f_j'(\mathrm{depth}(l_j', T_0')) = 0$. On the other hand, it holds $f_j'(\mathrm{depth}(l_j, T_0)) > 0$. One can see that for any $i \neq j$, $f_i'(\mathrm{depth}(l_i', T_0')) \leq f_i'(\mathrm{depth}(l_i, T_0))$. Therefore, we have $f'(T_0') < f'(T_0)$, which contradicts the optimality of T_0.

Hence, it follows that $\mathrm{depth}(l_j, T_0) \leq n-1$. By $f_j(\mathrm{depth}(l_j, T_0)) = f_j'(\mathrm{depth}(l_j, T_0)) = 0$, we have $f(T_0) = f'(T_0)$.

In the rest of the proof, we show the optimality of T_0 for the instance I. Assume that there is a binary tree T_1 such that $f(T_1) < f(T_0)$.

(i) The case where $\mathrm{depth}(l_j, T_1) \leq n-1$. Since $f_j'(\mathrm{depth}(l_j, T_1)) = f_j(\mathrm{depth}(l_j, T_1)) = 0$ holds, we have $f'(T_1) = f(T_1)$. Then, it is derived that $f'(T_1) = f(T_1) < f(T_0) = f'(T_0)$, which contradicts the optimality of T_0.

(ii) The case where $\mathrm{depth}(l_j, T_1) \geq n$. Again, by the property of a full binary tree and Lemma 1, there is a full binary tree T_1' that satisfies $f(T_1') \leq f(T_1)$. By $f_j'(\mathrm{depth}(l_j', T_1')) = f_j(\mathrm{depth}(l_j', T_1')) = 0$, we have $f'(T_1') = f(T_1')$. Then, we obtain $f'(T_1') = f(T_1') \leq f(T_1) < f(T_0) = f'(T_0)$, which contradicts the optimality of T_0.

Hence, it is concluded that T_0 is optimal also for the instance I. □

3 Kraft's Inequality and Construction of Trees

Our algorithm for GHT with unit step functions, given in Section 5, roughly consists of two steps: to determine the depth of each leaf and to construct a binary tree based on the sequence of depths. We here present some properties on binary trees and a subroutine for constructing a binary tree. The inequality in the following lemma is known as Kraft's Inequality [Kra49].

Lemma 3. *([Kra49]) Let T be a binary tree having leaves $l_1, l_2, \ldots, l_n$ which are located in the depth $d_1, d_2, \ldots, d_n$ in T, respectively. Then, it holds that*

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

The converse statement of Lemma 3 is also true. In this section, however, we give a constructive proof: We present a simple algorithm constructTree for constructing a binary tree. The fact that constructTree runs in linear time will later help the evaluation of our algorithm for GHT with unit step functions.

Lemma 4. *Let $(d_1, d_2, \ldots, d_n) \in \mathbb{N}^n$ be such that $\sum_{i=1}^{n} 2^{-d_i} \leq 1$, $d_i \leq d_{i+1}$ for $1 \leq i \leq n-1$, and $d_i \leq h(n)$ for some h and $1 \leq i \leq n$. Then,* constructTree *for $(d_1, d_2, \ldots, d_n)$ computes in $O(n + h(n))$ time a binary tree T with leaves $l_1, l_2, \ldots, l_n$ such that $\mathrm{depth}(l_i, T) = d_i$ for $1 \leq i \leq n$.*

Proof. (I) We first show that constructTree never gets stuck. In each iteration, it is observed that the algorithm gets stuck either (i) when the algorithm fails to find the vertex v, or (ii) when $d_i - \mathrm{dist}(r, v) - 1 < 0$ and the algorithm therefore fails to create a path w-l_i.

One can easily know that the case (ii) does not happen. Indeed, since the sequence $(d_1, d_2, \ldots, d_n)$ is ordered in ascending order, it follows that $\mathrm{dist}(r, v) \leq d_{i-1} - 1 \leq d_i - 1$.

Algorithm 1: constructTree

Input : A sequence $(d_1, d_2, \ldots, d_n) \in \mathbb{N}^n$
Output : A binary tree T with the depths of its leaves being $d_1, d_2, \ldots, d_n$
Assume: $(d_1, d_2, \ldots, d_n)$ is sorted in ascending order and $\sum_{i=1}^{n} 2^{-d_i} \leq 1$

```
Create a root vertex r;
Construct a path r-l_1 of length d_1 by repeatedly creating a left child;
for i ← 2 to n do
    v ← the first vertex (≠ l_{i−1}) on the path from l_{i−1} to r that does not have a right child;
    Create a right child w of v;
    if d_i − dist(r, v) − 1 = 0 then
        l_i ← w;
    else
        Construct a path w-l_i of length (d_i − dist(r, v) − 1) by repeatedly creating a left child;
    end
end
return the constructed tree;
```

The rest is to show that the case (i) does not occur. Consider a full binary tree with 2^{d_n} leaves such that all the leaves are at depth d_n. Note that d_n is the largest among $d_1, d_2, \ldots, d_n$. One can interpret the behavior of constructTree as successively choosing a vertex of the full binary tree and cutting down its descendants. Therefore, it is sufficient to show that constructTree can always choose a vertex of depth d_i in the full binary tree.

We first claim that in each iteration, what is cut down from the full binary tree includes the left most leaf of that tree. For the case of l_1 our claim is trivial. Assume that our claim holds for $l_1, \ldots, l_{i-1}$. Observe that until constructTree finds v on the path l_{i-1}-r, it looks only edges between a right child and its parent. In fact, it encounters an edge between a left child and its parent, then the right child of the parent should already have vanished with its descendants. This contradicts the assumption. Therefore, the vertex w is on the left most path, and so is the vertex l_i. Hence, our claim is correct.

At the beginning the full binary tree has 2^{d_n} leaves. In the j-th iteration it decreases by $2^{d_n - d_j}$. We calculate

$$\begin{aligned} 2^{d_n} - \sum_{j=1}^{i-1} 2^{d_n - d_j} &= 2^{d_n}(1 - \sum_{j=1}^{i-1} 2^{-d_j}) \\ &> 2^{d_n}(1 - \sum_{j=1}^{n} 2^{-d_j}) \\ &\geq 0, \end{aligned}$$

which means that immediately after the i-th iteration has started, there remains at least one leaf in the full binary tree. Thus, the algorithm can always find the vertex v.

(II) We next evaluate the running time. Instead of summing up the number of steps in each part, we consider the total running time based on the resulting tree T. It is observed that constructTree traces each edge of T exactly twice. Recall that $|E(T)| = |V(T)| - 1$ for any tree. In the rest of the proof we count the vertices, excluding the root.

We begin by bounding the number of vertices of degree two. We here prove the fact that after the i-th iteration, vertices of degree two all lie on the path r-l_i. For the case of $i = 1$, this is trivial. Assume that the fact is true by the end of the $(i-1)$-th iteration. In the i-th iteration, a path r-v-l_i is created after choosing v on the path r-l_{i-1}. The vertex v is the first vertex that does not have a right child along the path from l_{i-1} to r. In other words, the vertex is chosen so that there is no vertex of degree two on the path v-l_{i-1}. Therefore, vertices of degree two on the path r-v-l_{i-1}, if any, lie on the path r-v. Hence, the fact holds true for i-th iteration. The fact implies that the number of vertices of degree two in T is bounded by $d_n - 1 \leq h(n) - 1$.

One can straightforwardly know about the number of vertices of degree one and three: Every vertex of degree one in T is a leaf. The vertices of degree three increase by one on each iteration. Therefore, there are $n - 1$ vertices in T at the end.

Together with the root, we know $|V(T)| \leq 1 + n + h(n) - 1 + n - 1 = 2n + h(n) - 1$. Hence, we conclude that the running time is $O(n + h(n))$. □

The algorithm constructTree depends on the assumption that $(d_1, d_2, \ldots, d_n)$ is sorted in ascending order. Note that for example, the algorithm gets stuck for $(3, 2, 2, 3, 3, 3)$; the algorithm fails to find the vertex v for the last leaf. On the other hand, the algorithm works for $(2, 2, 3, 3, 3, 3)$.

4 Coin Collector's Problem

Our algorithm for GHT with unit step functions first checks whether for the given instance, it can return a binary tree such that the objective function takes value zero. If it is not the case, our algorithm calls the packageMerge algorithm as a subroutine, presented as Algorithm 2, for solving the *Coin Collector's problem* [LH90]. In the Coin Collector's problem, one is given a set of *coins* $1, 2, \ldots, N$ each associated with a *width* s_i being a power of two and a rational *weight* c_i, and a positive integer K. (From the viewpoint of coin collectors, the width is the face value of the coin, whereas the weight is the value of the coin common among coin collectors.) The goal is to find a subset of the coins of minimum total weight such that its total width is equal to K. We formulate this as follows:

Coin Collector's problem

Input: A sequence of tuples $J = ((s_1, c_1), (s_2, c_2), \ldots, (s_N, c_N))$ with each $s_i \in \{2^{-j} \mid j \in \mathbb{N}\}$ and each $c_i \in \mathbb{Q}$, and $K \in \mathbb{N}$.

Output: A subset Y of $\{1, 2, \ldots, N\}$ such that $\sum_{i \in Y} s_i = K$.

Objective: Minimize $c(Y) := \sum_{i \in Y} c_i$.

Without loss of generality, we have assumed here that the width of each coin is smaller than one. This problem can been seen as a special case of the Knapsack Problem. It is known that the algorithm packageMerge solves this problem in linear time with some preprocessing.

Lemma 5. *([LH90]) Let (J, K) be an instance of the Coin Collector's problem with N coins such that J is sorted by width in ascending order, then by weight in ascending order. Then, the* packageMerge *algorithm computes an optimal solution to the instance (J, K) in $O(N)$ time.*

Algorithm 2: packageMerge [LH90] for the Coin Collector's problem

Input : A sequence of tuples $J = ((s_1, c_1), (s_2, c_2), \ldots, (s_N, c_N))$ with each $s_i \in \{2^{-j} \mid j \in \mathbb{N}\}$ and each $c_i \in \mathbb{Q}$, and $K \in \mathbb{N}$
Output : A subset Y of $\{1, 2, \ldots, N\}$ such that $\sum_{i \in Y} s_i = K$
Assume: J is sorted by width in ascending order, then by weight in ascending order

```
D ← −log2 s_1;                                   // coin 1 is of smallest width
for d ← D downto 1 do
    A_d ← sequence of coins of width 2^{−d}, sorted by weight
end
for d ← D downto 1 do
    while |A_d| ≥ 2 do
        Remove the first pair of coins from A_d;          // the lightest two coins
        Define a new coin: let its width be d − 1 and its weight be the sum of weights
        of the removed pair;
        Record from which coins in J the new coin was defined;
        Insert the new coin into A_{d−1} with maintaining the order by weight;
    end
end
return sequence of coins that define the first K coins in A_0.
```

Our algorithm for GHT with unit step functions converts a given instance of the problem $(\mathcal{P})$ into that of the Coin Collector's problem as follows: Given an instance $I = ((t_1, r_1), (t_2, r_2), \ldots, (t_n, r_n))$ of the problem $(\mathcal{P})$, let

$$\begin{aligned} J = &((2^{-t_1}, -r_1), (2^{-t_2}, -r_2), \ldots, (2^{-t_n}, -r_n), \\ &(2^{-1}, 0), (2^{-2}, 0), \ldots, (2^{-n+1}, 0), \\ &(2^{-n+1}, -R - 1)), \end{aligned} \tag{1}$$

$$K = 1, \tag{2}$$

where $R = \max_{1 \le i \le n} r_i$.

The converted instance has $2n$ coins in total. The first n coins in J correspond to the leaves in the problem $(\mathcal{P})$. As seen, each of the $2n$ coins is assigned a negative weight. Intuitively, the weight $-r_i$ stands for the fact that: if one is allowed to relocate leaf i from depth over t_i to depth t_i, then the value of the objective function decreases by r_i. Our idea is to find a set of leaves whose contribution to the objective function is zero.

The following $n-1$ coins in J are dummy coins of weight zero, which we call the *zero-weight dummy coins*. The last coin in J is referred to as the *last dummy coin*. The aim of our conversion is to choose a set of leaves, in the context of the problem $(\mathcal{P})$, so that their total width does not exceed $1 - 2^{-n+1}$. An intuitive reason why we set $1 - 2^{-n+1}$, not 1, is that the resulting binary tree has to reserve "space" of width 2^{-n+1} for the leaves that have positive contributions to the objective function. The zero-weight dummy coins are introduced for dealing with this constraint on the total width as an equality. In the proof of Lemma 8, we will clarify how this set of zero-weight dummy coins plays a role.

Although we can give a version of the packageMerge algorithm that accepts K of the form of $\sum_i 2^{-j_i}$ $(j_1, j_2, \ldots \in \mathbb{N})$, the presented version for integer K is much simpler. The reason why we have added the last dummy coin is that we would like to set $K = 1$. (That is to say, we can

have an equivalent solution if we employ J without the last dummy coin, and $K = 1 - 2^{-n+1}$.) Lemma 6 guarantees that the last dummy coin is always included in the solution.

We have seen that all coins in the converted instance have non-positive weight. Although the Coin Collector's problem is originally a minimization problem, one can think of the instance as a maximization problem. The PACKAGEMERGE algorithm works correctly even for such an instance.

Lemma 6. *Any optimal solution to the instance (J, K) in (1) and (2) contains the last dummy coin.*

Proof. Consider an arbitrary optimal solution. We show the lemma by claiming that neither of the following two cases arises.

(I) Assume that the optimal solution does not include either a zero-weight dummy coin or the last dummy coin. Choose a coin in the solution which appears in the first n entries in J, that is, one which corresponds to a leaf in the problem $(\mathcal{P})$. Suppose that its width is 2^{-t} and its weight is $-r$. This coin can be replaced by a union of some zero-weight coins and the last dummy coin. Indeed, if we combine the zero-weight dummy coins of weight $2^{-t-1}, 2^{-t-2}, \ldots, 2^{-n+1}$, then the total width is

$$2^{-t-1} + 2^{-t-2} + \cdots + 2^{-n+1} = 2^{-t} - 2^{-n+1}.$$

This width plus the width of the last dummy coin makes exactly 2^{-t}. By replacing coins in this way, the objective function decreases by $-r - (-R - 1) > 0$, which contradicts the optimality.

(II) Assume that the optimal solution includes at least one zero-weight dummy coin and does not include the last dummy coin. Choose a zero-weight dummy coin with largest width. Suppose that its width is 2^{-t}. Similarly as (I), it is seen that this zero-weight coin can be replaced by the union of the last dummy coin and the zero-weight dummy coins of width $2^{-t-1}, 2^{-t-2}, \ldots, 2^{-n+1}$, which are not in the solution. This replacement of coins decreases the objective function. Therefore, the assumption again turns out to be false. □

We introduced in Section 1.2 that the case where the associated functions are all non-decreasing convex functions is solved in $O(n^2 \log n)$ time [FJ14], which is also based on reduction to the Coin Collector's problem. We here remark that the reduction is different from that in this paper: Whereas the reduction of [FJ14] involves n^2 coins, our reduction requires just $2n$ coins.

We mention a bit more of the reduction of [FJ14]. In the converted instance according to the reduction, the coins $(i, 1), (i, 2), \ldots, (i, j)$ are included in the solution if and only if the leaf i is located at the depth j in the resulting binary tree. On the other hand, in our instance, the coin i is included in the solution if and only if the depth of the leaf i is equal to or smaller than t_i.

One should note also that a unit step function is not convex, since

$$\frac{f_i(t_i) + f_i(t_i + 2)}{2} = \frac{r_i}{2}$$

whereas

$$f(t_i + 1) = r_i > \frac{r_i}{2}.$$

Although we do not present the detail, the reduction of [FJ14] in fact fails for an instance with a unit step function.

Algorithm 3: solveGHTUS

Input : A sequence of tuples $I = ((t_1, r_1), (t_2, r_2), \ldots, (t_n, r_n)) \in (\mathbb{N} \times \mathbb{Q})^n$
Output : An optimal binary tree T
Assume: $t_i \leq n-1$ and $r_i > 0$ for all $1 \leq i \leq n$

if $\sum_{i=1}^{n} 2^{-t_i} \leq 1$ **then** `// f(T) = 0`
  $S \leftarrow$ sequence of t's in I, sorted by in ascending order;
  return constructTree(S);
else `// f(T) > 0`
  $R \leftarrow \max_{1 \leq i \leq n} r_i$;
  $J \leftarrow ((2^{-t_1}, -r_1), (2^{-t_2}, -r_2), \ldots, (2^{-t_n}, -r_n), (2^{-1}, 0), (2^{-2}, 0), \ldots, (2^{-n+1}, 0), (2^{-n+1}, -R-1))$;
  $K \leftarrow 1$; `// construct instance`
  Sort J so that (s_i, c_i) precedes (s_j, c_j) if either (i) $s_i = s_j$ and $c_i \leq c_j$, or (ii) $s_i < s_j$; `// sort by width and then by weight`
  $Y \leftarrow$ packageMerge(J, K);
  $S \leftarrow$ sequence of t_i's such that $i \in Y$ and $-R-1 < r_i < 0$; `// discard dummy coins`
  Sort S in ascending order;
  Repeat $(n - |S|)$ times: Append $(n - 1 + \lceil \log_2(n - |S|) \rceil)$ to S;
  return constructTree(S);
end

5 Algorithm for GHT with unit step functions

We present here our algorithm solveGHTUS for GHT with unit step functions as Algorithm 3. The first "if" sentence checks whether the algorithm can return a binary tree with objective value zero. Only when the answer is "no", the algorithm converts the given instance to the Coin Collector's problem. We explain why this check is necessary. Suppose that the given instance satisfies $\sum_{i=1}^{n} 2^{-t_i} = 1$ and the algorithm converts it to the instance (J, K) in (1) and (2). Lemma 4 then claims that employing constructTree, one can have a binary tree T such that the leaves are located at exactly depth $t_1, t_2, \ldots, t_n$. Hence, $f(T) = 0$. On the other hand, by Lemma 6, any optimal solution to (J, K) contains the last dummy coin. The total width of non-dummy coins is thus no greater than $1 - 2^{-n+1}$, which means that some non-dummy coin has been excluded from the solution. The leaf corresponding to such a non-dummy coin is at depth $(n - 1 + \lceil \log_2(n - |S|) \rceil)$. Consequently, the value of the objective function becomes positive. In this way, if the algorithm skips the check at the beginning, it may not return an optimal solution.

For an instance with $\sum_{i=1}^{n} 2^{-t_i} > 1$, Lemma 3 says that there is some leaf of depth greater than t_i in an optimal binary tree. In a sense, the subroutine packageMerge excludes an optimal set of such leaves from its solution. The excluded leaves are determined to have positive contributions to the objective function. Our algorithm puts all these leaves at depth $(n - 1 + \lceil \log_2(n - |S|) \rceil)$. The last dummy coin in the converted instance creates a path from the root to these leaves.

The choice of the depth for such leaves is just a matter of taste. The reason why our algorithm uses the depth $(n - 1 + \lceil \log_2(n - |S|) \rceil)$ is simple. The algorithm knows that there are $n - |S|$ leaves whose function values are determined to be positive. In a full binary tree with $n - |S|$ leaves of minimum height, each leaf has a depth at most $\lceil \log_2(n - |S|) \rceil$. Our algorithm

constructs such a subtree under a vertex of depth $n-1$. Thus, the $n-|S|$ leaves have a depth $(n-1+\lceil \log_2(n-|S|)\rceil)$.

5.1 Correctness of the algorithm

We first show the correctness for an instance $I=((t_1,r_1),(t_2,r_2),\ldots,(t_n,r_n))$ with $\sum_{i=1}^{n} 2^{-t_i} > 1$.

Lemma 7. *Let T be a binary tree that is feasible to the instance $I=((t_1,r_1),(t_2,r_2),\ldots,(t_n,r_n))$ of the problem $(\mathcal{P})$, and $d_i = depth(l_i,T)$ for $1\le i\le n$. Let X be the set $\{i\in\mathbb{N} \mid d_i \le t_i, 1\le i\le n\}$. If $X\neq\{i\in\mathbb{N}\mid 1\le i\le n\}$, then $\sum_{i\in X} 2^{-t_i} \le 1-2^{-n+1}$ holds.*

Proof. By Lemma 3, we know that $\sum_{i=1}^{n} 2^{-d_i}\le 1$. Since

$$1 \ge \sum_{i=1}^{n} 2^{-d_i} > \sum_{i\in X} 2^{-d_i} \ge \sum_{i\in X} 2^{-t_i}$$

by the assumption of the lemma, we have $\sum_{i\in X} 2^{-t_i} < 1$.

We need a bit stronger inequality. We derive $\sum_{i\in X} 2^{-t_i} = 2^{-n+1}\sum_{i\in X} 2^{n-1-t_i}$. Since $n-1-t_i$ is a non-negative integer for each i, $\sum_{i\in X} 2^{n-1-t_i}$ is also a non-negative integer. Denote the value of $\sum_{i\in X} 2^{n-1-t_i}$ as A. Letting $B=2^{n-1}$, we have $A<B$ by the inequality $\sum_{i\in X} 2^{-t_i}<1$. Noting that the both are integers, it should hold that $A\le B-1$. Thus, we have

$$\sum_{i\in X} 2^{-t_i} = 2^{-n+1}A \le 2^{-n+1}(B-1) = 1-2^{-n+1}.$$

□

Lemma 8. *Let T be a binary tree that is feasible to the instance $I=((t_1,r_1),(t_2,r_2),\ldots,(t_n,r_n))$ of the problem $(\mathcal{P})$ with $\sum_{i=1}^{n} 2^{-t_i} > 1$. There exists a solution Y to the instance (J,K) in (1) and (2) of the Coin Collector's problem such that $c(Y) = f(T) - \sum_{i=1}^{n} r_i - R - 1$, where $R=\max_{1\le i\le n} r_i$.*

Proof. Let $d_i = \mathrm{depth}(l_i,T)$ for $1\le i\le n$, and X be the set $\{i\in\mathbb{N}\mid d_i\le t_i, 1\le i\le n\}$. We first confirm that $X\neq\{i\in\mathbb{N}\mid 1\le i\le n\}$. Indeed, otherwise we have

$$\sum_{i=1}^{n} 2^{-d_i} = \sum_{i\in X} 2^{-d_i} \ge \sum_{i\in X} 2^{-t_i} = \sum_{i=1}^{n} 2^{-t_i} > 1.$$

Lemma 3 says that there exists no such binary tree T.

By Lemma 7, we know that

$$\sum_{i\in X} 2^{-t_i} \le 1-2^{-n+1}.$$

Suppose that

$$0.z_1z_2\cdots z_{n-1}$$

is the binary representation of

$$1-2^{-n+1}-\sum_{i\in X} 2^{-t_i}.$$

Note that this is always possible; since $t_i\le n-1$ for all i, no smaller fraction than 2^{-n+1} appears. Let $D=\{n+i\mid z_i=1, 1\le i\le n-1\}$, which is a set of zero-weight dummy coins. Let $Y=X\cup D\cup\{2n\}$. Recall that the $2n$-th coin of the instance (J,K) is the last dummy coin

of width 2^{-n+1} and weight $(-R-1)$. From these arguments, we have $\sum_{i\in Y} 2^{-t_i} = 1$, that is to say, Y is feasible to the instance (J, K).

In the problem $(\mathcal{P})$, the leaves in X do not contribute to the objective function, while each of the other leaves adds r_i to the objective function. We thus have

$$f(T) = \sum_{i\notin X} r_i = \sum_{i=1}^{n} r_i - \sum_{i\in X} r_i.$$

Therefore,

$$\begin{aligned} c(Y) &= \sum_{i\in X}(-r_i) - R - 1 \\ &= \sum_{i=1}^{n} r_i - \sum_{i\in X} r_i - \sum_{i=1}^{n} r_i - R - 1 \\ &= f(T) - \sum_{i=1}^{n} r_i - R - 1. \end{aligned}$$

□

Lemma 9. *For an instance $I = ((t_1, r_1), (t_2, r_2), \ldots, (t_n, r_n))$ of the problem $(\mathcal{P})$ such that $\sum_{i=1}^{n} 2^{-t_i} > 1$, the algorithm* SOLVEGHTUS *computes an optimal solution in $O(n \log n)$ time.*

Proof. (I) First, we guarantee the running time. We begin by investigating that CONSTRUCTTREE(S) invoked in SOLVEGHTUS runs in $O(n)$ time, with the help of Lemma 4. Lemma 6 states that there is always the last dummy coin in Y. Thus, the sum of width over non-dummy coins in Y is no larger than $1 - 2^{-n+1}$. Therefore, immediately before CONSTRUCTTREE(S) is called, we can bound as

$$\begin{aligned} \sum_{w\in S} 2^{-w} &\le 1 - 2^{-n+1} + (n - |S|)2^{-(n-1+\lceil \log_2(n-|S|)\rceil)} \\ &\le 1 - 2^{-n+1} + (n - |S|)2^{-n+1}/(n - |S|) \\ &= 1. \end{aligned}$$

Besides, the maximum value in S is $n - 1 + \lceil \log_2(n - |S|)\rceil$ appearing at the tail, which is upper-bounded by $n + \log_2 n$. Hence, Lemma 4 guarantees the $O(n)$ running time.

The rest parts in SOLVEGHTUS, except sorting, are all executed in $O(1)$ or $O(n)$ time. The sorting is implemented with an $O(n \log n)$-time algorithm. Therefore, the overall running time is $O(n \log n)$.

(II) Next, we show the optimality of the returned tree T. Assume that there is a binary tree T_0 with n leaves such that $f(T_0) < f(T)$. Then, Lemma 8 states the existence of a solution Y_0 to the instance (J, K) such that $c(Y_0) = f(T_0) - \sum_{i=1}^{n} r_i - R - 1$. Now, we derive

$$\begin{aligned} c(Y) &= f(T) - \sum_{i=1}^{n} r_i - R - 1 \\ &> f(T_0) - \sum_{i=1}^{n} r_i - R - 1 \\ &= c(Y_0), \end{aligned}$$

which contradicts the optimality of Y. Therefore, T is an optimal solution to the problem $(\mathcal{P})$.
□

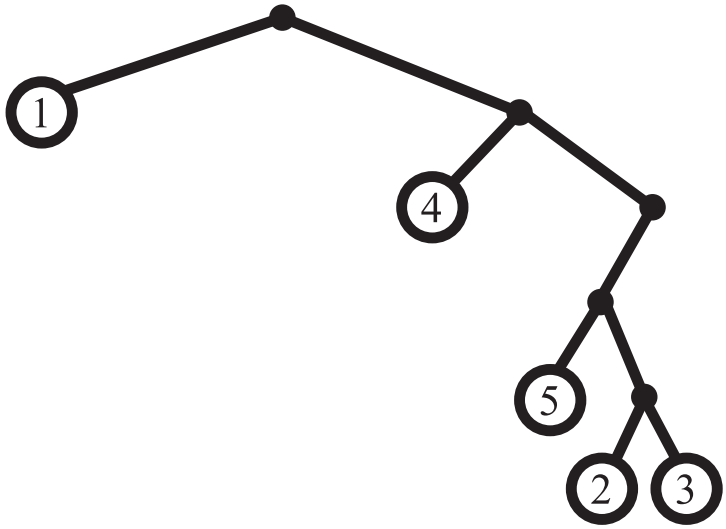

Figure 1: An optimal binary tree for instance $I = ((1, 4), (2, 1), (2, 2), (2, 3), (4, 5))$. The numbers on vertices identify leaves. Solid black points are non-leaf vertices. The objective value is $0 + 1 + 2 + 0 + 0 = 3$.

For an instance with $\sum_{i=1}^{n} 2^{-t_i} \leq 1$, the algorithm SOLVEGHTUS returns a binary tree with objective value zero in $O(n \log n)$ time; $O(n \log n)$ for sorting and $O(n)$ for CONSTRUCTTREE by Lemma 4. Together with the normalization in Section 2, which is done in $O(n)$ time, we establish our main theorem.

Theorem 1. *The algorithm* SOLVEGHTUS *solves GHT with unit step functions in* $O(n \log n)$ *time.*

5.2 Example

As an example, we trace the behavior of SOLVEGHTUS with an instance of the problem $(\mathcal{P})$:

$$I = ((1, 4), (2, 1), (2, 2), (2, 3), (4, 5)).$$

Since $\sum_{i=1}^{n} 2^{-t_i} = \frac{21}{2^4} > 1$, the algorithm SOLVEGHTUS converts the instance to that of the Coin Collector's problem. Adding dummy coins, the algorithm has

$$\begin{aligned} J = &((2^{-1}, -4), (2^{-2}, -1), (2^{-2}, -2), (2^{-2}, -3), (2^{-4}, -5), \\ &(2^{-1}, 0), (2^{-2}, 0), (2^{-3}, 0), (2^{-4}, 0), \\ &(2^{-4}, -6)), \end{aligned}$$

and $K = 1$. Note that the coins 6, 7, 8, 9, and 10 in J are dummy coins.

Next, PACKAGEMERGE is invoked after sorting J by width and then by weight. We keep on referring the i-th coin in J before sorting, displayed above, as the coin i. Immediately before PACKAGEMERGE returns a solution, the sequence A_0 has two coins: the first is a combined coin of $(2^0, -13)$ made of the coins 10, 5, 8, 4, and 1, while the last is a combined coin of $(2^0, -3)$ made of the coins 3, 2, and 6. Only the first one is picked up as a solution

$$Y = (1, 2, 4, 5, 9),$$

which corresponds to the coins 10, 5, 8, 4, and 1, respectively. Note that the entries of Y are indices in the sorted J. By discarding dummy coins and sorting the sequence, SOLVEGHTUS obtains a sequence of t_i's $S = (1, 2, 4)$, that is to say, the coins 1, 4, and 5. By appending

$$n - 1 + \lceil \log_2(n - |S|) \rceil = 5 - 1 + \lceil \log_2(5 - 3) \rceil = 5$$

twice, S finally becomes $(1, 2, 4, 5, 5)$. By calling CONSTRUCTTREE, the tree in Figure 1 is returned.

6 Discussion

The algorithm SOLVEGHTUS is designed with the intention of guarantee of its $O(n \log n)$ running time. Consequently, the algorithm may output an awkward tree; the leaves that are excluded from the solution of the Coin Collector's problem may be located at an unnecessarily deep level in the resulting tree. One can obtain a full binary tree by repeatedly contracting either of the two edges incident to each non-leaf vertex, as we did in the proof of Lemma 1.

In this paper we have seen instances consisting of only non-decreasing unit step functions. For an instance with some decreasing unit step functions, it is done if one puts such leaves at a sufficiently deep level in the tree. Nevertheless, if one is imposed the additional constraint that the output should be a full binary tree, the problem remains open.

References

[FJ14] H. Fujiwara and T. Jacobs. On the huffman and alphabetic tree problem with general cost functions. *Algorithmica*, 69(3):582–604, 2014.

[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.

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

[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.