A Space-Economical Algorithm for Minimum-Redundancy Coding

Ruy Luiz Milidiú
milidiu@inf.puc-rio.br
Artur Alves Pessoa
artur@learn.fplf.org.br
Eduardo Sany Laber
laber@inf.puc-rio.br

PUC-RioInf.MCC02/98 January, 1998

Abstract: The *minimum redundancy prefix-free code problem* is to determine, for a given set $W = \{w_1, \ldots, w_n\}$ of n positive weights, a set $\lambda = \{l_1, \ldots, l_n\}$ of n integer codeword lengths such that $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ and $\sum_{i=1}^{n} w_i l_i$ is minimized. In this technical report, we consider the case where W is already sorted and must remain unchanged after the codeword lengths are calculated. The algorithm proposed in this technical report solves this problem in $O(n)$ time using just $O(\min\{l_1^2, n\})$ additional space, where l_1 is the greatest codeword length in λ.

Keywords: Prefix Codes, Huffman Trees, Data Compression

Resumo: Calcular um código livre de prefixo com redundância mínima equivale a determinar para um dado conjunto de n pesos positivos $W = \{w_1, \ldots, w_n\}$, um conjunto de n comprimentos de código inteiros $\lambda = \{l_1, \ldots, l_n\}$ tal que $\sum_{i=1}^{n} 2^{-l_i} \leq 1$, minimizando $\sum_{i=1}^{n} w_i l_i$. Nesta monografia, consideramos que W é fornecido sob forma de uma lista ordenada que não pode ser destruída durante o cálculo dos comprimentos de código. O algorítmo proposto nesta monografia resolve o problema em tempo $O(n)$ utilizando um espaço adicional $O(\min\{l_1^2, n\})$, onde l_1 é o comprimento do maior código.

Palavras-Chave: Códigos de Prefixo, Árvores de Huffman, Compressão de Dados

1 Introduction

Minimum-redundancy coding plays an important role in data compression applications [Gai93, ZM95]. Methods for calculating a set of minimum-redundancy codewords that correspond to a set of input symbol weights are of great interest. For an input list of n symbol weights, the well-known Huffman's algorithm generates a set of codewords in $O(n \log n)$ time and $O(n)$ extra space [Huf52]. Leeuwen showed that the time complexity of Huffman's algorithm can be reduced to $O(n)$ if the input list is already sorted [vL76]. As a simplifying assumption, we assume that the input weights are already sorted and refer to Leeuwen's implementation as Huffman's algorithm throughout this technical report.

Moffat and Katajainen [MK95] showed that the code calculation can be done in-place, that is, using $O(1)$ additional space. This method overwrites the input list of weights with the corresponding list of codeword lengths.

In this technical report, we assume that the symbol weights need to be available after the corresponding codeword lengths are generated. The WARM-UP algorithm [ML96], a quasi-optimal method for the length-restricted code problem proposed by Milidiú and Laber, is an example where this assumption is true. This algorithm performs a sequence of Huffman tree constructions for modified sets of weights. The symbol weights may also be useful for statistical purposes. Under this assumption, the in-place method requires $2n + O(1)$ words of total memory, what is also a minimum requirement because the input list of weights and the output list of codeword lengths require both $2n$ words of memory.

However, Moffat and Turpin [MT99] proposed an space-efficient representation of minimum-redundancy codes: a list of l_1 *runlengths*, where l_1 is the length of the longest codeword. For $i = 1, \ldots, l_1$, the value of the ith runlength is equal to the number of codewords with length i. Observe that a message can be codified using only $O(l_1)$ additional space through the *canonical* coding method [HL90]. This coding method is also faster than the traditional one. Since we always have $l_1 \leq n$, we assume that the output is given as a list of runlengths.

Moffat and Turpin [MT99] also present an algorithm that constructs a minimum-redundancy code in $O(r + r\log(n/r))$ time, where r is the number of distinct weights in W. This algorithm requires only $O(r + r\log(n/r))$ additional space and does not destroy the input list of weights.

In this technical report, we use a lazy approach to find the value of l_1 for an input list of n weights in $O(n)$ time, and using $O(l_1)$ additional space. The resulting method is called the *LazyHeight* algorithm. Furthermore, we present the *LazyHuff* algorithm, that calculates a minimum-redundancy code in $O(n)$ time, and using $O(\min\{l_1^2, n\})$ additional space.

Buro [Bur93] proved that no codeword of a binary Huffman code is longer than $\lfloor \log_\psi(\frac{\psi+1}{p_1\psi+p_2}) \rfloor$, where p_1 and p_2 ($p_1 \leq p_2$) are the probabilities of the two least probable symbols and ψ is the golden ration $\frac{1+\sqrt{5}}{2} \approx 1.618$. Since the probability of one symbol in a source message with N symbols cannot be smaller than $1/N$, it is immediate that the codeword lengths are bounded above by $\lfloor \log_\psi N \rfloor$. Therefore, we always have $l_1 \leq \lfloor \log_\psi N \rfloor$.

Observe that, in most practical cases, $\lfloor \log_\psi N \rfloor$ is considerably smaller than n, particularly for large alphabets. For example, Moffat and Turpin used a word-based model to compress the three-gigabyte TREC document collection [MT99], finding $480,911,085$ word appearances and $1,073,971$ distinct words. In this case, the maximum codeword length cannot be greater than 41, what leads to a very small space consumption.

This technical report is organized as follows. In section 2, we present an overview of Huffman's algorithm. In section 3, some properties of Huffman trees are discussed. In section 4, we describe the *LazyHeight* algorithm. In section 5, we present the *LazyHuff* algorithm. In the following section, we present some considerations on the time and space requirements of both LazyHeight and LazyHuff algorithms. Finally, in section 7, we summarize our conclusions and comments.

2 Huffman's Algorithm

In this section, we present an overview of the well known $O(n)$-time implementation of Huffman's algorithm, assuming the input to be a sorted list of weights.

First, let us define some terminology for binary trees. A node s is a *tree* and, if t_1 and t_2 are two trees, then $t = s(t_1, t_2)$ is also a tree with s as its *root* node. In the tree $t = s(t_1, t_2)$, the node s is the *parent* of s_1 and s_2 if s_1 and s_2 are the root nodes of t_1 and t_2. Similarly, s_1 and s_2 are the children of s: s_1 is the *left child* of s and s_2 is its *right child.* If a node has no children then it is a *leaf* node of the tree. Otherwise, it is an *internal* node. The *level* of any node is one greater the level of its parent; the level of a root node is zero. Finally, two nodes are siblings if they have the same parent.

Huffman's algorithm can be described as the construction of a binary tree, where each weight in W is assigned to a leaf in the tree and the corresponding codeword length in λ is defined as its level in the tree. This method is based on the fact that every prefix code can be represented by a binary tree [Eve79]. The binary trees constructed by Huffman's algorithm will be refered as Huffman trees.

To each internal node of a Huffman tree it is also assigned a weight equal to the sum of the weights of its children. A stack and a queue are maintained, respectively for the leaf nodes and for the internal nodes as they are formed, such that the nodes in both the stack and the queue are arranged in a non-decreasing order by their weights. Initially, the queue is empty and the smallest weight w_1 is at the stack top. At each step, a new internal node y is formed. The two children of y are the ones with the smallest weight among four selected nodes, two from the current top of the queue and two from the current top of the stack. When ties occur, we choose leaf nodes. Each chosen node is removed and the corresponding top cursor updated for either the stack or the queue. The process stops when just one node (the root of the tree) remains in the queue and the stack is empty.

In the whole process, $n-1$ internal nodes are formed, what leads to the linear time and space complexity of the algorithm.

3 Huffman Trees

In this section, we define some concepts that will be extensively used throughout this technical report.

First, we assign to each node in a Huffman tree a distinct integer label between 1 and $2n-1$. The nodes in a Huffman tree are labeled in the following way. Given two nodes at different levels, the node at the lowest level has the greatest label. If two nodes are at the same level, then the right most node has the greatest label. Figure 1 shows a Huffman tree where internal nodes are represented by circles and leaf nodes are represented by squares. The weight of each node is the number that appears inside its corresponding pictorial representation. The numbers outside are node labels.

Now, we define two concepts related to the order of appearance of nodes in a Huffman tree. We say that a node s_2 *appears after* another node s_1 if and only if the label of s_2 is greater than that of s_1. If s_2 appears after s_1, then s_1 *appears before* s_2. Similarly, we say that a node s_2 *appears immediately after* another node s_1 if and

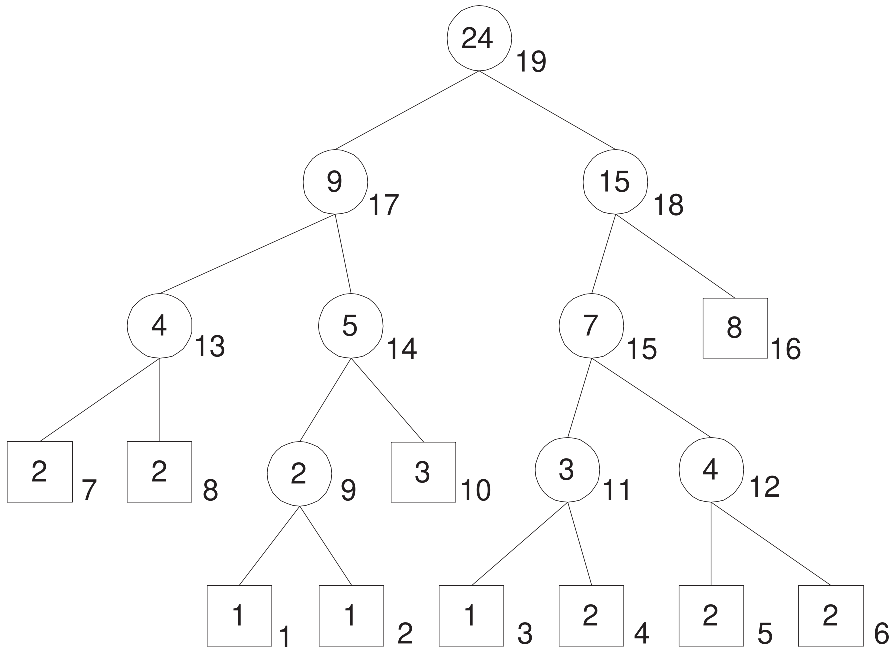

Figure 1: This figure shows a Huffman tree where the nodes are labeled

only if the label of s_2 is one greater than that of s_1. If s_2 appears immediately after than s_1, then s_1 *appears immediately before* s_2.

Observe that, if the weight of a node s_2 is greater than the weight of another node s_1, then the label of s_2 is greater than that of s_1, that is, s_2 appears after s_1 in the Huffman tree. In addition, if s_1 is a leaf node, s_2 is an internal node and both s_1 and s_2 have the same weight, then s_2 appears after s_1. Moreover, if two nodes s_1 and s_2 are, respectively, the left and the right child of another node s, then s_2 appears immediately after s_1 in the Huffman tree.

4 The LazyHeight Algorithm

Solving a problem with a lazy approach can be summarized by the following statement: "Perform steps as their outputs become necessary". Clearly, each step of Huffman's algorithm is characterized by the creation of a new internal node. Therefore, a natural question appears: *When does each internal node become necessary?*

Before we present the answer to this question, we show a lazy algorithm to find the height of a Huffman tree in $O(n)$ time, using only $O(l_1)$ additional space. Let us call it the *LazyHeight* algorithm.

4.1 Description

First, we need to define some additional terminology. Let us define the *height* of an internal node in a binary tree as one plus the maximum between the heights of its children. The height of a leaf node is zero. The height of a binary tree is the height of its root node. Every node is an ancestor of itself. If s is a node, the parent of any ancestor of s is also an ancestor of s. Finally, a node s_1 is a descendent of another node s_2 if and only if s_2 is an ancestor of s_1.

Observe that, if s is an ancestor of the first leaf node in a Huffman tree, then the height of s is greater than or equal to the height of its sibling. Based on this fact, the LazyHeight algorithm finds the height h of the tree by building the path that connects the first leaf node to the root node. Figure 2 illustrates the construction of a Huffman tree by the algorithm, for the list of weights $[1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 8]$. Circles represent internal nodes and squares represent leaves. The final state of the algorithm construction is presented in figure 2.(a). Let ϕ_0 be the first leaf node and let ϕ_i be the ancestor of ϕ_0 at level $h - i$, for $i = 1, \ldots, h$. In figure 2, we use shaded nodes to indicate the ancestors of ϕ_0 and a dashed line to bound the current tree that contains the path.

The nodes in the path are constructed by LazyHeight, starting from ϕ_0 until reaching the root. In the ith step, the algorithm obtains the sibling s_i of the internal node ϕ_i and constructs the internal node ϕ_{i+1}. In order to build the sibling of each ancestor of ϕ_0, LazyHeight maintains four lookahead variables at each level. In figure 2, lookaheads are represented by either bold circles or bold squares. There are two left-side lookahead variables containing a leaf and an internal node and two right-side lookahead variables containing another leaf and another internal node. We use x_i and y_i, respectively, to denote the left-side lookahead corresponding to a leaf and the left-side lookahead corresponding to an internal node, both associated to level $h - i$. Similarly, we use x'_i and y'_i to denote the right-side lookaheads. A pseudo-code for the algorithm is presented below.

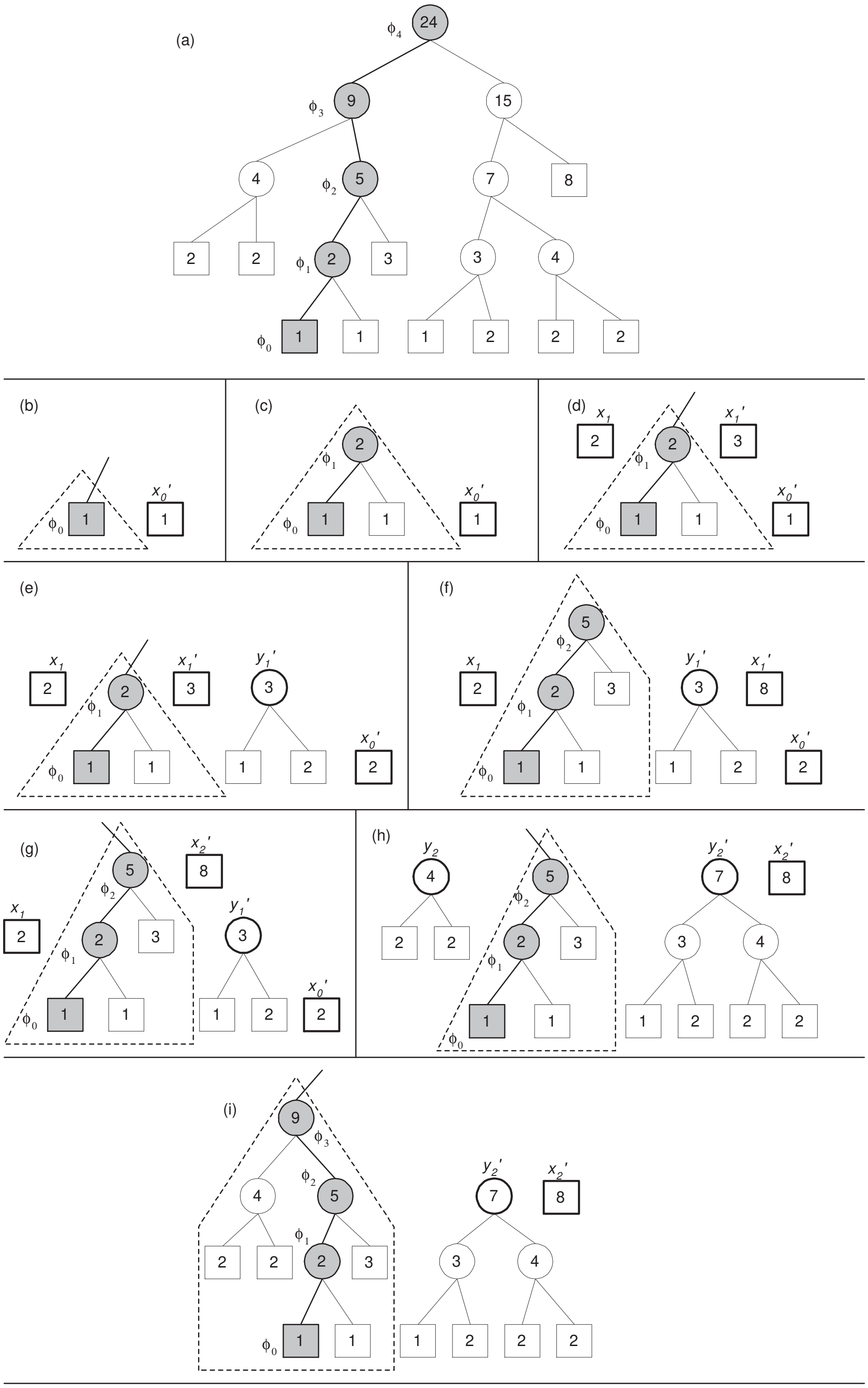

Figure 2: Construction of a Huffman Tree by the LazyHeight Algorithm

LazyHeight Algorithm
Let ϕ_0 be the first leaf node in the Huffman tree;
initialize x'_0 with the second leaf node;
Repeat until ϕ_i is the root of the tree
Step 1: choose from the lookahead variables at level $h-i$ which one is the sibling of ϕ_i;
Step 2: form the node ϕ_{i+1} and update the lookahead variable chosen in the previous step;
Step 3: initialize x_{i+1} and x'_{i+1} and determine if ϕ_{i+1} is a right child or a left child;
Step 4: initialize y_{i+1} and y'_{i+1};
Step 5: $i \leftarrow i+1$;
End Repeat

Let us examine the main loop of LazyHeight. Let ϕ_i be the currently highest ancestor of ϕ_0. If ϕ_i is a left child, then the algorithm chooses from the right-side lookaheads at level $h-i$, which one is the sibling s_i of ϕ_i. This is performed, by choosing the right-side lookahead with the smallest weight. If their weights are equal, then the leaf lookahead is chosen. In figure 1.(e), the currently highest ancestor of ϕ_0, the node ϕ_1, is a left child. Since both x'_1 and y'_1 have weights equal to 3, then x'_1 is chosen. If ϕ_i is a right child, the procedure is similar, with left-side lookaheads being used however.

In step 2, ϕ_{i+1} is formed and the lookahead chosen in step 1 must be updated. As an example, in figure 1.(f) the node ϕ_2 is formed and x'_1 is updated receiving a leaf with weight 8. When a right-side lookahead variable is updated, it receives the first node of the same type [1] that appear after itself in the final tree. On the other hand, when a left-side lookahead variable is updated, it receives the last node of the same type that appears before itself in the final tree.

In step 3, x_{i+1} and x'_{i+1} are initialized. Following our example, in figure 1.(g) x_2 and x'_2 are initialized. The lookahead x_2 receives a null value and the lookahead x'_2 receives the leaf with weight 8. At this point, one knows the total number of nodes between ϕ_i and ϕ_{i+1}. Thus, it is easy to determine whether ϕ_{i+1} is a left or a right sibling.

[1] We say that two nodes have the same type if both either are leaves or are internal nodes

Finally, in step 4, y_{i+1} and y'_{i+1} are initialized. In figure 1.(g), the lookahead y_2 receives the internal node with weight 4 and y'_2 receives the internal node with weight 7. Whenever a lookahead corresponding to an internal node is either initialized or updated, the algorithm chooses its children from the lookaheads at one level bellow in the tree. The chosen lookahead variables are recursively updated.

4.2 Data Structure

In this subsection, we define the data structure for the LazyHeight algorithm.

The lookahead variables corresponding to level $h - i$ are initialized as follows:

- x_i receives the last leaf node that appears before the node ϕ_i;
- y_i receives the last internal node that appears before the node ϕ_i;
- x'_i receives the first leaf node that appears after the node ϕ_i;
- y'_i receives the first internal node that appears after the node ϕ_i.

If the node assigned to a lookahead variable does not exist, this variable receives a null value that we call *nothing*. Observe that each node assigned to a lookahead variable is kept only until its parent is formed, what means that each variable may be updated several times during the execution of the algorithm.

In addition, we define a counter c_i corresponding to each node ϕ_i, for $i = 0, \ldots, h-1$, which is initialized with the number of internal nodes that appear after the node ϕ_i and before the node ϕ_{i+1}. If $i = h$, then c_h is initialized with zero. Whenever an internal node is formed, its corresponding counter is decreased by one. Hence, all the counters are zero when the algorithm stops. This counter avoids that an internal node is formed twice.

Finally, let the node ϕ_i be the highest ancestor of the first leaf node formed at a given point of an execution of LazyHeight algorithm. In this case, for $k = 1, \ldots, i$, the semantics of the data structure maintained by the algorithm can be described as follows:

- c_k stores the number of internal nodes that is not formed and that appears after the node ϕ_k and before the node ϕ_{k+1};
- x_k stores the last leaf node that is not used and that appears before the node ϕ_k and after the node ϕ_{k-1};

- y_k stores the last internal node that is not used and that appears before the node ϕ_k and after the node ϕ_{k-1};
- x'_k stores the first leaf node that is not used and that appears after the node ϕ_k and before the node ϕ_{k+1}.
- y'_k stores the first internal node that is not used and that appears after the node ϕ_k and before the node ϕ_{k+1}.

We say that a node is used if its parent is already formed. Furthermore, both y_{k+1} and y'_k receive nothing if ϕ_{k+1} is the first internal node that appears after ϕ_k or if all internal nodes that appear between them are used. However, x_{k+1} and x'_k may store any leaf node (or nothing) if no unused leaf node is found between ϕ_k and ϕ_{k+1}.

In order to illustrate the data structure maintained by the LazyHeight algorithm, consider the Huffman tree shown in figure 3.(a). Figure 3.(b) represents the state of this data structure immediately before the node ϕ_4 is formed. It shows the ancestors of the first leaf node that are already formed ($\phi_0,\ldots,\phi_3$), the node currently assigned to each lookahead variable (x_0, …, x_3, y_0, …, y_3, x'_0, …, x'_3, y'_0, …, y'_3) and the nodes that are used, drawn with dashed lines. Recall that the nodes that are used are not maintained in memory.

Observe that nothing is assigned to y_0, y'_0 and y_1 because the node ϕ_1 is the first internal node, that is, there is no internal node before it. For the same reason, the value of c_0 is zero. Similarly, the leaf node assigned to x_0 is unknown because ϕ_0 is the first leaf node. The leaf nodes assigned to x'_0 and x_1 are also unknown, since all the leaf nodes that appear after ϕ_0 and before ϕ_1 (the nodes labeled from 2 until 9) are used. The internal node with label 17 is assigned to both y'_1 and y_2. Since an internal node cannot be formed twice, this is a special case that is discussed in subsection 4.7. Observe that all the internal nodes that appears after ϕ_1 and before ϕ_2 are already formed, and then the value of c_1 is also zero. The node with label 20, that is assigned to x_2, is the last leaf node that appears before ϕ_2 and after ϕ_1 that is not used, since the leaf node with label 21 is used. Similarly, the node with label 27, that is assigned to y'_2, is the first internal node that appears after ϕ_2 and before ϕ_3 that is not used, since the internal nodes with labels 24 and 25 are used. Both the lookahead variables x'_2 and x_3 store unknown leaf nodes because the node with label 26, the only leaf node that appears after ϕ_2 and before ϕ_3, is used. The value of c_2 is 2 since both the internal nodes with labels 28 and 29 appear after ϕ_2 and before

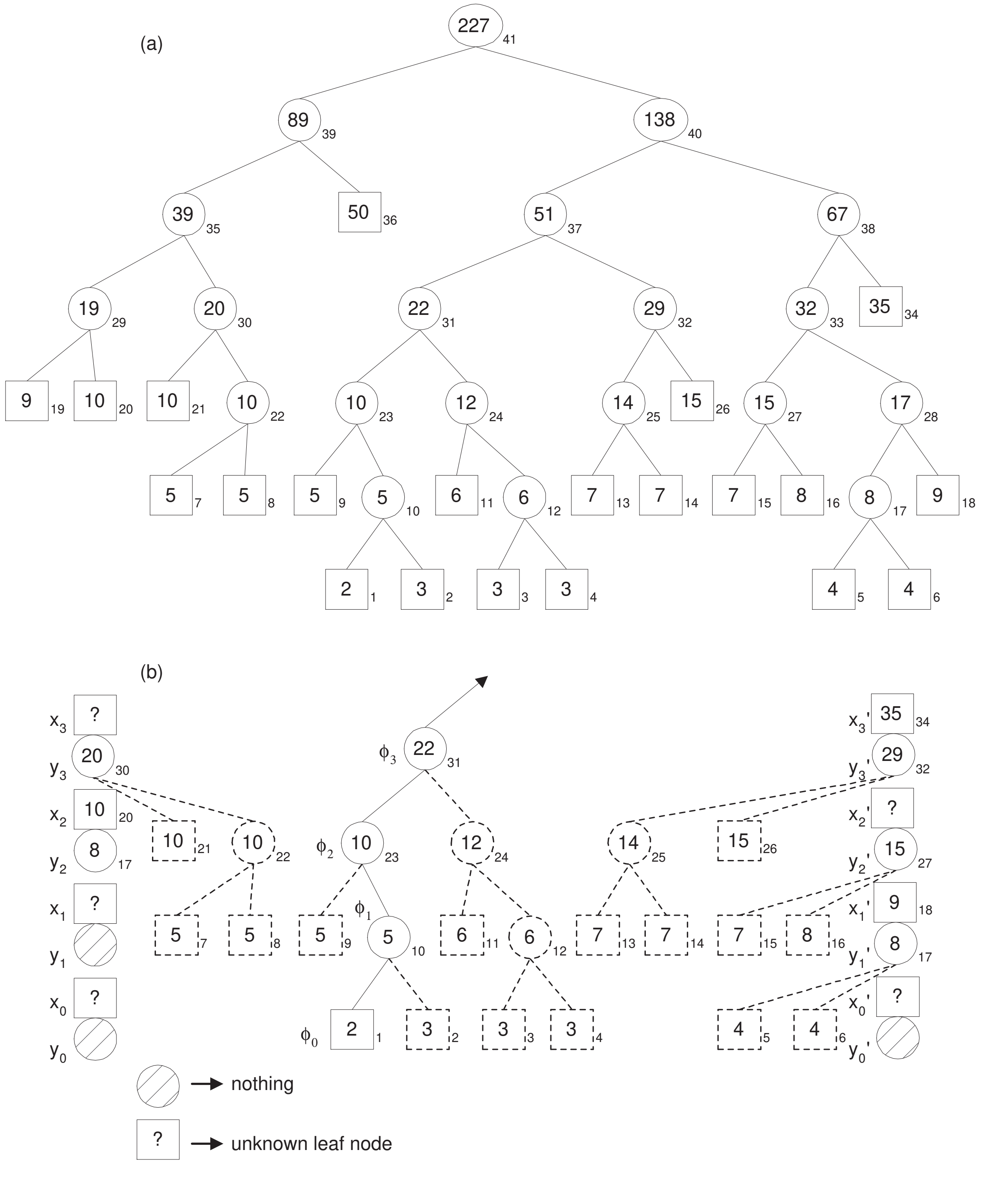

Figure 3: (a) Example of Huffman tree where the nodes are labeled; (b) Data structure of the LazyHeight algorithm for the tree of figure (a) after ϕ_3 is formed.

ϕ_3 (see figure 3.(a)) and are not formed. Similarly, $c_3 = 2$ because both the internal nodes with labels 33 and 35 are not formed.

4.3 The Main Loop

Now, let us formaly describe the algorithm. First, LazyHeight initializes y_0 and y'_0 with nothing, and x'_0 receives the second leaf node, since ϕ_0 is the first leaf node. c_0 receives zero since ϕ_1 is the first internal node. In addition, the node ϕ_0 is the left child of ϕ_1 because it is the leftmost node that appears in its level.

Now, we define an invariant satisfied at the beginning of the main loop of the LazyHeight algorithm. Let the node ϕ_i be the highest ancestor of the first leaf node that is already formed. The lookahead variables x_i, y_i, x'_i and y'_i were recently initialized and so, their states are as described in subsection 4.2. The counter c_i stores the number of internal nodes that appear after the node assigned to y'_i and before the node ϕ_{i+1}.

The algorithm also knows if ϕ_i is a left child or a right child. The root is considered a left child.

If the node assigned to a lookahead variable does not exist, this variable is nothing. In order to form the node ϕ_{i+1} (if it exists) and maintain the invariant for the next iteration, the following steps are performed:

Step 1: Choose from the lookahead variables corresponding to ϕ_i which one is sibling of ϕ_i. Step 2: Calculate the weight of the node ϕ_{i+1} and call procedure UPDATE in order to update the lookahead variable chosen in the previous step. Step 3: Initialize x'_{i+1}, x_{i+1} and c_{i+1}. Determine if ϕ_{i+1} is a right child or a left child. Step 4: Call procedure UPDATE to initialize y_{i+1} and y'_{i+1}.

The procedure UPDATE is described in subsection 4.5.

4.4 Choosing the Sibling of ϕ_i

First, we define the function AFTER and the function BEFORE.

The function AFTER(i) chooses between the right-side lookaheads stored in x'_i and y'_i, which one is the node that appears at first in the tree. Similarly, the function BEFORE(i) chooses between the left-side lookaheads stored in x'_i and y'_i, which one is the node that appears at last in the tree. The function $\rho(s)$ returns the weight of the node s.

Function AFTER(i)

If x'_i = nothing **Then**
        return y'_i;
Else If y'_i = nothing **Then**
        return x'_i;
Else If $\rho(x'_i) \leq \rho(y'_i)$ **Then**
        return x'_i;
Else
        return y'_i;
End If
End If
End If

```
Function BEFORE(i)

If x_i = nothing Then
        return y_i;
Else If y_i = nothing Then
        return x_i;
Else If ρ(x_i) ≤ ρ(y_i) Then
        return y_i;
Else
        return x_i;
End If
End If
End If
```

In particular, either AFTER(i) or BEFORE(i) is used to determine the sibling of ϕ_i. If the node ϕ_i is a left child, its sibling appears immediately after it. Hence, if AFTER(i) does not return nothing, then the returned node is the sibling of ϕ_i, otherwise the algorithm must stop. Similarly, if the node ϕ_i is a right child, then its sibling appears immediately before it and is returned by BEFORE(i).

4.5 Updating the Lookahead Variables

In both steps 2 and 4, some lookahead variables are updated (or initialized). In this subsection, we present the procedure UPDATE, that assigns new nodes to the lookahead variables.

Whenever a node s assigned to a lookahead variable is used to form a new internal node, this lookahead variable must be updated, that is, it must receive another node s'. If it is a left-side lookahead, then it must receive either the leaf or the internal node that appears immediately before s. Otherwise, it must receive either the leaf or the internal node that appears immediately after s. If s and s' are internal nodes, the algorithm must choose the children of s' from the lookahead varibles corresponding to one level bellow. After each child of s' is chosen, the lookahead variable that stores it is updated, which leads to a recursive implementation of procedure UPDATE.

The pseudo-code for the procedure UPDATE is shown in figure 4, where the function $\rho(s)$ returns the weight assigned to the node s. The action performed by

Case 1: Updating x_i

If x_i is the first leaf node **Then** $x_i \leftarrow$ nothing
Else $x_i \leftarrow$ the leaf node before x_i

Case 2: Updating y_i

If $c_{i-1} > 0$ **Then**
  "Get the right child of the new internal node"
  $s \leftarrow$ BEFORE$(i-1)$; $w \leftarrow \rho(s)$
  If s is an internal node **Then** UPDATE(y_{i-1})
  Else UPDATE(x_{i-1})

  "Get the left child of the new internal node"
  $s \leftarrow$ BEFORE$(i-1)$; $w \leftarrow w + \rho(s)$
  If s is an internal node **Then** UPDATE(y_{i-1})
  Else UPDATE(x_{i-1})

  Form a new internal node with weight w
  Assign it to y_i; $c_{i-1} \leftarrow c_{i-1} - 1$
Else
  $y_i \leftarrow$ nothing
End If

Case 3: Updating x'_i

If x'_i is the last leaf node **Then** $x'_i \leftarrow$ nothing
Else $x'_i \leftarrow$ the leaf node before x'_i

Case 4: Updating y'_i

If $c_i > 0$ **Then**
  "Get the left child of the new internal node"
  $s \leftarrow$ AFTER$(i-1)$; $w \leftarrow \rho(s)$
  If s is an internal node **Then** UPDATE(y'_{i-1})
  Else UPDATE(x'_{i-1})

  "Get the right child of the new internal node"
  $s \leftarrow$ AFTER$(i-1)$; $w \leftarrow w + \rho(s)$
  If s is an internal node **Then** UPDATE(y'_{i-1})
  Else UPDATE(x'_{i-1})

  Form a new internal node with weight w
  Assign it to y'_i; $c_i \leftarrow c_i - 1$
Else
  $y'_i \leftarrow$ nothing
End If

Figure 4: The pseudo-code for the procedure UPDATE

the procedure UPDATE depends strongly on which lookahead variable is updated. If x_i or x'_i is updated, the procedure is simple and needs no recursive call. However, these lookahead variables must be already initialized, since the procedure uses the node previously assigned to it. In subsection 4.6, we show how to find the leaf nodes initially assigned to x_i and x'_i, for $i = 1, \ldots, h$. This initialization is performed in the step 3 of the main loop.

Observe that the action performed to update either y_i or y'_i is not so simple and may require some recursive calls to the procedure UPDATE. However, the node previously assigned to these lookahead variables are not needed, and so, the procedure UPDATE is also used to initialize y_i and y'_i.

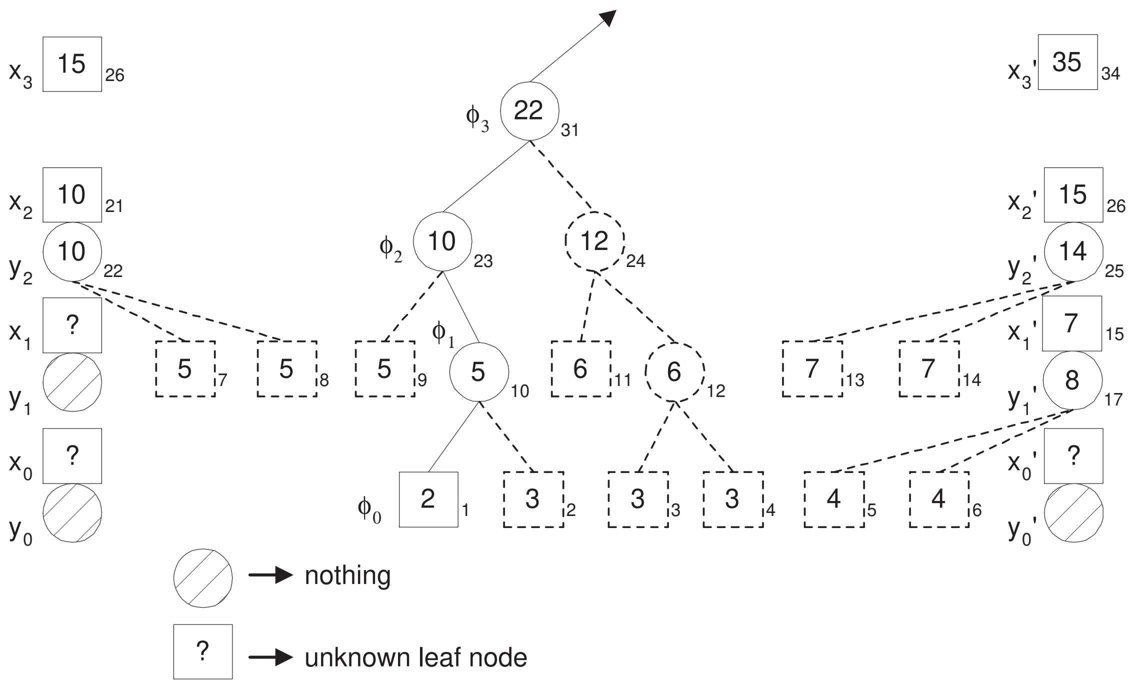

Figure 5: This figure shows the same data structure represented in figure 3.(b) before y_3 and y'_3 are initialized.

Figure 5 shows the same data structure represented in figure 3.(b) before y_3 and y'_3 are initialized. In this case, if the procedure UPDATE is called in order to initialize y_3, then this procedure returns the node with label 30. Before that, the counter c_2 is checked and the function BEFORE(2) is called in order to choose the right child of the new node. This function compares x_2 and y_2 and chooses the internal node with label 22. After that, the procedure UPDATE calls itself recursively in order to update y_2. In the next step, the algorithm chooses the left child of the new node (the leaf node with label 21) and update x_2 through another recursive call.

After the new node is assigned is y_3, the counter c_2 is decreased by one. If y'_3 is updated, the execution is similar and the node with label 32 is formed. First, the algorithm checks if the counter c_3 is greater than zero. After that, x'_2 and y'_2 are compared and the internal node with label 25 is used as the left child of the new node. The procedure calls itself in order to update y'_2 with the internal node with label 27. The right child of the new node is the leaf node with label 26. Finally, after the new node is assigned to y'_3, the counter c_3 is decreased by one.

4.6 Skiped Nodes

In this subsection, we explain the step 3.

First, the algorithm finds the first leaf node that appears after ϕ_{i+1}. It is clear that this node, which is initially assigned to x'_{i+1}, is also the first leaf node with a weight greater than the weight of ϕ_{i+1}. In addition, this node is also either the leaf node assigned to x'_i or some leaf node that appears after it. Hence, this node is searched starting from x'_i until the first leaf node with a weight greater than that of ϕ_{i+1} is found. It is immediate that the leaf node initially assigned to x_i is the last leaf node that appears before x'_i.

For example, the initialization of the lookahead variables x_3 and x'_3 is characterized in the figure 3. Observe that x'_3 receives the leaf node with label 34, which is the first leaf node with the weight greater than that of ϕ_3. The leaf node with label 26 is initially assigned to x_3. The figure 3.(b) shows an unknown leaf node beside the label x_3 because the node with label 26 is used, that is, no leaf node satisfies the semantics of the variable x_3 described in subsection 4.2. Both lookaheads appear after the node currently assigned to x'_2, from where the search starts.

Next, the algorithm assigns to c_{i+1} the number of internal nodes that appears after ϕ_{i+1} and before ϕ_{i+2} and decides if ϕ_{i+1} is a left child or a right child. This step is performed by calculating how many nodes are children of these internal nodes. Let k_i be the number of nodes that appear before ϕ_{i+1} and after its children. If k_i is odd, then one of the k_i nodes counted is the sibling of ϕ_{i+1}, and the other $k_i - 1$ nodes have $(k_i - 1)/2$ parents, which appear after ϕ_{i+1} and before ϕ_{i+2}. Otherwise, if k_i is even, then the sibling of ϕ_{i+1} appears after it and exactly $k_i/2$ internal nodes appears after ϕ_{i+1} and before ϕ_{i+2}. Hence, c_{i+1} is initialized with $\lfloor k_i/2 \rfloor$ and the parity of k_i

determines if ϕ_{i+1} is a left child or a right child. In order to calculate the value of k_i, the algorithm sums the number of internal nodes and the number of leaf nodes that appear before ϕ_{i+1} and after its children. The number of leaf nodes is easily obtained because the weights of the leaf nodes are stored is an array. It is given by $i_r - i_l + 1$, where i_r and i_l are respectively the indexes of the leaf nodes assigned to x_{i+1} and x'_i. Some special care must be taken with the cases where either x_{i+1} or x'_i is nothing. Furthermore, the number of internal nodes is given by $c_i + 1$ if y'_i is not nothing, and is zero otherwise.

For example, let us consider the case of figure 5 (see also the whole Huffman tree in the figure 3.(a)). In this case, we show how to calculate the value of c_3 and how to decide if the node ϕ_3 is a left child or a right child. Observe that $k_2 = 6$ because there is one leaf node (with label 26) and 5 internal nodes (with labels 25, 27, 28, 29 and 30) between ϕ_2 and ϕ_3. Observe also that $c_2 = 4$. These six nodes are children of the three internal nodes with labels 32, 33 and 35, that appear after the node ϕ_3 and before its parent. Hence, $c_3 = 3$. Since k_2 is even, ϕ_3 is a left child.

4.7 Stolen Internal Nodes

Now, let us discuss a special case that may occur during the execution of the Lazy-Height algorithm. In subsection 4.2, we state that y_k stores the last internal node that is not used and that appears between the nodes ϕ_k and ϕ_{k-1}, for $h = 1, \ldots, i$, where i is the height of the last ancestor of the first leaf node that is formed. Similarly, y'_k stores the first internal node that is not used and that appears between the node ϕ_k and ϕ_{k+1}. Hence, if an internal node s is the only one that is not used and that appears between ϕ_k and ϕ_{k+1} for some value of k, then this node must be assigned to both y'_k and y_{k+1}. Since an internal node cannot be formed twice, the node s is assigned either to y'_k or to y_{k+1} and one of the lookahead variables receives nothing.

For example, observe in figure 3.(b) that the internal node with label 17 must be assigned to both y'_1 and y_2. Now, suppose that this node is assigned to y_2 and that y'_1 stores nothing. When the node with label 28 (see the figure 3.(a)) is formed, the function AFTER(1) must return the node with label 17. However, this function cannot return this node because it is assigned y_2 instead of y'_1, that is, this node is an *stolen* internal node.

Observe that it is not possible to decide which lookahead variable must receive each new internal node at the time it is formed. The lookahead variables receive new nodes as the old ones are used. Furthermore, we do not allow that an internal node is formed twice in order to simplify the time analysis of the algorithm. Hence, the following fix is needed in the function AFTER(i): if y'_i is nothing and y_{i+1} is not nothing, then act as if the node assigned to y_{i+1} was also assigned to y'_i. In this case, if y'_i is updated then y_{i+1} receives nothing. Similarly, the function BEFORE(i) must also be fixed. If y_i is nothing and y'_{i-1} is not nothing, then act as if the node assigned to y'_{i-1} was also assigned to y_i. In this case, if y_i is updated then y'_{i-1} receives nothing.

5 The LazyHuff Algorithm

Though the LazyHeight algorithm only finds the height of a Huffman tree, in this subsection, we show that this algorithm can be slightly modified in order to calculate the Huffman code for a sorted input list of weights W. As we describe in section 1, the output is generated in runlength format, that is, the algorithm provides the number of leaf nodes at each level of the Huffman tree. The modified algorithm is called the *LazyHuff* algorithm.

Unfortunately, some extra data must be stored in order to provide the Huffman code. First, let us define a structure for this extra data. Let s be an internal node formed by the algorithm. We define a list of integer values $\Delta_s = [\delta_1, \ldots, \delta_{h_s-1}]$, where h_s is the height of the tree rooted by s and δ_k is the number of internal nodes that are positioned in the level k in this tree, for $k = 1, \ldots, h_s - 1$. If $h_s = 1$ then Δ_s is *empty*. If ϕ_i is the ancestor of the first leaf node most recently formed, a corresponding list Δ_{ϕ_i} is maintained. For each internal node assigned to a lookahead variable y_k or y'_k, for $k = 1, \ldots, i$, the algorithm also maintains a corresponding list Δ_{y_k} or $\Delta_{y'_k}$.

Now, we define two basic operations under these lists: insertion and melding. Let $\Delta_1 = [\delta_1, \ldots, \delta_{h_1-1}]$. If an integer value is inserted in the list Δ_1, then the labels of all elements in the list are incremented, such that the resulting list is $[\delta_2, \ldots, \delta_{h_1}]$. After that, the new element is assigned to δ_1 and h_1 is incremented. Furthermore, let $\Delta_2 = [\delta'_1, \ldots, \delta'_{h_2-1}]$ and $\Delta_m = [\delta^*_1, \ldots, \delta^*_{h_m-1}]$, where the list Δ_m is obtained by melding Δ_1 and Δ_2. Without loss of generality, we also assume that $h_2 \leq h_1$,. In this case, the list Δ_m is defined as follows:

- $h_m = h_1$;
- if $k < h_2$ then $\delta^*_k = \delta_k + \delta'_k$, otherwise $\delta^*_k = \delta_k$, for $k = 1, \ldots, h_m - 1$.

Observe that this melding operation requires only h_2 operations. For example, if the lists $[3, 2, 4, 1]$ and $[2, 3, 3, 8, 1, 2]$ are melded, then the list $[5, 5, 7, 9,\ 1, 2]$ is obtained through four operations.

Finally, we describe how these lists are maintained. Let s be the next internal node formed by the LazyHuff algorithm. The list Δ_s corresponding to the new node s is initially empty. If the left child of s is an internal node, then its corresponding list and Δ_s are melded and the resulting list is assigned back to Δ_s. If the right child of s is an internal node, then a similar action is performed. At last, an integer value equal to the number of internal nodes that are children of s is inserted in the list Δ_s.

After the whole tree is constructed, only the list $\Delta_{\phi_h} = [\delta_1, \ldots, \delta_{h-1}]$, which corresponds to the root node, is remaining. Observe that each internal node in the tree has exactly two children, where each child is either a leaf node or another internal node. Since all the nodes that are positioned in the level one in the tree are children of the root node, there are exactly $2 - \delta_1$ leaf nodes in this level. Similarly, all the nodes in level k, for $k = 2, \ldots, h-1$, are children of any internal node in level $k-1$, and so there are $2\delta_{k-1} - \delta_k$ leaf nodes in this level. Finally, there are $2\delta_{h-1}$ leaf nodes in level h.

6 Time and Space Requirements

In this section, we present some considerations on the time and space requirements of both the LazyHeight and the LazyHuff algorithms, respectively described in sections 4 and 5. Let us refer to the height of a Huffman tree as l_1 instead of h.

Theorem 1 *Both the LazyHeight and the LazyHuff algorithms require $O(n)$ time.*

Proof: Observe that the main loop described in subsection 4.3 performs only l_1 iterations. However, an iteration of the main loop cannot be considered a unit cost operation. Hence, we bind the cost for the calls to the procedure UPDATE, which is described in subsection 4.5, and the cost of searching the leaf node initially assigned to each lookahead variable x'_i, for $i = 1, \ldots, l_1$, which is described in subsection 4.6.

Let us assume that the searching method is a simple linear search. Since the search for the leaf node initially assigned to x'_{i+1} starts from the leaf node currently assigned to x'_i, for $i = 0, \ldots, l_1 - 1$, then at most n comparisons are done during one complete execution of the algorithm.

Now, we consider the calls to the procedure UPDATE. Observe that if either x_i or x'_i is updated then no recursive call is performed. This is also observed if either y_i or y'_i is updated and the corresponding counter is zero. In these cases, we consider the cost of calling the procedure UPDATE is included in the cost of the action which performs the call. If either y_i or y'_i is updated and the corresponding counter greater then zero, then a new internal node is formed. Since the algorithm does not form an internal node twice, this last case corresponds to exactly $n-1$ calls to the procedure UPDATE.

However, the LazyHuff algorithm maintains a list of integer values corresponding to each internal node formed, what is described in section 5. Now, we bind the total cost of both the operations defined for these lists: insertion and melding. It is clear that the sum of the integer values contained in all the lists maintained by the algorithm is equal to the number of internal nodes currently formed and deleted, which is bounded above by $n-1$. Since there is no integer value smaller than one in the lists and the sum of all integer values never decreases, the number of insertions performed is also bounded above by $n-1$. Moreover, each unit cost operation performed during the melding of two lists causes that one integer value is deleted. In fact, each pair of melded integer values is replaced by another corresponding integer value in the resulting list, such that the sum of all integer values remains unchanged. Hence, the total cost of all melding operations performed by the algorithm is bounded above by $n-l_1$, since the remaining list, which corresponds to the root node, has l_1-1 integer values.

Finally, we consider that the remaining actions performed during each iteration of the main loop costs one unit, that is, all the remaining work is done with a cost equal to l_1. ■

Theorem 2 *The LazyHeight algorithm requires an extra space $O(l_1)$, what increases to $O(\min\{l_1^2, n\})$ for the LazyHuff algorithm.*

Proof: First, the number of lookahead variables and the number of counters maintained by the algorithm are both proportional to the number of ancestors of the first leaf node formed, which is bounded above by l_1+1 (including the root node and the first leaf node). Moreover, each recursive call to the procedure UPDATE updates a lookahead variable corresponding to a node that is one level down in the tree, which means that the program's stack size required is bounded above by some value proportional to the height of the tree.

However, the LazyHuff algorithm maintains a list of at most $i-1$ integer values corresponding to each lookahead variable y_i or y'_i, for $i = 1, \ldots, l_1$. In this case, the total number of integer values is bounded above by $l_1^2 - l_1$. Furthermore, the total number of integer values cannot be greater than the number of internal nodes formed, which is bounded above by $n-1$. ■

Observe that the memory usage of this method is particularly small if a very large alphabet is handled. For example, compressing the TREC data with a word-based model generates an alphabet with about one million words, where the greatest codeword is 30 bits long [MT99]. If this collection were computed by LazyHufff algorithm then at most $30^2 - 30 = 870$ integer values would appear simultaneously in the lists of integers, and at most 124 lookahead variables and 31 counters would be maintained in memory.

7 Conclusions

The algorithm described throughout this technical report is a space-efficient method for the construction of minimum-redundancy prefix-free codes.

Furthermore, we believe that this method may be used to improve the space requirements of some approximative algorithms for length-restricted coding that depend on the construction of Huffman trees [ML96, ML97].

References

[Bur93] Michael Buro. On the maximum length of Huffman codes. *Information Processing Letters*, 45(5):219–223, 2 April 1993.

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[Gai93] J. L. Gailly. *Gzip program and documentation*, 1993. Available by anonymous ftp from prep.ai.mit.edu:/pub/gnu/gzip-*.tar.

[HL90] Daniel S. Hirschberg and Debra A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, April 1990.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, September 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[MK95] Alistair Moffat and Jyrki Katajainen. In-place calculation of minimum-redundancy codes. In Selim G. Akl, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, 4th International Workshop*, volume 955 of *Lecture Notes in Computer Science*, pages 393–402, Kingston, Ontario, Canada, 16–18 August 1995. Springer.

[ML96] Ruy L. Milidiú and Eduardo S. Laber. Warm-up algorithm: A lagrangean construction of length restricted huffman codes. Technical Report 15, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brasil, January 1996.

[ML97] Ruy L. Milidiú and Eduardo S. Laber. Improved bounds on the inefficiency of length-restricted prefix codes. Technical Report 33, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brasil, January 1997.

[MT99] A. Moffat and A. Turpin. Efficient construction of minimum-redundancy codes for large alphabets. *To appear on Transactions on Information Theory*, 1999.

[vL76] J. van Leeuwen. On the construction of Huffman trees. In S. Michaelson and R. Milner, editors, *Third International Colloquium on Automata, Languages and Programming*, pages 382–410, University of Edinburgh, 20–23 July 1976. Edinburgh University Press.

[ZM95] Justin Zobel and Alistair Moffat. Adding compression to a full-text retrieval system. *Software—Practice and Experience*, 25(8):891–903, August 1995.