Hang Lung Mathematics Awards
Vol. 3 (2008), pp. 53–87

SUFFICIENT CONDITION OF WEIGHT-BALANCE TREE

TEAM MEMBERS
Chi-Yeung Lam, Yin-Tat Lee[1]

SCHOOL
The Methodist Church HK Wesley College

Abstract. Huffman's coding provides a method to generate a weight-balanced tree, but it is not generating progressively. In other words, we cannot have meaningful output if we terminate the algorithm halfway in order to save time. For this purpose, we want to design an alternative algorithm, therefore this paper aims at finding out a sufficient condition of being a weight-balanced tree.

In this paper, we have found out a sufficient condition. Besides, as the solution of building a weight-balanced can be applied to solving other problems, we abstract the problem and discuss it in the manner of graph theory. The applications are also covered.

1. Introduction

In many situations, we need to compress information in order to transmit it effectively. However, Claude Shannon found that the rate of compression has lower bound, which is also known as Shannon entropy[1]. After a while, David A. Huffman found an algorithm to compress information by variable-length code which is optimal for independent input[2]. However, the algorithm is not suitable for some problems of compression. For example, we cannot terminate the algorithm halfway to have meaningful output in order to save time. Also, the algorithm assumes that the occurrence of the inputs are independent and requires the probabilities of all inputs. Therefore we want to create another algorithm for these purposes.

[1]This work is done under the supervision of the authors' teacher, Mr. Chun-Kit Ho

The process of finding the optimal code scheme, which is the same as finding the weight-balanced tree[2], has much more applications than we thought. For example, we found that we can accelerate the searching speed for some sequences by making a binary decision tree according to the probability distribution of each element being searched, as an alternative to using binary search or linear search. Therefore, we abstract the problems and try to answer this question: what a good sufficient condition of being a weight-balanced tree can be? If we answer this question, we may be able to design an alternative algorithm instead of Huffman's.

At first, in order to give us a better understanding and some tools for further investigation, we chose to investigate general binary trees. During the investigation, we found we can give sufficient conditions for some weight(or probability) distributions more easily, therefore, we solved these special cases first. We observed that the corresponding trees to these cases were similar, so we drew out the trees and investigated them. After that, we investigated the properties of the weight-balanced tree and found out the sufficient condition.

In this paper, besides the answer to the question, there are some by-products, which are also discussed.

This paper divided into 5 sections. Section 1 is this introduction. Section 2 is the preliminary, which will introduce some common definitions, some basic theorems and some operations about binary trees that will help us to prove. Section 3 is the main body of this paper, we will discuss the properties of weight-balance tree, the bound of the minimum weighted mean of heights of the leaves, different types of trees, different weight functions, a sufficient condition of being a weight-balanced tree and algorithms. Finally, Section 4 will discuss the applications and we will show that how this paper is related. Section 5 is conclusion that will summarize the main body.

2. Preliminary

In this section, we will introduce some terminologies which have been defined by someone else and widely used. Also we will prove some basic theorems that will be used in later sections.

[2]The terminology "weight-balanced tree" in this paper is referred to a binary tree with minimum weighted mean of heights of leaves among trees with the same weights on the corresponding leaves and the same number of leaves.

2.1. Terminology

This subsection introduces the terminologies and notations describing a tree, which will be used in the whole paper.

For all tree T, we have the following terminologies:

1. A **root** is a designated node and the root must exists. The root of T is denoted as $r(T)$ or simply r.
2. The **parent** of a node n in T is a neighbour of n and in the path from n to root, which is denoted as $\mathrm{par}_T(n)$ or simply $\mathrm{par}(n)$. Notice that $\mathrm{par}(r)$ is undefined, that is a root has no parent. We can simply call a parent of some nodes as a parent.
3. We say node n is a **child** of $\mathrm{par}(n)$.
4. A **sibling** of node a is another node b with $\mathrm{par}(a) = \mathrm{par}(b)$.
5. If the path from node a to root includes another node b, then b is an **ancestor** of a and a is a **descendant** of b.
6. A **leaf** is a node with no children.
7. $V(T)$ denotes the set of nodes of T.
8. $L(T)$ denotes the set of leaves of T, obviously $L(T) \subseteq V(T)$.
9. The **height** of a node n of T is the length of the path from the root to node n, which is denoted as $h_T(n)$ or simply $h(n)$. Note that the height of the root is zero.
10. The **height** of T is the maximum value of height of a node which is denoted as $h(T)$.
11. A **subtree** at node n is an induced subgraph. The nodes of a subtree are exactly n and all its descendant. The subtree of T at n is denoted as T_n.

Example 1. *Let a be the root*[3] *of tree T,*

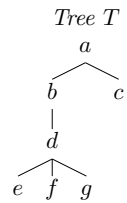


FIGURE 1

The following statements are true for T:

[3]For convenience, the root of a tree in this paper is always shown at the top of the graph.

1. $par(b) = a$, *and* b *is a child of* a.
2. b *has one child, while* d *has three children.*
3. e, f, g *are siblings;* b, c *are siblings; but* a *and* d *have no siblings.*
4. $h(a) = 0, h(b) = h(c) = 1, h(d) = 2, h(f) = h(g) = h(i) = 3$
5. $h(T) = 3$
6. $V(T) = \{a, b, c, d, e, f, g\}$
7. $L(T) = \{c, e, f, g\}$

2.2. Operation

Later we will perform two operations several times on trees in proofs, therefore we choose to state it first.

Operation 2. *A* ***swap*** *is an operation performed on two nodes which do not have ancestor-descendant relationship. After we swap node* a *and node* b*, the parents of the two nodes are exchanged. Let* T *and* T' *be the trees before and after the swap respectively. Here are some effects of this operation:*

1. *The number of leaves is unchanged.*
2. *The number of nodes is unchanged.*
3. *For all nodes* n_a *in* $V(T_a)$ *and* n_b *in* $V(T_b)$*, let* $k = h(b) - h(a)$*, we have*
$$\begin{cases} h_{T'}(n_a) = h_T(n_a) + k \\ h_{T'}(n_b) = h_T(n_b) - k \end{cases}$$

Example 3.

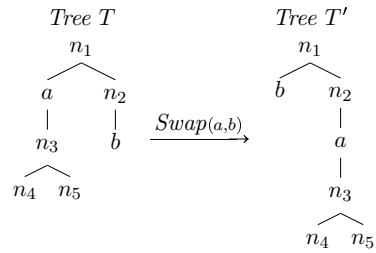


FIGURE 2

Operation 4. *A* ***merge*** *is an operation performed on a node and its only child* [4]*. After we merge the node* a *and the only child* b*, node* b *will be taken*

[4]It is because we will only perform merge on nodes and their only child in the following proofs.

away and all the children of b will become the children of a. Let T and T′ be the trees before and after merge respectively. Here is some effects of this operation:

1. *The number of leaves is unchanged.*
2. *The number of nodes is reduced by 1.*
3. *For all descendants of node b, the heights are decreased by 1, while heights of other nodes are unchanged.*

Note that even if b is a leaf, the number of leaves will not change after merging.

Example 5.

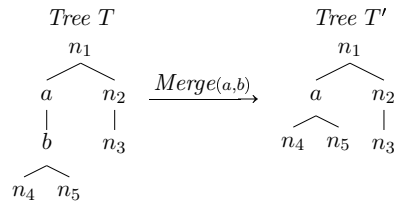


FIGURE 3

2.3. Binary Tree

In this paper, we only concern about binary trees, so that we do not need to handle many special cases. Also, as it is more common to use two alphabets to encode messages for computer, focusing on binary tree does not affect the importance of this research. In this subsection, we will discuss some basics about binary trees.

Definition 6. *A **binary tree** is a tree that all nodes have 0, 1 or 2 children.*

Example 7.

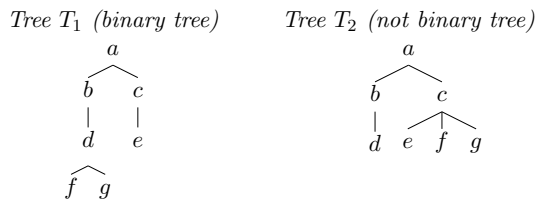


FIGURE 4

Tree T_1 is a binary tree while tree T_2 is not, because node c of T_2 has 3 children.

Before we directly investigate our problem, we want to eliminate some irrelevant binary trees. When we view binary tree as a decision tree, it is unusual to have a parent node with only one child since that node cannot reveal any information. Just like the following figure shown:

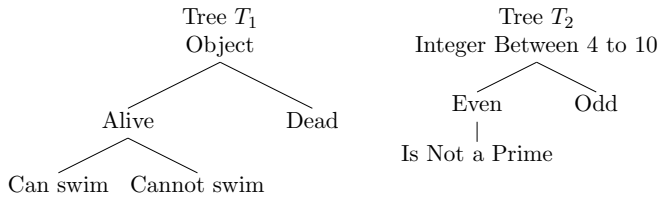


FIGURE 5

Since all even numbers between 4 and 10 are not prime numbers, tree T_2 has a redundant node and tree T_1 does not have. We said T_1 is proper and tree T_2 is improper[5].

As a proper tree is a more compact structure, we believe that weight-balanced tree is a proper tree. Also, we believe that there are some special properties in proper tree so that we can use them to find out the answer. Now, we define the proper tree first, then we will try to find some properties for proper tree.

Definition 8. *A binary tree is said to be **proper** if all nodes of the tree has 0 or 2 children. Otherwise, the binary tree is said to be **improper** which has some nodes with exactly one child.*

After dividing binary tree into two types, we wonder if there are some other ways to determine whether a binary tree is proper. After a while, we have discoverd the first method and borrowed some idea from code theory to find out the second method.

Theorem 9 provide a method about $|V|$ and $|L|$ that is straightforward and we believe there is someone else who has discoverd this relation.

Theorem 11 provide a method about $\sum_{l \in L} 2^{-h(l)}$. Note that Theorem 11 is first stated in the thesis of Kraft, Leon G.[7]. In his thesis, it is proved by counting the combinations of prefix codes therefore that proof can be

[5]They are not original terms, but this may be a good explanation for these terms.

applied to prefix code only. After that, McMillan extends the theorem to any uniquely decodable code[9]. And we find that the theorem can be applied to binary tree and the equality can be used to distinguish proper and improper tree. Although the theorem here is equivalent to Kraft's, but we give the inequality a new interpretation and usage. We will prove that in the manner of graph.

Theorem 9. *A binary tree is proper if and only if*

$$|V| = 2|L| - 1$$

Proof.

For $|V| = 1$, the only node has no children, therefore the tree is proper. Also because $|V| = 1 = 2|L| - 1$, the statement is true.

For $|V| > 1$, we divide the proof into two parts:

Only if part:

We know that for all graphs, $\sum_{n \in V} d(n) = 2|E|$. For all trees, $|E| = |V| - 1$. And for proper binary trees, the degree of a node is

$$d(n) = \begin{cases} 1, & n \in L \\ 2, & n = r \\ 3, & \text{otherwise} \end{cases}$$

By summing up the degrees of nodes in a proper tree, we have

$$\begin{aligned} \sum_{n \in V(T)} d(n) &= \sum_{n \in L} d(n) + d(r) + \sum_{n \in V \setminus (\{r\} \cup L)} d(n) \\ 2|E| &= |L| + 2 + 3(|V| - |L| - 1) \\ 2(|V| - 1) &= 3|V| - 2|L| - 1 \\ |V| &= 2|L| - 1 \end{aligned}$$

If part:

If a binary tree T is improper, there exists a parent node with exactly one child. Let m be the number of nodes with exactly one child. After we merge all these nodes with their children as Operation 4 on page 56 stated, we get a proper tree T' with same number of leaves and

$$|V(T')| = |V(T)| - m$$

Therefore,

$$|V(T)| > |V(T')| = 2|L(T')| - 1 = 2|L(T)| - 1$$

□

Corollary 10. *For all binary tree,*

$$|V| \geqslant 2|L| - 1$$

And equality holds if and only if the tree is proper.

Proof. From the proof of Theorem 9, if T is proper, $|V| = 2|L| - 1$. Otherwise, $|V| > 2|L| - 1$. □

Although this theorem is easy to use, we seldom deal with the number of nodes. Therefore, we have not used this in our research. Instead, our research is mainly dealing with the heights of leaves. And the next theorem is exactly for this purpose.

Theorem 11. *A binary tree is proper if and only if*

$$\sum_{l \in L} 2^{-h(l)} = 1$$

Proof.

Only if part:

Let $S(h)$ be a statement

$$\text{“For all proper tree } T \text{ with } h(T) = h, \sum_{l \in L(T)} 2^{-h_T(l)} = 1\text{”}$$

For $h = 0$, there is only root and its height is 0. So, we have

$$\sum_{l \in L(T)} 2^{-h(l)} = 2^{-0} = 1$$

Therefore, $S(0)$ is true.

Assume that $S(h)$ is true for all $0 \leqslant h \leqslant k$.

For $h = k + 1$, the root of proper binary tree T has two children a and b. For all leaves l_a and l_b which are in $L(T_a)$ and $L(T_b)$ respectively,

$$\begin{cases} h_{T_a}(l_a) = h_T(l_a) - 1 \\ h_{T_b}(l_b) = h_T(l_b) - 1 \end{cases}$$

Since the subtrees are proper and their heights are smaller than k, from induction hypothesis, $S(h(T_a))$ and $S(h(T_b))$ are true, that is

$$\sum_{l \in L(T_a)} 2^{-h_{T_a}(l)} = \sum_{l \in L(T_b)} 2^{-h_{T_b}(l)} = 1$$

From this, we can prove $S(k+1)$,

$$\begin{aligned} \sum_{l \in L(T)} 2^{-h_T(l)} &= \sum_{l \in L(T_a)} 2^{-h_T(l)} + \sum_{l \in L(T_b)} 2^{-h_T(l)} \\ &= \frac{1}{2} \sum_{l \in L(T_a)} 2^{-(h_{T_a}(l))} + \frac{1}{2} \sum_{L \in L(T_b)} 2^{-(h_{T_b}(l))} \\ &= 1 \end{aligned}$$

Since $S(k+1)$ is true, for all nonnegative integer h, $S(h)$ is true.

If part:

If a binary tree T is improper, there exists a parent node n which has only one child. After we merge all these nodes with their child as Operation 4 on page 56 stated, we can get a proper tree T' with unchanged number of leaves but the height of some leaves decrease, therefore

$$\sum_{l \in L(T)} 2^{-h(l)} < \sum_{l \in L(T')} 2^{-h(l)} = 1$$

□

Corollary 12. *For all binary tree,*

$$\sum_{l \in L} 2^{-h(l)} \leqslant 1$$

And equality holds if and only if the tree is proper.

Proof. From the proof of Theorem 11, if a binary tree is proper, we know that $\sum_{l \in L} 2^{-h(l)} = 1$. Otherwise, $\sum_{l \in L} 2^{-h(l)} < 1$. □

We realize that Theorem 11 is useful. For example, if we want to prove that a proper binary tree is perfect[6] if and only if $|L| = 2^h$, we can simply apply Theorem 11 to prove it.

Corollary 13. *A proper binary tree T is perfect if and only if*

$$|L| = 2^{h(T)}$$

Proof.

Only if part: Since $\sum_{l \in L} 2^{-h(l)} = 1$, $|L| = 2^{h(T)}$.

If part: If T is not perfect, there exists a leaf a with $h(a) < h(T)$. So, we have

$$1 = \sum_{l \in L} 2^{-h(l)} \geqslant (|L| - 1)2^{-h(T)} + 2^{-h(a)}$$

Therefore

$$|L| \leqslant 2^{h(T)} - 2^{h(T)-h(a)} + 1 < 2^{h(T)}$$

□

3. Weight-balanced Tree

Because there are two types of weighted trees, we need to declare which type of weighted tree we will use. In our research, only leaves are with assigned weights instead of nodes. Moveover, for future proofs, we will define the weight of a parent as the sum of weights of its descendant leaves. In this way, we can simplify our proofs and statements.

Definition 14. *An **L-tree**[7] is a tree that there is a function assigning each node a positive number which is called weights. The function is said to be a weight function, denotes as w_T, where T is the L-tree. Weight function is constrained that the weights of parent nodes equal to the sum of all weight of its descendant leaves, that is*

$$w_T : V(T) \to R^+, w_T(n) = \begin{cases} \textit{assigned weight of a leaf}, & \textit{if } n \in L(T) \\ \sum_{l \in L(T_n)} w_T(l), & \textit{otherwise} \end{cases}$$

Considering the applications of an L-tree, what we want to do is to minimize the cost, which is the weighted mean of heights of the leaves.

[6] A proper binary tree T is **perfect** if and only if for all leaf l, $h(l) = h(T)$.

[7] L stands for leaf-weighted.

Definition 15. *The weighted mean of heights of the leaves of an L-tree T is denoted by H_T or simply H. We denote that the sum of weights of all leaves as W_T or simply W, therefore*

$$H = \frac{\sum_{l \in L} w(l)h(l)}{W}$$

And the tree with the minimum H is what we want to generate.

Definition 16. *If an L-tree T has a minimum H among all T' with same weight function, we say that T is a* ***weight-balanced*** *tree. And the minimum H is denoted as H^*.*

Example 17. *Let w is a weight function where $w(a) = 1, w(b) = 2, w(c) = 3, w(d) = 4$. As $H_A = 1.9 < 2 = H_B$, B is not weight-balanced.*

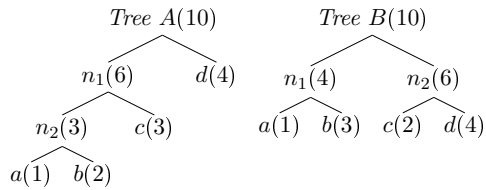


FIGURE 6

Actually, by exhaustive method, we deduced that $H_A = H^*$ and thus A is a weight-balanced tree.

As what we guessed, we have found that, a weight-balanced tree is proper, therefore the problems become easier to be solved.

Theorem 18. *A weight-balanced tree is proper.*

Proof. If an L-tree T is improper, there exists a node a with its only one child b. After we merge nodes a and b to produce L-tree T' without changing the weight function as Example 5 on page 57 stated, all the height of leaves of T_a decreases by 1 while others remain unchanged. Therefore $H_T > H_{T'}$ and T is not weight-balanced. □

3.1. Properties of Weight-balanced Tree

We found some common properties of all weight-balanced tree. These properties are useful for us to understand weight-balanced tree more and lead us

closer to the answer of the question, what a sufficient condition of weight-balanced tree can be. The first property we have found is about the relation between weight and height.

Theorem 19. *If a tree is weight-balanced, for all nodes*[8] *a and b, the following statements are true:*

1. $w(a) > w(b) \Rightarrow h(a) \leqslant h(b)$
2. $w(a) = w(b) \Rightarrow |h(a) - h(b)| \leqslant 1$

Proof. These two statements can be proved by swap.

(1): If a and b has an ancestor-descendant relationship, as $w(a) > w(b)$, node b is a descendant of a and $h(a) < h(b)$.

If a and b do not have an ancestor-descendant relationship, assume that $w(a) > w(b)$ and $h(a) > h(b)$. As Operation 2 on page 56 stated, we can produce another L-tree T' by swapping a and b without changing the weight function. Notice that

$$H_{T'} - H_T = \frac{(h(a) - h(b))(w(b) - w(a))}{W} < 0$$

It contradicts the assumption that the tree is weight-balanced.

(2): Assume that $w(a) = w(b)$ and $h(a) - h(b) \geqslant 2$. Because $w(a) = w(b)$, nodes a and b do not have an ancestor-descendant relationship.

As the tree has at least two nodes a and b, we know that $h(b) \geqslant 1$. Since $h(a) - h(b) \geqslant 2$, we know that $h(a) \geqslant 3$. Therefore, node b has a parent and the parent of node a has a parent, in other words, there exists $p_b = \mathrm{par}(b)$ and $p_{p_a} = \mathrm{par}(\mathrm{par}(a))$.

Then, we rearrange the nodes as following:

By considering the difference of H_T and $H_{T'}$,

$$\begin{aligned} H_{T'} - H_T &= \frac{1}{W}(w(b) - w(s_a) - (k-1)w(a)) \\ &= \frac{1}{W}(-w(s_a) - (k-2)w(a)) < 0 \end{aligned}$$

It contradicts the assumption that T is weight-balanced.

□

[8] Either parents or leaves.

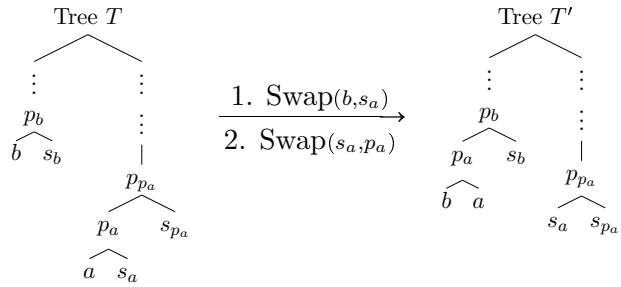


FIGURE 7

After we found out these relations, we wonder if there are any non-weight-balanced trees satisfies all these properties. However, the answer is yes. Tree B in Example 5 is one of the non-weight-balanced trees satisfies all these properties. Therefore, they are not sufficient conditions of weight-balanced tree.

3.2. Bound of H^*

The next property is the bound of H^* which we have seen in a book about information theory by David J.C. MacKay[8]. However, we have found another way to prove it. Although it cannot help us to find the weight-balanced tree, it give us an algorithm to create a nearly weight-balanced tree. Before the proof, we have to prove that we can construct a tree by giving a suitable sequence of heights of leaves.

Lemma 20. *Given a finite sequence of positive integers* $\{a_n\}$ *with*

$$\sum_k 2^{-a_k} \leqslant 1$$

there exists a binary tree that the finite sequence of heights of all leaves equals to $\{a_n\}$.

Proof. We will show the construction method in order to prove the binary tree exists.

For $\sum_k 2^{-a_k} = 1$, let h be the maximum integer in $\{a_n\}$. Let finite sequence $\{b_n\} = \{2^{h-a_n}\}$. Notice that

$$\sum_k b_k = 2^h \sum_k 2^{-a_k} = 2^h$$

Therefore, we can first construct a perfect binary tree with height h. From left to right, we label the leaves with k for b_k times. For example, if $\{a_n\} = \{1, 2, 3, 3\}$, we have $\{b_n\} = \{4, 2, 1, 1\}$ and we will label the tree as the following figure shown.

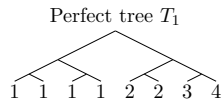


FIGURE 8. Label the leaves of the perfect tree.

Because all b_k is the power of two, we can put all label k into one subtree. Then, we can delete the whole subtree except the root of that subtree. Label the root of the subtree as k, and its height is exactly $h - \log_2(b_k) = a_k$. Using the same example, as all leaves of the left child of the root are labelled "1", after we perform the operation, the tree become this:

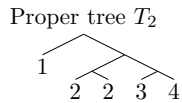


FIGURE 9. Perform an operation to leaves labelled 1.

After deleting all the specific nodes, we build up the binary tree we want.

For $\sum_k 2^{-a_k} < 1$, we extend the sequence. Let $D = 1 - \sum_k 2^{-a_k}$. By converting D from decimal to binary, we can easily find out a finite sequence $\{c_n\}$ that $\sum_k 2^{-c_k} = D$. Let L_a be the length of $\{a_n\}$, L_c be the length of $\{c_n\}$ and

$$d_k = \begin{cases} a_k, & \text{if } 1 \leqslant k \leqslant L_a \\ c_{(k-L_a)}, & \text{if } L_a + 1 \leqslant k \leqslant L_a + L_c \end{cases}$$

As $\sum_k 2^{-d_k} = 1$, using the above-metioned method, we can generate a binary tree. After deleting the node with height in $\{c_k\}$, we can generate the tree we want.

□

Theorem 21. *For all weight function w,*

$$-\sum_{l_k \in L(T)} \frac{w(l_k)}{W} \log_2 \frac{w(l_k)}{W} \leqslant H^* < 1 - \sum_{l_k \in L(T)} \frac{w(l_k)}{W} \log_2 \frac{w(l_k)}{W}$$

Proof. As any weight-balanced tree is a proper tree, from Theorem 11 on page 60, we know that $\sum_{l\in L} 2^{-h(l)} = 1$.

First, we wish to minimize H subject to the constraint $\sum_{l\in L} 2^{-h(l)} = 1$, but it is difficult to solve this problem since heights of leaves are integer. Therefore we ignore this constraint in order to find the lower bound of H.

Let $l_1, \ldots, l_n$ be the leaves of the tree, $p_k = w(l_k)/W$ and $\{h_k\}$ be any sequence of numbers satisfies $\sum_k 2^{-h_k} = 1$.

Let f and g be two functions that

$$\begin{cases} f(h_1, h_2, \ldots, h_n) = \sum_k p_k h_k \\ g(h_1, h_2, \ldots, h_n) = 1 - \sum_k 2^{-h_k} \end{cases}$$

Considering the partial derivatives of function g, we find that (cf. **Reviewer's Comment** 1)

$$\frac{\partial g}{\partial h_k} = \ln(\frac{1}{2})(\frac{1}{2})^{h_k} \neq 0$$

As $\nabla g \neq 0$, we can use the Lagrange multiplier to find the extrema of function f.

Let j be a function that

$$j(h_1, h_2, \ldots, h_n, \lambda) = \sum_k p_k h_k + \lambda(1 - \sum_k 2^{-h_k})$$

Set the derivative $\nabla j = 0$, which yields the system of equations:

$$\begin{aligned} \frac{\partial j}{\partial h_k} &= p_k - \lambda \log(\frac{1}{2})(\frac{1}{2})^{h_k} = 0 \\ \frac{\partial j}{\partial \lambda} &= 1 - \sum_k (\frac{1}{2})^{h_k} = 0 \end{aligned}$$

Combining the first two equations,

$$\begin{aligned} p_k &= \lambda \log(\frac{1}{2})(\frac{1}{2})^{h_k} \text{----------(*)} \\ \sum_k p_k &= \lambda \log(\frac{1}{2}) \sum_k (\frac{1}{2})^{h_k} \\ 1 &= \lambda \log(\frac{1}{2}) \\ \text{From(*), } p_k &= (\frac{1}{2})^{h_k} \\ h_k &= -\log_2(p_k) \end{aligned}$$

Then we are going to prove that point is the global minimum point.

Claim (1): H has no uppper bound.

By Considering the case when $h_1 = -\log_2(1-(n-1)2^{-M})$ and $h_2 = h_3 = \cdots = h_n = M$, we found that

$$H = \sum_k p_k h_k > p_n h_n = M p_n$$

As M is independent of p_n, there is no upper bound of H.

Claim (2): H has lower bound.

Since $\sum_k (\frac{1}{2})^{h_k} = 1$, we know that $h_k \geqslant 0$ for all k. Therefore $H \geqslant 0$. Therefore, lower bound of H exists.

Because H has lower bound and has not upper bound, at the only critical point $h_k = -\log_2(p_k)$, H attains a global minimum value, in other words,

$$-\sum_{l_k \in L} \frac{w(l_k)}{W} \log_2 \frac{w(l_k)}{W} \leqslant H^*$$

From Lemma 20 on page 65, since we can get a tree T' by setting $h(l_k) = \lceil -\log_2(p_k) \rceil$, where $\sum_{l_k \in L} 2^{-h(l_k)} \leqslant 1$,

$$H_{T'} = \sum_k p_k \lceil -\log_2(p_k) \rceil < \sum_k p_k (1 - \log_2(p_k)) = 1 - \sum_k p_k \log_2(p_k)$$

Therefore,

$$-\sum_{l_k \in L} \frac{w(l_k)}{W} \log_2 \frac{w(l_k)}{W} \leqslant H^* < 1 - \sum_{l_k \in L} \frac{w(l_k)}{W} \log_2 \frac{w(l_k)}{W}$$

□

By giving examples, we can understand the lower and upper bound of H^* more. Let T is a proper tree with exactly two leaves a and b. We know that $H^* = 1$. Suppose $w(a) = 1$ and $w(b) = m - 1$, and E is the lower bound of H^*.

$$E = -\frac{1}{m} \log_2(\frac{1}{m}) - \frac{m-1}{m} \log_2(\frac{m-1}{m}) = \log_2(m) - \frac{m-1}{m} \log_2(m-1)$$

When $m = 2$, $E = 1 = H^*$.

When $n \to 1$ (cf. **Reviewer's Comment** 2),

$$E = 0 - \lim_{m \to 1} \frac{m-1}{m} \log_2(m-1) = 0 = H^* - 1$$

Therefore, we can see the lower bound is achievable in some cases and H^* can near the upper bound no matter how small we want.

After proving the bound of H^*, we find that there is a method to create a binary tree within the bound of H^* just as the proof have mentioned. However, this is not our target, so we do not discuss this algorithm or the implement.

3.3. Special Trees

After we have proved some properties of weight-balanced tree, we still cannot find out a sufficient condition. Therefore, we try to ask a simple question first, when weight function is a constant function, how the trees with minimum and maximum H look like? We call them S-tree and H-tree.

Definition 22. *If a tree T has minimum $\sum_{l \in L} h(l)$ among all proper trees with the same number of leaves, that is for all binary tree T' where $|L(T')| = |L(T)|$,*

$$\sum_{l \in L(T)} h(l) \leqslant \sum_{l \in L(T')} h(l)$$

we say that T is an ***S-tree***[9].

Definition 23. *If a tree T has maximum $\sum_{l\in L} h(l)$ among all proper trees with the same number of leaves, that is for all binary tree T' where $|L(T')| = |L(T)|$,*

$$\sum_{l\in L(T)} h(l) \geqslant \sum_{l\in L(T')} h(l)$$

we say that T is an ***H-tree***[10].

After we have defined the tree want to investigate, we now try to find some equivalent definitions. In this way, we can find how these trees look like. Now, we investigate S-tree first.

Theorem 24. *A proper tree T is a S-tree if and only if for all leaf a and leaf b,*

$$|h(a) - h(b)| \leqslant 1$$

Proof.

Only if part:

Assume that there is a tree T with $\max h(l) - \min h(l) \geqslant 2$. Let a and b be a leaf with maximum height and a leaf with minimum height respectively. Now we perform swap on T to produce T' as the following figure shows.

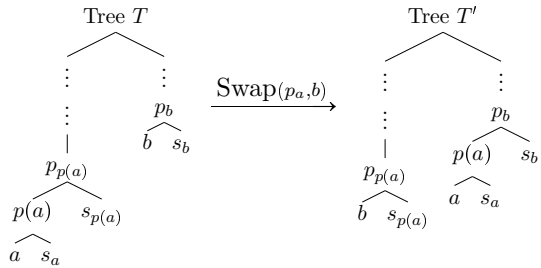


FIGURE 10

Let $k = \max h(l) - \min h(l)$. The sum of the height of all leaves of tree T' is

$$\sum_{l\in L} h_{T'}(l) = \sum_{l\in L} h_T(l) - |L(T_{s_a})|(k-1) < \sum_{l\in L} h_T(l)$$

Therefore T is not an S-tree.

[9]S stands for shortest, however, a tree T with minimum $h(T)$ is not necessarily a S-tree.
[10]H stands for the 'highest'

If part:

Let $h = h(T)$, $n = |L|$ and there are m leaves with height $h-1$. So, there are $n-m$ leaves with height h.

From Theorem 11 on page 60,

$$1 = m(\frac{1}{2})^{h-1} + (n-m)(\frac{1}{2})^{h}$$

$$m + n = 2^h$$

Using this, we can find that

$$\sum_{l \in L} h(l) = (h-1)m + h(n-m) = (h+1)n - 2^h$$

Also, As $0 \leqslant m < n$,

$$n \leqslant m + n = 2^h < 2n$$

Taking logarithm for both sides,

$$\log_2(n) \leqslant h < \log_2(n) + 1$$

$$h = \lceil \log_2(n) \rceil$$

Therefore,

$$\sum_{l \in L} h(l) = (\lceil \log_2(n) \rceil + 1)n - 2^{\lceil \log_2(n) \rceil}$$

For all given value of n, S-tree exists. From definition, an S-tree has minimum $\sum_{l \in L} h(l)$. And we know that, if a tree is an S-tree, the statement "for all leaf a and b, $|h(a) - h(b)| \leqslant 1$" is true. Also, for all proper trees, the trees with the statement is true have the same $\sum_{l \in L} h(l)$, therefore they are S-trees. □

Example 25. *Tree T is an S-tree, but T' is not because $|h_{T'}(a) - h_{T'}(b)| = 2$.*

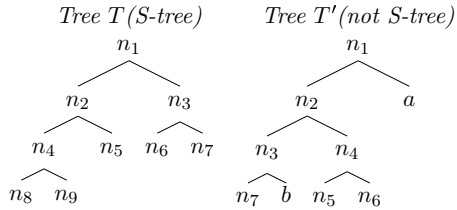


FIGURE 11. Figure of Example 5.

Theorem 26. *Let A_k and B_k be two sets of nodes that, A_k is the set of all parents with height k in the tree while B_k is the set of all leaves with height k in the tree, that is*

$$\begin{cases} A_k = \{n \in V \backslash L : h(n) = k\} \\ B_k = \{n \in L : h(n) = k\} \end{cases}$$

For all proper tree T with more than two nodes, these statements are equivalent:

1. *T is an H-tree*
2. $|A_k| = \begin{cases} 0, & \text{if } k = h(T) \\ 1, & \text{if } 0 \leqslant k \leqslant h(T) - 1 \end{cases}$
3. $|V(T) \backslash L(T)| = h(T)$
4. $|B_k| = \begin{cases} 0, & \text{if } k = 0 \\ 1, & \text{if } 1 \leqslant k \leqslant h(T) - 1 \\ 2, & \text{if } k = h(T) \end{cases}$

Proof. For $k = h(T)$, as there is no nodes with height $h(T) + 1$, there is no parents with $h(T)$, that is $|A_k| = 0$.

For $0 \leqslant k \leqslant h(T) - 1$, by considering the path from a leaf l with $h(l) = h(T)$, we know there exists a parent n where $h(n) = k$. Therefore $|A_k| \neq 0$.

(1)⇒(2):

For $0 \leqslant k \leqslant h(T) - 1$, if $|A_k| \geqslant 2$, there exists two parents a and b where $h(a) = h(b) = k$. Let c be a descendant leaf of a where $h(c) \geqslant h(a)$. After swapping nodes b and c, we get another tree T' with the same number of leaves as Operation 2 on page 56 stated. Let $K = h(c) - h(b) > 0$, considering the difference between the two trees, we have

$$\sum_{l \in L(T')} h_{T'}(l) - \sum_{l \in L(T)} h_T(l) = K(|L(T_b)| - 1)$$

As T is proper and b is a parent node, $|L(T_b)| > 1$, T is not a H-tree.

(2)⇒(3):

Directly counting the number of parents from (2),

$$|V\backslash L| = 1 \times h(T) + 0 \times 1 = h(T)$$

(3)⇒(2):

It is proved that $|A_{h(T)}| = 0$ and for $1 \leqslant k \leqslant h-1$, $|A_k| \neq 0$. If there exists $k \geqslant 1, |A_k| \geqslant 2$. Therefore, we have

$$|V\backslash L| \geqslant 2 \times 1 + 1 \times (h(T) - 1) + 0 \times 1 > h(T)$$

(2)⇒(4):

For $k = 0$, as $|V| \geqslant 2$ and the root is the only node with height zero, the root is a parent and $|B_0| = 0$. For $1 \leqslant k \leqslant h(T)$, as $|A_{k-1}| = 1$, there are two nodes with height k. As $|B_k| = 1$, $|A_k| = 1$. Also as $|A_{h(T)}| = 0$, $|B_{h(T)}| = 2$.

(4)⇒(1):

Summing up the height of leaves according to (4),

$$\sum_{l \in L} h(l) = 1 + 2 + \cdots + (h(T) - 1) + 2 \times h(T) = \frac{1}{2} h(T)(h(T) + 3)$$

Also, we know that the relation of L and h is as follows:

$$|L| = 1 \times (h(T) - 1) + 2 \times 1 = h(T) + 1$$

Combining the two equation, we have

$$\sum_{l \in L} h(l) = \frac{1}{2}(|L| - 1)(|L| + 2)$$

For all given value of L, H-tree exists. From definition, an H-tree has maximum $\sum_{l \in L} h(l)$. And we know that, if a tree is an H-tree, statement (4) is true. Also, for all proper trees, if statement (4) is true, they have the same $\sum_{l \in L} h(l)$. Therefore all proper trees with (4) is true are H-trees. □

3.4. Weight Function

After we have found how S-tree and H-tree look like, we want to investigate S-tree and H-tree more to see under what sorts of weight functions, they are weight-balanced. We hope that investigating this problem will give us a sufficient condition.

For all S-tree, it is easy to know that if all the leaves are with equal weight, it is weight-balanced. But, there are much more sorts of weight function satisfies this condition. However, they cannot be easily found before we find a sufficient condition and they seem to be tedious. Therefore we do not handle it.

For all H-tree, the answer is a simple inequality. It is quite beautiful and we have found a way towards a sufficient condition.

Theorem 27. *Let T is an L-tree. $l_1, l_2, \ldots, l_{|L|}$ be the leaves of T.*

We said a weight function is an H-function if for all $1 \leqslant k \leqslant |L| - 2$

$$\sum_{i=1}^{k} w(l_i) \leqslant w(l_{k+2})$$

We can construct a weight-balanced H-tree if and only if the weight function is an H-function.

Proof.

Only if part:

From Thereom 26 on page 72, for all $0 \leqslant i \leqslant h - 1$, there is exactly one parent and exactly one leaf with height i. Also, there are two leaves with height $h(T)$, just as the following figure shows.

By considering leaf l_i and node n_{i+1}, for $1 \leqslant i \leqslant h(T) - 2$, we have

$$h(n_{i+1}) > h(l_i)$$

From Theorem 19 on page 64,

$$w(n_{i+1}) \leqslant w(l_i)$$

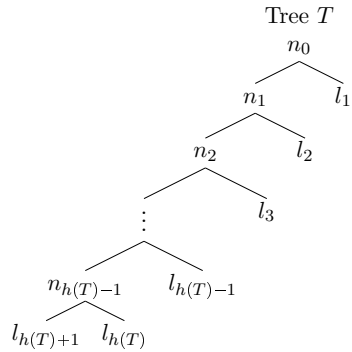


FIGURE 12

As the weight of parent node equals to the sum of all weight of its descendant leaves and leaf l_j is a decendant of node n_i for all $j > i$

$$\sum_{j=h(T)+1}^{i+2} w(l_j) \leqslant w(l_i)$$

If we alter the label of leaf from k to $h(T) + 2 - k$, we have

$$\sum_{j=1}^{h(T)-i} w(l_j) \leqslant w(l_{h(T)-i+2})$$

If part:

Assume that weight-balanced tree T with H-function is not a H-tree.

From Theorem 26 on page 72, there exists i where there are more than two leaves with height i, that is

$$|\{l : h(l) = i\}| \geqslant 3$$

Consider the largest i. Let the three leaves with largest weights among the leaves with height i be l_1, l_2 and l_3, and their weights are w_1, w_2 and w_3 respectively such that $w_1 \leqslant w_2 \leqslant w_3$.

We can put these leaves under the same parent of parent by swapping without changing their heights as the following figure shows.

If we swap l_3 and p_2 as Operation 2 on page 56 stated, we can produce T' as following figure shows.

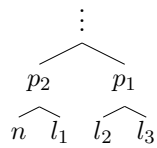


FIGURE 13

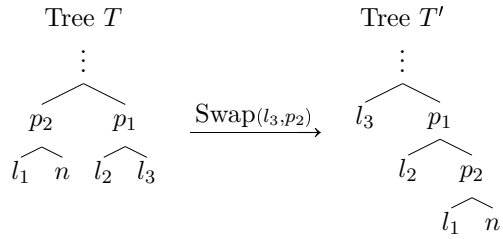


FIGURE 14

Notice that, from the inequality the difference of H between two trees T' and T is

$$H_{T'} - H_T = \left(\sum_{l \in L(T_n)} w(l) + w_1 \right) - w_3 \leqslant 0$$

By repeating this procedure, we can reduce the number of height i, which there are at least three leaves with height i. Ultimately, we can obtain a H-tree $T^\sharp$ such that $H_{T^\sharp} \leqslant H_T = H^*$. Therefore, $T^\sharp$ is weight-balanced. □

After finding this interesting inequality, we find something about weight-balanced tree which is related to golden ratio.

Corollary 28. *Suppose an L-tree has more than three leaves and the ordered sequence of the weights of leaves is geometric. We can construct a weight-balanced H-tree if the ratio is greater than the golden ratio, that is* $\frac{\sqrt{5}+1}{2}$.

Proof. Let $w(l_i) = ar^{i-1}$.

$$\begin{aligned} r &\geqslant \frac{\sqrt{5}+1}{2} \\ r^2 - r - 1 &\geqslant 0 > -1 \\ \frac{a(r^k-1)}{r-1} &\leqslant ar^{k+1} \\ \sum_{i=1}^{k} w(l_i) &\leqslant w(l_{k+2}) \end{aligned}$$

Therefore the weight function is also an H-function. From Theorem 27 on page 74, we can construct a weight-balanced H-tree. □

However, other types of geometric weight functions are not our focus. Also, we find that someone has found a general weight-balanced tree for geometric weight functions[6].

3.5. A Sufficient Condition

In this subsection, we will introduce a visual sufficient condition of being a weight-balanced tree. Before that, we will introduce a labeling method that is directly related to the condition.

Definition 29. *We can label a tree from left to right and from top to bottom like this:*

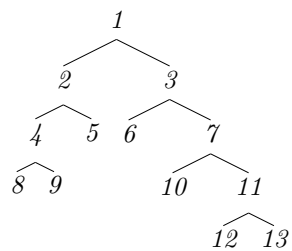


FIGURE 15

This label method is called ***natural labeling****. We can name the nodes and weights by their labels. For example, on the above figure,* n_1 *is the root,* n_3 *is a child of* n_1*, and* $w(n_1) = w_1$*.*

In the investigation of H-tree, we find that the weights of nodes in a weight-balanced tree is ordered under natural labeling after swapping the nodes with same height. And we find that it is the sufficient condition for our research.

Theorem 30. *Under natural labeling, L-tree T is weight-balanced if*

$$w(n_1) \geqslant w(n_2) \geqslant \cdots \geqslant w(n_{|L|})$$

Proof.

In this proof, we call $w(n_1) \geqslant w(n_2) \geqslant \cdots \geqslant w(n_{|L|})$ as Inequality (*). Now, we use induction on the number of leaves m.

For $m = 1$, L-tree is weight-balanced.

For $m \geqslant 2$, assume that the statement holds for all proper tree with $m - 1$ leaves. Consider T_1 is a weight-balanced tree and T_2 is a proper L-tree with the same weight function that satisfies Inequality (*). By swapping the leaves with height of $h(T_1)$, we can put the two leaves with smallest weight under the same parent node without changing H_{T_1}.

Let the least two weights be w_1 and w_2. If we remove the two leaves from T_1 and T_2 to produce T_1' and T_2' , the parent of the leaves will become a leaf. Set the weight of the parent to $w_1 + w_2$ as following figure shows.

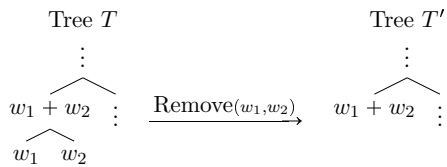


FIGURE 16. Weights are shown in the labels

The value of H will become:

$$\begin{cases} H_{T_1'} = H_{T_1} - (w_1 + w_2) \\ H_{T_2'} = H_{T_2} - (w_1 + w_2) \end{cases}$$

As Inequality (*) is still true for tree T_1' and T_2' and $|L(T_1')| = |L(T_2')| = m-1$, from the induction hypothesis, T_1' is a weight-balanced tree. Therefore, $H_{T_1'} \leqslant H_{T_2'}$ and $H_{T_1} \leqslant H_{T_2} = H^*$.

Hence, T_1 is a H-tree. (cf. **Reviewer's Comment** 3) □

Although this condition seems to be difficult to use, actually as we can rearrange the nodes with the same height without changing H_T, if an L-tree is not weight-balance, we cannot rearrange from left to right in descending order of weights for every height of leaves. Therefore, we can recognize the non-weight-balanced tree by this condition. Furthermore, this condition can be used to prove an algorithm that can generate weight-balanced tree and to help design another algorithm.

3.6. Algorithms

If we do not alow to swap the order of node, our condition is not necessary because the weights on same level can be in wrong order. However, if we allow to swap the order, it is sufficient by using Huffman Algorithm. In this section, we will discuss the algorithms that generate a weight-balanced tree. First we will prove the Huffman tree is valid by using our sufficient condition. Although our proof maybe longer than some widely known proof of this problem, it is a good way to show how to use the sufficient condition. After that we will introduce an idea of another algorithm, which may solve some problems that Huffman coding cannot.

Definition 31. *A Huffman tree is a tree generated by the optimum binary coding procedure.*[2] *(cf.* ***Reviewer's Comment*** *4)*

Corollary 32. *Huffman tree is weight-balanced.*

Proof. Assume that there is a Huffman tree T that is not weight-balanced. And we will use natural labeling method to label the nodes and its weights. By swapping, we can rearrange the node without changing H_T such that for all nodes with same height, if label $i < j$, $w_i \geqslant w_j$. From Theorem 30 there exists $b > a$ such that

$$w_1 \geqslant w_2 \geqslant \cdots \geqslant w_{a-1} \geqslant w_b > w_a$$

As for all node n_k, $w_1 \geqslant w_k$, the node n_a is not the root and $a \neq 1$. Denote $\mathrm{lab}(n)$ as the label of the node. By considering the labeling method, we know that $\mathrm{lab}(\mathrm{par}(n_a)) \leqslant \mathrm{lab}(\mathrm{par}(n_b))$ because $a < b$.

If $\mathrm{lab}(\mathrm{par}(n_a)) = \mathrm{lab}(\mathrm{par}(n_b))$, nodes n_a and n_b are siblings. Therefore, $h(n_a) = h(n_b)$. From the assumption of rearrangment, $b > a$ and $w_a \geqslant w_b$ which is a contradiction to the assumption $w_b \geqslant w_a$. Thus $\mathrm{lab}(\mathrm{par}(n_a)) < \mathrm{lab}(\mathrm{par}(n_b))$ and $w(\mathrm{par}(n_a)) \geqslant w(\mathrm{par}(n_b))$.

As nodes n_a and n_b are not siblings and the tree is proper, therefore, there exists nodes s_a and s_b are siblings of n_a and n_b respectively as following figure shows.

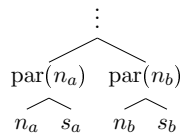


FIGURE 17. We have assumed $w_b > w_a$ and $w(\text{par}(n_a)) \geqslant w(\text{par}(n_b))$

As each step of the Algorithm, two subtrees with smallest weights will be put together to become one subtree, therefore

$$w(\text{par}(n_b)) = w(s_b) + w(n_b) > w(s_a) + w(n_a) = w(\text{par}(n_a))$$

It is a contradiction.

□

The method of Huffman's coding can construct a weight-balanced tree in $O(n \log n)$ time. Besides, because the weights of leaves are ordered in a weight-balanced tree, while inserting or deleting a leaf, we can update a weight-balanced tree in $O(n)$ time, where n is the number of leaves. However, sometimes we may need to construct or update a weight-balanced tree progressively. As we may want to terminate the algorithm in order to generate a lower H in less time as a trade-off, we want to design an algorithm for this purpose. Here is the general idea of our algorithm to optimize an L-tree:

Algorithm 33. *(Optimization by swap)*

1. [***Find***] *Under natural labeling, find any pair of nodes n_i and n_j, where $i > j$ and $w_i < w_j$. If we cannot find them, terminate.*
2. [***Swap***] *Swap nodes n_i and n_j.*
3. [***Repeat***] *Go to (1).*

When the algorithm terminates, as we cannot find any pair of specific nodes, therefore $w_1 \leqslant w_2 \leqslant \ldots \leqslant w_{|V|}$. By Theorem 30, we know the tree is weight-balanced.

However, the algorithm is not workable yet, because there are some problems we have not solved due to insufficient time. First, how to locate the pairs effectively? Second, does this algorithm end in all cases? We have a

conjecture that the total number of swap is smaller than $|V|^2$ in all cases. Third, after we locate the pairs, which pair should we swap first? These are open questions.

4. Applications

4.1. Prefix Code

When we talk about code, we are discussing how to represent an information from a set of information. Usually, we would like to reduce the transmission cost of the set of information.

We call a code **prefix** if any code word is not a prefix of another code word. The advantage is that we can identify each code word from the message easily. Just like other code, a prefix code is usually presented as a finite sequence of bits. We can draw a prefix code out as a binary tree. For example $\{0, 10, 110, 111\}$ is a prefix code. If we draw the code out, we will the next figure. Each leaf would present a piece of information, and the code word of each information would be the path from the roots to the leaves. For instance $\{0, 01, 10, 11\}$ is not a prefix code because 0 is not a leaf.

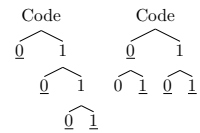


FIGURE 18. The left is prefix code and the right is not.

Suppose every piece of information would have certain probability of being transmitted. If we define the weight of a leaf as equal to the probability of the information being transmitted, the expected code length is equal to H. Therefore finding a weight-balanced tree is exactly finding the optimal code scheme.

4.2. Binary Operation

Also, a question of Hong Kong Olympiad in informatics 2002 [10] inspired us the third application.

Suppose there is a factory building a machine. The cost of building each subpart n is $C(n)$ and the cost connecting subpart a and b is $C(a) + C(b)$.

Then, we want to know how to connect all subpart together with the minimum cost. We denote the cost of connecting a and b as

$$C(a \oplus b) = C(a) + C(b)$$

Example 34. *The figure represent the process of* $(a \oplus b) \oplus (c \oplus d)$.

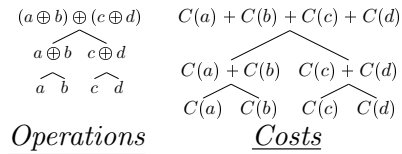


FIGURE 19

First, we need to find out the relation between H and the total cost of connecting different subparts.

Proposition 35.

$$\sum_{l \in L} w(l)h(l) = \sum_{n \in N \setminus \{r\}} w(n)$$

Proof. From the definition of weight function w,

$$\sum_{n \in V} w(n) = \sum_{n \in V} \left(\sum_{l \in L(T_n)} w(l) \right)$$

Since each leaf l is a descendant of $h(l)$ different leaves, each leaf appears in the expansion exactly $h(l) + 1$ times. Therefore we have,

$$\begin{aligned} \sum_{n \in V} w(n) &= \sum_{l \in L} w(l)(h(l) + 1) \\ \sum_{n \in V \setminus \{r\}} w(n) &= \sum_{l \in L} w(l)(h(l) + 1) - \sum_{l \in L} w(l) \\ &= \sum_{l \in L} w(l)h(l) \end{aligned}$$

□

Let C is the total cost of connecting and building subparts.We have,

$$C = \sum_{n \in V} w(n) = \sum_{l \in L} w(l)h(l) + W = W(H + 1)$$

As W is a constant, if we find the minimum H, the minimum C is also found.

Example 36. *We have n_1, n_2, n_3, n_4 and their costs are respectively $1, 2, 3, 4$. By Huffman's algorithm, we can generate a weight-balanced tree like this:*

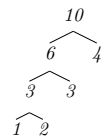


FIGURE 20

Hence, in order to acheive the minimum total cost, the operating process should be like this:

$$((n_1 \oplus n_2) \oplus n_3) \oplus n_4$$

4.3. Search

After reading about searching an ordered table in The Art of Computer Programming Volumn 3 [3], it inspired us to think a new application. Assume that we have ordered sequence of values and the probability of searching for the values are known. If we want to locate a particular value in the list, we commonly use "linear search" or "binary search". However, as they do not consider the probability of searching for a particular value, they are not optimal in all cases.

Therefore we find that searching in ordered sequence may be also an application for our research. Given that we have an ordered sequence with the probability which is monotonic with the value. We have found another algorithm that can optimize the searching process.

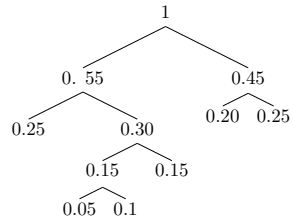
Algorithm 37. *(Building up a binary search tree)*

1. [***Build up a weight-balanced tree***] *Using probability of searching values to bulid up a weight-balanced tree.*
2. [***Swap***] *Swap the nodes with same height to put node n that have lower $\max_{l \in L(T_n)} w(l)$ on lefter.*
3. [***Label parents***] *Label each parent node by the largest value*[11] *of the leaves of the left subtree.*

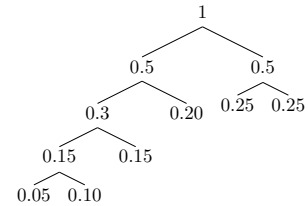
[11] Here, we label the largest for probabilities increasing with the values and the smallest value for probabilities decreasing with the values.

Example 38. *The ordered sequence is* $\{1, 2, 3, 4, 5, 5\}$ *and the probabilities are* $\{0.05, 0.1, 0.15, 0.2, 0.25, 0.25\}$ *respectively.*

(1) [**Build up a weight-balanced tree**] (weights are shown)



(2) [**Swap**] (weights are shown)



(3) [**Label parents**] (values are shown)

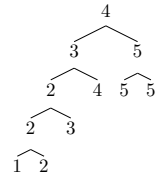


FIGURE 21

After creating the search tree, we can locate a value by the tree.

Example 39. *Use the previous example. When we search for value 2, as the leaves with the biggest value on the left subtree is smaller than the value of the parent, the path is:* $4 \to 3 \to 2 \to 2 \to 2$.

Although this searching algorithm uses double memory comparing to using binary search tree, we guess this algorithm may have some special uses, such as analyzing which searching method is better. For example, if the ratio of successive probabilities of values greater or equal to $\frac{\sqrt{5}+1}{2}$, we can find that the generated search tree is an H-tree and searching the ordered sequence simply by linear search is better than binary search.

5. Conclusion

The objective of this paper is achieved, as we have found a sufficient condition of a weight balance tree(Theorem 30 on page 78).

Besides the objective, mainly we have these by-products:

1. The bound of H of weight-balanced tree. (Theorem 21 on page 66)
2. The relationship between heights and weights of nodes in a weight-balanced tree. (Theorem 19 on page 64)
3. A weight-balanced tree with H-function is an H-tree. (Theorem 27 on page 74)
4. For geomatric weight function with the ratio greater than the golden ratio, weight-balanced H-tree exists. (Corollary 28 on page 76)
5. Huffman's algorithm is valid. (Corollary 32 on page 79)

For the proposed swap algorithm, there are three open questions:

1. How to effectively locate the pair that the labels $i > j$ but the weights $w_i < w_j$ under natural labeling?
2. Does the algorithm end in all case? If yes, how many times we need to swap in order to generate a weight-balanced tree?
3. Which pair should be swap first?

Answering these questions helps to create an effective algorithm which may allow us to have a halfway termination with valid output.

6. Acknowledgments

The authors would like to express their thanks to Prof. CHUNG Tsz Shun of The Chinese University of Hong Kong for his helpful suggestions.

REFERENCES

[1] C.E. Shannon, A Mathematical Theory of Communication. Bell System Technical Journal, vol. 27, pp. 379-423, 623-656, 1948.

[2] David A. Huffman, Huffman, A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the Institute of Radio Engineers, 40(9):1098-1101, 1952.

[3] Donald E. Knuth, Sorting and Searching, The Art of Computer Programming, Volume 3, Second Edition. Addison Wesley, 1998.

[4] Donald E. Knuth, Tracy Larrabee and Paul M. Roberts, Mathematical Writing. The Knowledge Engineering Review, Volume 12, Issue 3:331–334, (http://portal.acm.org/citation.cfm?id=976246.976258), 1989.
[5] Douglas B. West, Introduction to Graph Theory. Prentice Hall, 1996.
[6] Golomb, S.W., Run-length encodings. IEEE Transactions on Information Theory, IT–12(3):399–401, 1966.
[7] Kraft, Leon G, A device for quantizing, grouping, and coding amplitude modulated pulses. Cambridge, MA:MS Thesis, Electrical Engineering Department, MIT, (http://dspace.mit.edu/handle/1721.1/12390), 1949.
[8] MacKay, D. J. C., Information Theory, Inference, and Learning Algorithms. Cambridge University Press, 2003.
[9] McMillan, Brockway, Two inequalities implied by unique decipherability. IEEE Trans. Information Theory 2(4): 115!V116,
(http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1056818), 1956.
[10] The HKOI Organizing Committee, Question 2 Addition, Past Paper of HKOI2002 Final Event Senior Group. (http://www.hkoi.org/ref/2002fse.doc), 2002.

Reviewer's Comments

1. On page 13, line 1, $\frac{\partial g}{\partial h_k} = \ln\left(\frac{1}{2}\right)\left(\frac{1}{2}\right)^{h_k}$ should be $\frac{\partial g}{\partial h_k} = \ln 2\left(\frac{1}{2}\right)^{h_k}$. Subsequent equalities are okay. All the log should be $\log_2$.

2. On page 14, line 14, "When $n \to 1$" should be "When $m \to 1$".

3. At the end of the proof of Theorem 30, "T_1 is a H-tree" should be "T_1 is a weight-balanced tree".

4. On page 23, originally there is no definition of Huffman tree. To the reviewer, the explanation of "prefix code" is not clear. It is hard to understand what a prefix code is after reading the paragraph.

Final Words: Congratulations to the authors and the supervisor for such a good paper. It is good to see that mathematics is flourishing in Hong Kong.