Efficient algorithms for updating Huffman codes

Yasser El-Sonbaty and Nahla Belal
College of Computing and Information Technology, Arab Academy for Science and Technology, P.O. BOX 1029 Alexandria, Egypt

Given a list $W = [w_1,..., w_n]$ of n positive integer symbol weights, and a list $L = [l_1,...,l_n]$ of n corresponding integer codeword lengths, it is required to find the new list L when a new value x is inserted in or when an existing value is deleted from the list of weights W. The presented algorithm uses the given information about the weights and their corresponding levels in order to perform the required update. No other knowledge about the original Huffman tree is assumed to be known. Instead of rebuilding the Huffman tree, the new algorithm redistributes the weights among the levels to obtain the new Huffman code. In many special cases, the updated Huffman code can be generated with lower complexity than reconstructing the Huffman tree from scratch by efficiently using the information of weights and their levels. In this paper, we present an updating algorithm that requires a linear complexity in many practical cases rather than the $O(n \log n)$ needed for reconstructing the Huffman tree. We also give a practical $O(n \log n)$ implementation for our algorithms.

باستخدام قائمة الأوزان الموجبة الصحيحة $W = [w_1, w_2, ..., w_n]$، و المقابل لها من أطوال الأكواد الموجبة الصحيحة $L = [l_1, l_2, ..., l_n]$، المطلوب إيجاد قائمة أطوال الأكواد الجديدة عند إضافة أو حذف قيمة من أو إلى القائمة. الخوارزم المقدم يستخدم فقط المعلومات الموجودة عن الأوزان و أطوال الأكواد لتنفيذ التغييرات المطلوبة. فبدلاً من إعادة بناء شجرة الهوفمان، يعيد الخوارزم المقترح توزيع الأوزان على المستويات المختلفة للحصول على الكود الجديد الهوفمان. الخوارزم الجديد يحتاج إلي درجة تعقيد خطية في حالات كثيرة لتعديل شجرة الهوفمان، و قدمنا أيضاً خوارزم بدرجة تعقيد $O(n \log n)$ لتعديل أكواد الهوفمان.

Keywords: Huffman codes, Updating algorithms, Level consistency, Weight consistency, Complexity analysis

1. Introduction

Huffman coding [1] is a well known code tree problem, which encodes symbols according to their probabilities in order to minimize the expected codeword length. Given a list $W = [w_1,..., w_n]$ of n positive symbol weights, Huffman codes are constructed to determine a list $L = [l_1,...,l_n]$ of n corresponding integer codeword lengths, such that $\sum_{i=1}^{n} 2^{-l_i} \le 1$ and $\sum_{i=1}^{n} w_i l_i$ is minimized. Huffman coding plays an important role in data compression and other applications [2, 3, 4].

The classical algorithm described by Huffman [1], constructs the Huffman code in $O(n \log n)$ time. Van Leeuwen has shown that if elements are sorted according to their weights, a Huffman code can be constructed in $O(n)$ time using two queues [5].

Another approximation technique for Huffman codes is dynamic Huffman coding [6, 7, 8, 9], which saves the first pass taken to find the frequencies of occurrence of symbols and constructs a time varying tree at both the sender and receiver sides. The algorithm starts by building the Huffman tree for the first t symbols, and resumes by either incrementing the weight of an already existing symbol by one or adding a new symbol of weight one. The process continues until the end of the message to be encoded. The first algorithm in dynamic Huffman coding was FGK algorithm [6, 7, 8]. Vitter [9] introduced another one pass algorithm to produce shorter encodings than those produced by FGK algorithm.

In this paper, we give an insertion algorithm and a deletion algorithm for updating Huffman codes. Given a list $W = [w_1,..., w_n]$ of n positive integer symbol weights, and a list $L = [l_1,...,l_n]$ of n corresponding integer codeword lengths, it is required to find the new list L when a new value x is inserted in the list of weights W and when an already existing value is deleted from the list. The input to our algorithm is just the

weights and their corresponding levels, no other information about the Huffman tree is known.

The standard approach for updating Huffman codes is by rebuilding the tree after insertion or deletion of a certain weight. This approach disregards the information already available about each weight and its corresponding codeword length, or level, which in many cases speeds up the update process, resulting in a linear-time update. For example, consider the trivial case where the new weight to be inserted is greater than or equal to the value of the root of the corresponding Huffman tree, i.e. the new node is of a value greater than or equal to the sum of all n weights given, in this case the update to be done simply requires incrementing the codeword length of each weight by 1, with the new weight getting a codeword of length 1. And for the case of deletion, if the deleted node is deleted from the lowest level, the one closest to the root**,** the new code is obtained by simply moving the largest node in this level, with its sub-trees, one level closer to the root.

Similar cases to the previously mentioned ones were a motivation to generalize the updating algorithms and achieve better complexity than $O(n \log n)$.

This paper is organized as follows. In the next section we introduce theorems and definitions that will be needed to prove the correctness of the given algorithms. In Section 3, the insertion and deletion algorithms are discussed. Section 4 presents the complexity analysis of the algorithms given in Section 3. In Section 5, we illustrate the presented algorithms by examples. Finally conclusions are discussed in Section 6.

2. Properties of Huffman codes

In this section, we present some new definitions and obtain several results that will help in proving the correctness of the algorithms presented in Section 3.

We start with the following property regarding the levels of a Huffman tree.

2.1. Definitions

2.1.1. The exclusion property [10]

In a Huffman code the weights of the nodes (leaves and internal nodes) at level L are not smaller than the weights of the nodes at level $L+1$.

An implication of this property is that in any level L of a Huffman tree, the sum of the smallest two nodes is not less than the largest node in the level.

In light of the exclusion property, we present the following definitions and theorems,

2.1.2. Weight consistency

A level L is weight consistent if the sum of the smallest two nodes is not less than the largest node in the same level and the number of nodes in level L is even, except for the lowest level containing leaves, the number of nodes must be a power of two.

2.1.3. Level consistency

A level L is level consistent if no node in level $L-1$ has a value less than a node in level L.

2.1.4. Tree consistency

A tree is consistent if all of its levels are level consistent and the number of nodes in each level is even, except for the lowest level containing leaves which must have a number of nodes that is a power of two.

It is clear that a Huffman tree is consistent in that sense.

2.2. Theorems

Theorem 1

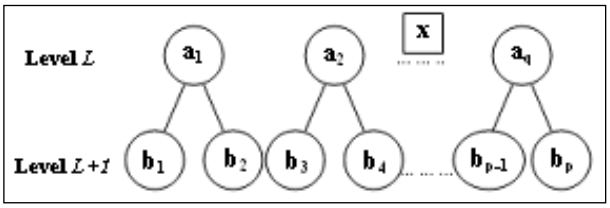
Given a consistent tree, and a new node is added to level L, if no node in level $L-1$ has a value less than the sum of the smallest two nodes in level L, possibly including the new node, then the smallest two nodes in level L can be moved to level $L+1$, and level L and all higher levels are all now weight and level consistent.

Proof

First, we need to show that the number of nodes in levels L and $L+1$ become even. For level L, there is an extra node x, and moving the smallest two nodes to level $L+1$ results in a

new internal node in level *L* which makes it contain an even number of nodes. As for level *L+1*, two nodes will be added to an originally even number, which results in a new even number. Second, we need to prove that the sum of the smallest two nodes in each level higher than *L-1* remains not greater than the largest node in the same level.

Case 1: If the smallest two nodes in level L are internal node.(Internal nodes are represented by◯ circles □ and leaves are represented by squares)

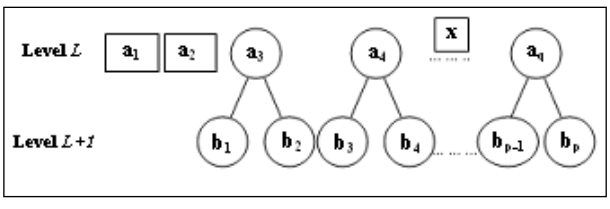


Moving the smallest two nodes, a_1 and a_2, from level *L* to level *L+1*, results in shifting the sub-trees of both nodes a_1 and a_2 to the next higher level. This will not affect the parity of the number of nodes in any level, since we will always be adding some even number of nodes to each level and the total number will thus remain even.

As for level consistency, it is clear that since we are always moving the smallest nodes from one level to the next higher level, these two levels will remain level consistent. This applies to level *L+1* and all higher levels. The level consistency of level *L* is guaranteed since the largest node in level *L* is now a_1+a_2 and no smaller node in level *L-1* exists.

It is clear that level consistency is a sufficient condition for the first condition of weight consistency, because if no node in a level *L-1* has a value that is less than the largest node in level *L*, then it is guaranteed that the sum of the smallest two nodes in level *L*, which forms an internal node in level *L-1*, is not less than the largest node in level *L*.

Case 2: If the smallest two nodes in level *L* are external nodes, possibly including the new node.



Moving the smallest two nodes, a_1 and a_2, from level *L* to level *L+1*, makes a_1 and a_2 the largest two nodes in level *L+1* forming the largest node in level *L*. This case is proved similarly as Case 1, but without propagation to higher levels, as a_1 and a_2 do not have sub-trees.

Case 3: If the smallest two nodes in level L are one external node, possibly the new node, and an internal node.

This case is the same as the previous two cases, with one of the smallest two nodes having a sub-tree.

Lemma 1.1

If level *L*, where the insertion is made**,** is equal to or lower than the lowest level containing leaves, then moving the smallest two nodes in *L* to level *L+1*, keeps the tree consistent.

Proof

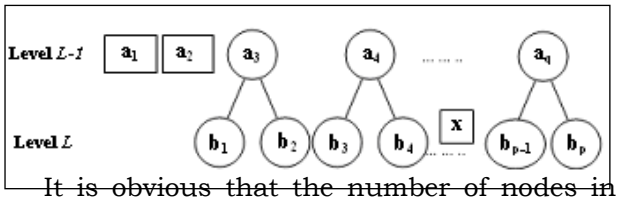
The nodes in levels lower than *L* are all combinations of nodes in level *L*, since there are no leaves in any of these levels. Therefore, the level consistency propagates to the rest of the levels, and this implies weight consistency as well, resulting in a consistent tree.

The next theorem takes care of the case when the condition in Theorem 1 does not hold.

Theorem 2

Given a consistent tree, and a new node is added to level *L*, if the smallest leaf in level *L-1* has a value less than the sum of the smallest two nodes, possibly including the new node, in level *L*, then moving the smallest leaf in level *L-1* to level *L* keeps the weight and level consistency properties holding for level *L* and all higher levels.

Proof



It is obvious that the number of nodes in both levels *L* and *L-1* will be even. Level *L*

contained an odd number of nodes, after the insertion of **x,** so moving the smallest leaf from level *L-1* to level *L* adjusts the number of nodes in level *L* to an even number. As for level *L-1*, a leaf was removed, but it will combine with the largest node in level *L* to form an internal node in level *L-1* resulting in an even number of nodes. The second weight consistency condition, which states that the sum of the smallest two nodes in any level must not be less than the largest node in the same level, is guaranteed because it is known that the smallest leaf moved from level *L-1*, a_1, to level *L* has a value less than the sum of the smallest two nodes in level *L*, b_1+b_2. In Level *L-1*, the leaf that was moved to level *L* is a_1. We know that $a_1+a_2 \geq a_q$, and also $a_2+a_3 \geq a_q$. However, a new node is now added to level *L-1* resulting from the combination of a_1 with the largest node in level *L*, b_p, but all nodes in level *L-1* are not less than a_1 or b_p, hence, $a_2 \geq a_1$, $a_2 \geq b_q$, $a_3 \geq a_1$, and $a_3 \geq b_p$, therefore, $a_2+a_3 \geq a_1+b_p$. Therefore, level *L-1* remains weight consistent.

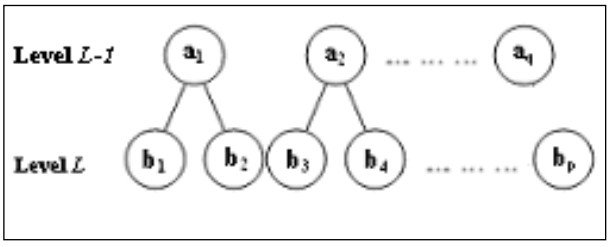
The following theorems address the case of deleting a node.

Theorem 3

Given a consistent tree, and a weight is removed from a certain level *L*, if the sum of the largest node in level *L* and the smallest node in level *L-1* is not less than the largest node in level *L-1*, then moving the largest node in level *L* to level *L-1* keeps the tree consistent.

Proof

Case 1: If the largest node in level *L* is an internal node.



Concerning the number of nodes in each level, all the levels that will be affected by moving the largest node, b_p, from level *L* to level *L-1*, will still have an even number of nodes, as the sub-tree of the node b_p in any level contains an even number of nodes. As for level *L*, removing the node b_p will adjust the number of nodes. And level *L-1* as well will have one node instead of the internal node that was formed before the deletion of a node from level *L*.

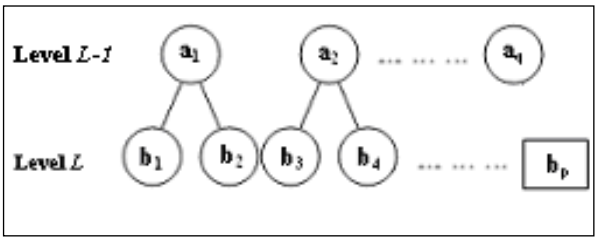
Before moving the node b_p to level *L-1*, we make sure that the sum of a_1 and b_p is not less than the largest node in level *L-1*, a_q. Therefore, level *L-1* is weight consistent. Level *L* was weight consistent before the deletion of a node, this implies that $b_1+b_2 \geq b_p$, therefore, $b_1+b_2 \geq b_{p-1}$, since $b_{p-1} \leq b_p$.

The effects in levels higher than *L*, will be the same as levels *L* and *L-1*. As for levels lower than *L-1*, closer to the root, a key observation is that moving a node from level *L* to level *L-1*, i.e. from a certain level to the next lower level, does not result in having a node in level *L-1* with a larger value than a node that was already there. This means that the node that was moved to level *L-1* is definitely smaller than the internal node that was formed in level *L-1* before the deletion (knowing that all nodes in level *L* are of a value smaller than all nodes in level *L-1*).

Case 2: If the largest node in level L is an external node.

Similar to the previous case, but no other levels will be affected as the node b_p does not have a sub-tree.

When the condition in theorem 3 does not hold, we apply the following.

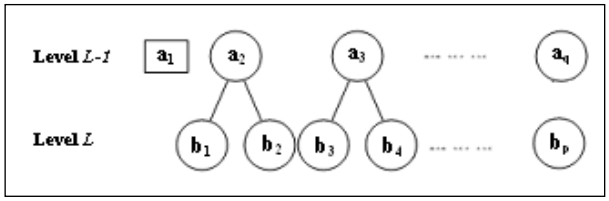


Theorem 4

Given a consistent tree, and a weight is removed from a certain level *L*, if the sum of the largest node in level *L* and the smallest

node in level *L-1* is less than the largest node in level *L-1*, then moving the smallest leaf in level *L-1* to level *L* keeps level *L* and higher levels weight and level consistent.

Proof



Moving a node form level *L-1* to level *L* will adjust the number of nodes in both levels *L* and *L-1*. The new node that was moved from level *L-1* to level *L*, a_1, will be the largest node in level *L*, so the sum of the smallest two nodes in level *L* must not be less than this node, but the sum of the smallest two nodes in level *L*, b_1 and b_2, forms an internal node in level *L-1*, a_2, and we are sure that this internal node is of a value greater than the leaf that was moved to level *L*, or else, we would have not needed to move the smallest leaf in level *L-1* to level *L*. Therefore, level *L* is weight consistent. We also need to show that level *L-1* will be weight consistent. The largest node in level *L-1* will either be a_q or the new internal node formed by a_1+b_p. If it is a_q, then it is clear that since $a_1+a_2 \geq a_q$, then $a_2+a_3 \geq a_q$ since a_2 and a_3 are both not less than a_1. But if it is a_1+b_p, then since $a_2 \geq a_1$, $a_2 \geq b_p$, $a_3 \geq a_1$, and $a_3 \geq b_p$, therefore, $a_2+a_3 \geq a_1+b_p$.

3. Insertion and deletion algorithms

In this section we present the insertion and deletion algorithms for updating a Huffman code.

3.1. Insertion algorithm

Assuming a Huffman tree of *Q* levels and given a number of leaves distributed among *K* levels, we have a new weight *x* to be inserted. We first need to find the level where *x* can be inserted. We use an implication of the exclusion property, that the sum of the smallest two nodes in any level must not be less than the largest node in the same level. We start from the highest level, furthest from the root by comparing *x* to the sum of the smallest two nodes, amongst all nodes, in the level, if *x* is greater than or equal to their sum then we move to the next lower level and repeat the process until *x* is less than the sum of the smallest two nodes in a certain level *L*, where *L:* 0..*Q-1,* and no node in level *L-1* is of a value less than *x*.

Inserting *x* in level *L* will certainly result in some other changes in the distribution of the rest of the weights among the levels, to keep the weight consistency and level consistency properties in each level.

When we claim that *x* will be inserted in a certain level *L*, this means that level *L* will be weight inconsistent as the number of nodes in level *L* now becomes odd, so some other node must be moved to another level. According to the level where *x* will be inserted, we perform some changes to the distribution of the nodes among the levels so that all levels are both weight consistent and level consistent.

Case 1: If level *L* is a low level, lower than or equal to the lowest level containing leaves, then just move the smallest two nodes, with their sub-trees, possibly including the new node *x*, from level *L* to level *L+1*.

Proof of correctness

Using Theorem 1 and Lemma 1.1, it is shown that moving the smallest two nodes from level *L* to level *L+1* keeps the tree both weight and level consistent.

Case 2: The other case is that *L* is in a higher level than the lowest level containing leaves. In that case we will need to perform an extra check before we shift the smallest two nodes with their sub-trees to the next higher level. We must make sure that there is no leaf in level *L-1* smaller than the sum of the smallest two nodes in level *L*, because if we move the smallest two nodes to level *L+1* they will combine and their sum will be an internal node in level *L*. This creates a node in level *L* with a larger value than a node in level *L-1*, leaving level *L* level-inconsistent. In this case level *L* is left as it is and the smallest leaf in level *L-1* is moved to level *L*, otherwise, it is treated similarly as the Case 1, where the smallest two nodes in level *L* are moved to level *L+1*.

Proof of correctness

In this case we have one of two resulting trees. First, if we move the smallest two sub-trees to level *L+1*, the correctness follows from Theorem 1 .The second case, moving the smallest leaf from level *L-1* to level *L*, is handled in Theorem 2.

After completion of the mentioned steps among levels *L-1*, *L*, and *L+1*, we still need to check the level consistency among all levels smaller than *L*, since it could have possibly been violated in performing the previous steps. This is done by checking that the smallest leaf in each level *P*, where *P* ranges from level *L-2* to 1, is greater than or equal to the largest node in the next higher level *P+1*, if this condition is not satisfied then all leaves in level *P* that have a value less than the largest node in level *P+1* are moved to level *P+1*. This might result in weight inconsistency for any of the two levels *P* or *P+1* which are also adjusted. The simple procedure of Section 3.3 adjusts the level and weight consistencies when needed in linear time.

Before performing all the previous steps, we need to perform one easy step, if *x* is greater than or equal to the sum of all weights given, then *x* will combine with the root of the tree and it will be in level 1 and all the other levels will just be increased by 1.

Proof of correctness

Since *x* will combine with the root of the tree, this will result in the same old tree but with one extra level, leaving all nodes in the other levels as they were both weight and level consistent.

In fig. 1, we present the pseudo-code for the insertion algorithm.

3.2. Deletion algorithm

We are given a node *x* to be deleted from level *L*.

Nodes in levels higher than *L*, are nodes with smaller values than *x*, will be combined in the same manner as the original tree, and their levels will be weight consistent, however, level *L* will be weight consistent except for the number of nodes which will be an odd number. To adjust the number of nodes in level *L* and lower levels we perform a number of steps.

Case 1: Starting at level *L*, if the sum of the largest node in level *L* and the smallest node in level *L-1* is greater than or equal the largest node in level *L-1*, then moving the largest node in level *L* to level *L-1* will not affect the weight consistency of level *L-1*, and definitely not of level L as well, and it will adjust the number of nodes in both levels, resulting in a new distribution of nodes keeping both weight and level consistencies.

Proof of correctness

By using Theorem 3, we can prove the consistency of the resulting tree.

Case 2: The other case occurs when the sum of the largest node in level *L* and the smallest node in level *L-1* is less than the largest node in level *L-1*. In this case we move the smallest leaf in level *L-1* to level *L* to adjust the number of nodes in level *L*.

Proof of correctness

It is shown in Theorem 4 that level *L* and higher levels remain level consistent**.** However, lower levels are handled by performing the level adjustment module after adjusting the number of nodes in each level.

This will leave level *L-1* having an odd number of nodes, so the process is repeated for all levels lower than *L* until the lowest level containing leaves.

The deletion algorithm is shown in fig. 2.

3.3. Consistency adjust

Adjusting the consistency of the tree requires adjusting both level and weight consistencies. It is proven in the previous theorems that the weight consistency of all levels is not violated by performing the mentioned modifications, however, level consistency can be violated in case of increasing the value of nodes in any level. It is obvious that any node in a certain level having a value smaller than another node in the next higher level must be a leaf, this is because all internal nodes are combinations of nodes in the next higher level. The level consistency is

```
Insert(x)
Begin
Given: n weights distributed among K levels, each weight is given with its level, a weight x to be
inserted.
1.  root = sum of all n weights
    If x ≥ root
    Then
        x will be inserted in level 1 and all other levels will be increased by 1.
2.  Else
        L_max=highest level containing leaves
        L_min=lowest level containing leaves

3.      For L=L_max downto 1            //find the level in which x will be inserted
                S_1=smallest node in level L amongst internal and external nodes
                S_2=second smallest node in level L amongst internal and external nodes
                If x≥S_1+S_2 Then go to next L
                Else exit loop
                End If
        End For                         //L=level where x will be inserted
        S_1'=smallest leaf in level L-1
        If S_1'<x Then L=L-1 End If
        If L≤L_min                      //Case 1
        Then    S_1 and S_2 go to level L+1
        //levels of the weights contributing in S_1 and S_2 are increased by 1
        Else                            //Case 2
                If S_1+S_2 ≤ smallest leaf in level L-1
                Then S_1 and S_2 go to level L+1
                Else the smallest leaf in level L-1 goes to level L
                End If
    End If
4.              Adjust LevelandWeightConsistency(L, L_min)

End If
End
```

Fig. 1. Insertion algorithm.

adjusted by moving all leaves in a certain level having a value smaller than the largest node in the next higher level to the next higher level. This can result in having an odd number of nodes in any of the updated levels, so this is handled by theorems 1 and 2, by either moving the smallest two nodes in the current level one level up, further from the root, or the smallest leaf from the next lower level is moved to the current level, this is proven to be correct in theorems 1 and 2.

Proof of correctness

Let the current level be *L*. Moving the leaves from level *L-1* to level *L* implies that these leaves are of value smaller than the sum of the smallest two nodes in level *L*, the smallest internal node in level *L-1*. This assures the weight consistency in level *L* and higher levels. Level *L-1* now contains new internal nodes, however, all these new nodes are combinations of nodes smaller than the two smallest nodes in level *L-1*, this guarantees that the sum of the smallest two nodes in *L-1* remains greater than or equal to the largest node in the level.

Fig. 3 shows the algorithm for adjusting tree consistency.

Delete(*x*, *L*)
Begin
Given a node *x* to be deleted from a certain level *L*
Let S_{maxi} be the maximum node in level *i*
Let S_{mini} be the minimum node in level *i*
Let L_{min} be the lowest level containing leaves

1. **If** $S_{maxL}+S_{minL-1} \geq max_{L-1}$
   Then S_{maxL} goes to level *L-1*
   Else
2. S_{minL-1} goes to level *L* //this leaves level *L-1* having an odd number of nodes
3. **For** *i* = 1 to $L-L_{min}$
   If $S_{maxL-i} + S_{minL-i-1} \geq S_{maxL-i-1}$
   Then S_{maxL-i} goes to level *L-i-1*
   Else $S_{minL-i-1}$ goes to level *L-i*
   End If
   End For
4. Adjust LevelandWeightConsistency(*L*, L_{min})
   End If
End

Fig. 2. Deletion algorithm.

Adjust_Level_Weight_Consistency(*L*, *Lmin*)
For *i* = 1 to *L-Lmin*
If smallest leaf in level *L-i-1*< largest node in level *L-i*
Then Move all leaves in level *L-i-1* that are less than largest node in *L-i* to level *L-i*
End If
//adjust the number of nodes in levels *L-i* and *L-i-1*
If *L-i* has an odd number of nodes
Then
S1 = smallest node in *L-i*
S2 = second smallest node in *L-i*
If *S1+S2* ≤ smallest leaf in level *L-i-1*
Then Move *S1* and *S2* to level *L-i+1*
Else Move smallest leaf in level *L-i-1* to level *L-I*
End If
End If

If *L-i-1* has an odd number of nodes
Then
S1 = smallest node in *L-i-1*
S2 = second smallest node in *L-i-1*
If *S1+S2* ≤ smallest leaf in level *L-i-2*
Then Move *S1* and *S2* to level *L-i*
Else Move smallest leaf in level *L-i-2* to level *L-i-1*
End If
End If
End For
End

Fig. 3. Adjusting level and weight consistencies algorithm.

4. Complexity analysis

The complexity of the given algorithms results from the need to find the minimum and maximum elements in each level. To do this, we need to sort each level in order to obtain the internal nodes in the next lower level. Instead we keep the internal nodes unknown, and by avoiding sorting give an $O(nK)$ implementation, where n is the total number of weights given, and K is the number of levels containing leaves. We also present another practical $O(n \log n)$ implementation.

4.1. An $O(nK)$ implementation

The idea behind this implementation is to avoid sorting the weights in order to obtain a linear complexity, at least in some cases of practical use. In order to do that, a linear amount of work in each level is done. In each level, we divide the nodes into two halves, by finding the middle element and partitioning the nodes around it. This process is repeated for each level until the smallest two nodes in that level are found, and also the middle two elements in that level are found, which will form the middle element in the next level. This process is continued until the lowest level containing leaves is reached. As for lower levels, no extra work will be needed, because all the nodes in lower levels can be obtained from the last level containing leaves as a sum of its smallest half, quarter, eighth, and so on.

Here are some of the details. Assume a level L having some internal and external nodes. In order to divide the nodes in this level into two halves it is required to find its middle element, but the problem is that we do not know the internal nodes. However, the sum of the two middle elements in the next higher level forms the middle element in the current level. As for the external nodes, the middle element can be found by a selection algorithm in linear time [11]. At this point, we have two elements, the median of the leaves and the median of the internal nodes, by comparing these two elements, we can exclude some nodes that are definitely not going to contribute in the smaller half of the nodes in that level. The nodes to be excluded are those nodes that are of a value greater than the larger median. Recursively, the rest of the needed information can be obtained.

Knowing that finding the median of a list of elements takes linear time using a selection algorithm, this process takes $O(n_{L-I} + n_{L-i+1} + n_{L-i+2} + \ldots + n_L)$ for a certain level L, where n_j is the number of leaves in the j^{th} highest level.

Therefore, the mentioned steps take a linear time in the number of nodes of the sub-tree below the current level.

In total the time taken will be:

$(K)n_1 + (K-1)n_2 + (K-2)n_3 + \ldots + n_K$

Therefore, this algorithm takes linear time if the number of levels, K, is constant, or if the number of nodes in each level is constant, i.e. not a function of n. And it takes $O(n \log n)$ if K is $O(\log n)$. We can use this algorithm as long as Kn is less than $n \log n$.

Next we give a practical $O(n \log n)$ implementation.

4.2. A practical $O(n \log n)$ implementation

We can simply implement the given algorithms in $O(n \log n)$ time by actually finding all the internal nodes in each level, just sort the weights given in each level and combine each two nodes then merge with the sorted leaves in the next lower level.

Given the sequence of nodes distributed among the levels, we have n_1 weights in the highest level, n_2 weights in the second highest level, and so on until n_K weights in the K^{th} highest level.

Starting from the highest level, we first sort the n_1 weights in $O(n_1 \log n_1)$ time, and then we combine each two nodes to obtain $n_1/2$ nodes in the next level which are merged with the sorted n_2 weights of that level. We repeat for the next level. The time taken to complete this process can be expressed as follows:

$$T_k < C_1(n_1 Log\ (n_1) + \frac{n_1}{2} + n_2 + n_2 Log\ (n_2) + \frac{n_1}{4} + \frac{n_2}{2} + n_3 + n_3 Log\ (n_3) + \ldots + n_k + n_k Log\ (n_k))$$

$$T_k < C_1(\frac{n_1}{2} + \frac{n_1}{4} + \ldots + \frac{n_1}{2^k} + n_2 + \frac{n_2}{2} + \frac{n_2}{4} + \ldots$$

$$+\frac{n_2}{2^{k-1}}+n_3+\frac{n_3}{2}+\frac{n_3}{4}+\ldots$$

$$+\frac{n_3}{2^{k-2}}.+\ldots+n_k+n_1 Log\ (n_1)+n_2 Log\ (n_2)+n_3 Log\ (n_3)+\ldots$$

$$+n_k Log\ (n_k))$$

$$T_k < C_2(2n_1+2n_2+2n_3+\ldots+2n_k+n_1 Log\ (n_1)+n_2 Log\ (n_2)+\ldots$$

$$+n_k Log\ (n_k))$$

$$T_k < C_3(2n+nLog\ (n))$$

Where C_1, C_2, C_3 are constants.

Therefore, the above steps take $O(n \log n)$ time.

When the given algorithm is applied, it will not take any extra time more than $O(n)$ time in order to perform the consistency checks.

For the insertion algorithm, shown in fig. 1, the time taken for each step is as follows:

Step 1: Already done and takes $O(n)$ time to scan the n weights and sum them.

Step 2: Scans the levels once in $O(n)$ time to obtain L_{max} and L_{min}.

Step 3: For each level, the smallest two nodes are obtained and their sum is compared to x. Scanning the levels requires $O(n)$ time, and the smallest nodes in each level are already available so it takes constant time to find their sum.

Step 4: Takes $O(n)$ time as will be discussed in Section 4.3.

As for the deletion algorithm, Shown in fig. 2, the maximum and minimum elements of all the levels are now ready to use without any extra time, all we need to do is just shifting nodes from one level to another.

Therefore, the total time taken is $O(n \log n)+O(n)$. Hence, the computational complexity of the algorithm is $O(n \log n)$.

In many cases, this algorithm takes less time than $O(n \log n)$, because in both the insertion and deletion algorithms, it is not always needed to sort and merge until the last level. In fig. 1, consider all the cases that are solved in Step 3, Case 1, the sorting and merging is needed only up to the insertion level. Also Step 1 in the deletion algorithm in fig. 2, this needs sorting and merging until one level lower than the deletion level.

4.3. Complexity analysis of adjusting level and weight consistencies algorithm

Given that the smallest and largest nodes in each level are already obtained as explained in Sections 4.1 and 4.2, the time taken by this algorithm is described as follows:

Step 1: This step is linear in the number of leaves in the current level.

Step 2: Takes constant time as we already know the required nodes.

Therefore, the time taken to adjust the tree consistency is linear.

5. Illustrative examples

The following examples are illustrations for the different cases mentioned in the insertion and deletion algorithms.

Given the distribution of weights in fig. 4, and a weight x to be inserted or deleted.

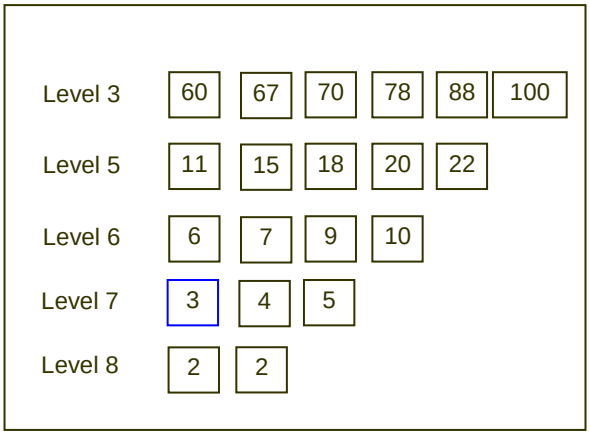


Fig. 4. Given distribution of weights.

Example 1: insert x = 65

Starting at level 8 by comparing the sum of the smallest two nodes in each level to the value of x, we find that the new weight x can be inserted in level 3, where the sum of the smallest two nodes is formed by summing the weights 60, 11, 15, 6, 7, 9, 3, 2, and 2, and results in a sum of 115.

Shifting the nodes that form the smallest two nodes in level 3 one level, we obtain the following distribution, shown in fig. 5.

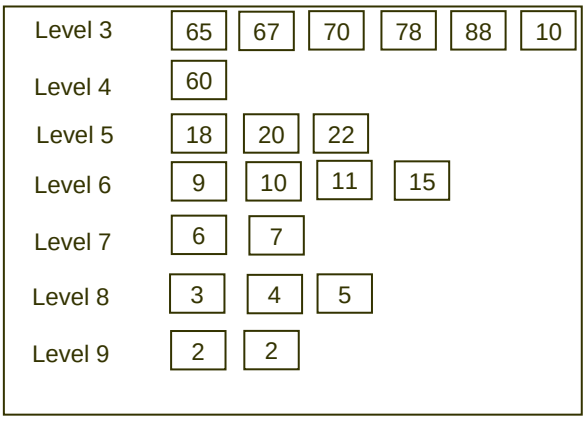


Fig. 5. Example 1 – new distribution of weights.

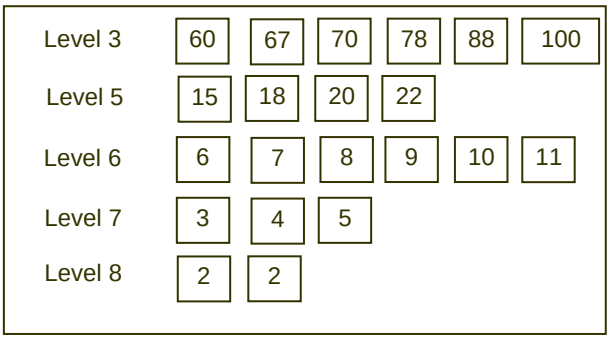


Fig. 6. Example 2 – new distribution of weights.

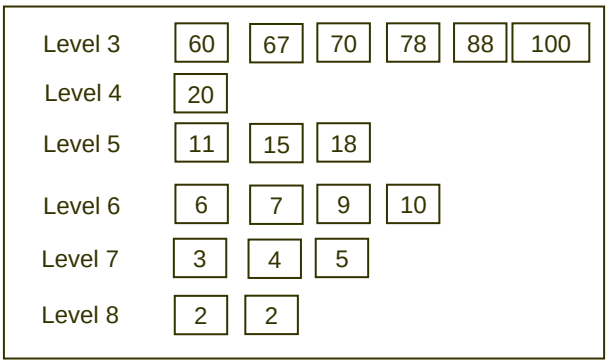


Fig. 7. Example 3 – new distribution of weights.

Example 2: insert x = 8

In this case, shifting the smallest two nodes one level will cause weight inconsistency, and therefore, we follow case 2 of the insertion algorithm and obtain the distribution shown in fig. 6.

Example 3: delete x = 22

Following the deletion algorithm, when we remove the weight 22 from level 5, it is seen that moving the largest node, of value 20, to level 4 will keep level 4 weight consistent, resulting in the distribution in fig. 7.

6. Conclusions

In this paper we presented two algorithms for the update of a Huffman code knowing only the list $W = [w_1,\ldots, w_n]$ of n positive symbol weights, and a list $L = [l_1,\ldots,l_n]$ of n corresponding integer codeword lengths, with the n weights being distributed among K levels, and no other information is know about the tree. The first algorithm handled the insertion of a new weight, and the second was for the deletion of an already existing weight. As for the update of a weight, this can be done by simply deleting the old value and reinserting the new one.

The algorithms presented treat many practical cases in linear time. The insertion algorithm takes $O(nK)$ time, so it needs linear complexity if the weights given are distributed among a constant number of levels, i.e. the number of levels is not a function of n, it also takes linear time if the number of weights in each level is constant, and this makes the time needed to sort the weights linear.

Another practical $O(n \log n)$ implementation is given. It is obvious that in many cases, the time taken by the algorithm is faster than rebuilding the Huffman tree, because it is not always required to sort and merge all levels, so this results in a faster algorithm.

Similarly, the deletion algorithm takes linear time in many cases, in all cases where it is possible to make the update by just moving the largest node one level closer to the root, the algorithm takes linear time if the number of the level from which the node was deleted is not a function of n.

References

[1] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proc. IRE 40, pp. 1098-1101 (1952).

[2] J. Zobel and A. Moffat, "Adding Compression to a Full-Text Retrieval System", Software Practice and Experience Vol. 25 (8), pp. 891-903 (1995).

[3] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression", IEEE Transactions on Information Theory, Vol. 23, pp. 337-349 (1977).

[4] A. Bookstein and S.T. Klein, "Models of Bitmap Generation: A Systemayic Approach to Bitmap Compression", Information Processing and Management, Vol. 28, pp. 735-748 (1992).

[5] J. Van Leeuwen, "On the construction of Huffman trees", 3rd International Colloquium for Automata, Languages and Programming (ICALP) pp. 382-410 (1976).

[6] N. Faller, "An Adaptive System for Data Compression", In Proceedings of the 7th Asilomar Conference on Circuits, Systems, and Computers, pp. 593-597 (1973).

[7] R. G. Gallager, "Variations on a Theme by Huffman", IEEE Transactions on Information Theory, Vol. 24 (6), pp. 668-674 (1978).

[8] D. E. Knuth, "*Dynamic Huffman Coding*", Journal of Algorithms, Vol. 6 (2), pp. 163-180 (1985).

[9] J. S. Vitter, "Design and Analysis of Dynamic Huffman Codes", Journal of the ACM, Vol. 34 (4), pp. 825-845 (1987).

[10] A. Belal and A. Elmasry, "Verification of Minimum-Redundancy Prefix Codes", DIMACS Technical Report (29) (2004).

[11] T. Cormen, C. Leiserson and R. Rivest. Introduction to Algorithms. The MIT Press (1990).

Received March 28, 2005
Accepted July 3, 2005