Improved Analysis of FGK Algorithm

Ruy Luiz Milidiú

milidiu@inf.puc-rio.br

Eduardo Sany Laber

laber@inf.puc-rio.br

Artur Alves Pessoa

artur@learn.fplf.org.br

PUC-RioInf.MCC39/97 September, 1997

Abstract: An important issue related to coding schemes is their compression loss. A simple measure ϵ of the compression loss due to a coding scheme different than Huffman coding, is defined by

$$\epsilon = A_C - A_H$$

where A_H is the average code length of a static Huffman encoding and A_C is the average code length of an encoding based on the compression scheme C.

When the scheme C is the FGK algorithm, Vitter[12] conjectured that $\epsilon \leq K$ for some real constant K. Here, we use an amortized analysis to prove this conjecture. We show that $\epsilon < 2$. Furthermore, we show through an example that our bound is asymptotically tight. This result explain the good performance of FGK that many authors have observed through practical experiments.

Keywords: Dynamic Huffman Codes, Amortized Analysis, Data Compession

Resumo: Um importante aspecto relacionado a um esquema de codificação é a sua perda de compressão. A medida ϵ da perda de compressão de um esquema de codificação é definida por

$$\epsilon = A_C - A_H$$

onde A_H é o comprimento médio de uma codificação de Huffman e A_C é o comprimmento médio de uma codificação baseada em um esquema de compressão C.

Quando o esquema de compressão é o algoritmo FGK, Vitter[12] conjeturou que $\epsilon \leq K$ para alguma constante real K. Neste trabalho, utilizamos uma técnica de análise amortizada para provar esta conjectura. Além disto, mostramos através de um exemplo que nossa cota é justa. Este resultado explica o bom desempenho do FGK que alguns autores haviam constatado através de resultados experimentais.

Palavras-Chave: Códigos de Huffman Dinâmicos, Análise Amortizada, Compressão de Dados.

1 Introduction

A Static Huffman Code for a given source alphabet $a_1, \ldots, a_n$ is a prefix code generated by Huffman's algorithm [5, 7, 4, 10]. Such a code minimizes $\sum_{i=1}^{n} w_i l_i$ among all prefix codes, where w_i is the weight of a_i and l_i is the number of bits used to encode a_i.

Huffman's algorithm uses a greedy strategy to construct a binary code tree with n leaves, where each leaf corresponds to an alphabet symbol. The codes can be easily constructed from the tree by assigning a bit with value "0" to all edges that connect a node to its left child and a bit with value "1" to all edges that connect a node to its right child. Then, the code for the symbol a_i is given by the bit string associated to the path that connects the root of the tree to the leaf that corresponds to a_i.

For several applications, the symbol weights are estimated by parsing the message to be encoded counting the number of occurrences of each source alphabet symbol. However, for data communication purposes, the parsing required by the Static Huffman Algorithm represents a big disadvantage. That is because the data must be transmitted on-line. As soon as the symbol arrives to the transmitter, it must be encoded and transmitted to the receptor. For these situations, adaptive Huffman codes have been largely used. This method determines the mapping from symbol alphabets to codewords based upon a running estimate of the alphabet symbol weights. The code is adaptive, just changing to remain optimal for the current estimates.

Adaptive Huffman coding was first conceived independently by Faller and Gallager [1, 2]. Knuth [3] proposed an efficient implementation to generate such codes. The resulting algorithm is referred as the FGK algorithm. Another version for the adaptive Huffman coding is the Λ algorithm described by Vitter [12, 13]. Both algorithms require $O(l)$ time for each encoding and decoding operation, where l is the current length of the codeword assigned to the symbol to be transmitted.

An important issue related to coding schemes is their compression loss. A simple measure ϵ of the compression loss due to a coding scheme different than Huffman coding, is defined by

$$\epsilon = A_C - A_H$$

where A_H is the average code length of a static Huffman encoding and A_C is the average code length of an encoding based on the compression scheme C.

Milidiú and Laber [9] proved that if the scheme C is the length restricted Huffman coding, then $\epsilon < 1/\psi^{L-\lceil \log n \rceil -1}$, where n is the size of the alphabet, ψ is the golden ratio $(1+\sqrt{5})/2$ and L is the length restriction. That result explains the usual small compression loss observed on practical uses of length restricted codes [14].

For dynamic Huffman encoding, Vitter [12] proved that the Λ algorithm uses no more than $S_t + t - 2n$ bits to encode a message of size t, where S_t is the number of bits required by the static Huffman method. This result implies that $\epsilon < 1$ for the Λ algorithm. Vitter [12] also proved that the total number of bits D_t transmitted by the FGK algorithm for a message with t symbols is bounded below by $S_t - n + 1$, and bounded above by $2S_t + t - 4n + 2$. Furthermore, he conjectured that D_t is bounded above by $S_t + O(t)$, that is, $\epsilon \leq K$ for some real constant K. Here, we prove this conjecture by showing that $D_t \leq S_t + 2t - 2k - \lceil \log Min(k+1, n) \rceil$, where k is the number of distinct symbols in the message. This result assures that $\epsilon < 2$ for the FGK algorithm. We also present an example where $D_t = S_t + 2t - 2k - 3\lfloor (t-k)/k \rfloor - \lceil \log(k+1) \rceil$, what shows that the proposed bound is asymptotically tight. These results explain the good performance of FGK that some authors have observed through practical experiments [3, 12].

The paper is organized as follows. In section 2, we present basic concepts of the FGK algorithm. In section 3, we show our improved analysis of the algorithm. In section 4, we prove that our bound is asymptotically tight. Finally, in section 5, we comment on our findings.

2 The FGK Algorithm

The basis for the FGK algorithm is the Sibling Property, defined by Gallager [2]. A binary code tree with p leaves has the sibling property if an only if

> (i) the p leaves have nonnegative weights $w_1, \ldots, w_p$, and the weight of each internal node is the sum of the weight of its children; and
>
> (ii) the $2p-1$ nodes (external and internal) can be numbered in nondecreasing order by weight, so that nodes $2j-1$ and $2j$ are siblings, for $j = 1, \ldots, p$, and their common parent has a higher number.

Gallager proved that a binary prefix code is a Huffman code if and only if the code tree has the sibling property. In the FGK algorithm, both sender and receiver maintain dynamically changing Huffman code trees. The leaves of the code tree represent the alphabet symbols. The weight of each leaf represents the frequency occurrences of its corresponding symbol in the message. At any time during the coding-transmission-reception-decoding cycle, exactly k of the n possible alphabet symbols have occurred in the message.

Initially, the code tree consists of a single leaf node, called the 0-node. The 0-node is a special node used to represent the $n-k$ unused symbols. For each symbol transmitted, both parties must increment the corresponding weight and update the code tree to maintain the sibling property. At the point in time where t symbols have been transmitted, k of them are distinct, with $k < n$, the current tree is a legal Huffman code tree with $k+1$ leaves, one for each of the k symbols and one for the 0-node. If the (t+1)th symbol $a_{i_{t+1}}$ is one of the k already seen, the algorithm transmits the $a_{i_{t+1}}$'s current code, increments the appropriate counter and updates the tree. If an unused symbol occurs, the 0-node is split to create a pair of leaves, one for $a_{i_{t+1}}$, and a sibling which is the new 0-node. Again, the tree is updated. In this case, the code for the 0-node is sent followed by a code that indicates which of the $n-k$ unused messages has appeared.

The tree's update can be easily described as a two-phase process, although for implementation purposes both phases can be combined into one. In the first phase the current Huffman tree is transformed into another Huffman tree. The first phase begins with the leaf corresponding to $a_{i_{t+1}}$ as the current node. We repeatedly interchange the current node, along with the subtree rooted there, with the node of the highest number with the same weight, and make the parent of the latter node the new current node. The interchanges stop when the current node is the root of the tree. The sibling property assures that the tree obtained at the end of the first phase is a Huffman tree. In the second phase, the algorithm increments by an unity the weight of $a_{i_{t+1}}$ and the weights of all its ancestors in the tree obtained at the end of the first phase. The updates of the first phase assure that the current tree still has the sibling property for the new weights.

Figure 1.(a) shows the Huffman tree maintained by the FGK algorithm immediately before the arrival of the 140th symbol. The number inside each node corresponds to its weight, while the number outside is related to the sibling property. The 140th symbol corresponds to the shaded leaf in figure 1.(a). When this symbol arrives, the procedure Update interchanges the leaf with outside number 6 and the internal node with outside number 9. After that, the current node becomes the node with number 14. This node is not interchanged and the current node becomes the node with outside number 17. This node is interchanged with the node with outside number

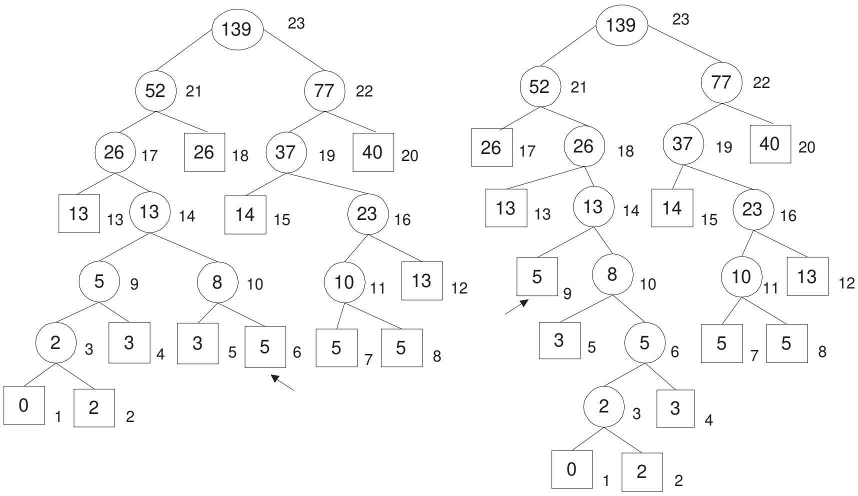

Figure 1: Figure (a) shows the Huffman tree maintained by FGK immediately before of the arrival of the 140 symbol. Figure (b) shows a the Huffman tree maintained by FGK immediately after the arrival of the 140-th symbol.

18. Next, the node with outside number 21 becomes the current node and finally, the root of the tree becomes the current node. Figure 1.(a) shows the resulting Huffman tree. A pseudo-code for the procedure Update is shown in figure 2.

3 Improved Bound for FGK Compression

3.1 Vitter's Approach

We follow the same notation used by Vitter [12]. Let $a_1, \ldots, a_n$ be the source alphabet, and let $M_t = a_{i_1}, a_{i_2}, \ldots, a_{i_t}$ be a message of size $t \geq 0$. Let also k denote the number of distinct symbols in M_t.

Some definitions that will be useful throughout our analysis are listed below.

```
Procedure Update;
Phase 1:
    q:=leaf node corresponding to a_{i_{t+1}} ;
    If (q is the 0-node) and (k < n - 1) then
        Replace q by a parent 0-node with two leaf 0-node children,
number
        in order left child, right child, parent ;
        q:= parent 0-node just created;
    If q is the sibling of the 0-node then
        Interchange q with the highest numbered leaf of the same
weight ;
        q:= q.parent ;
    while q is not the root of the Huffman tree do
        Interchange q with the highest numbered node of the same
weight;
        q:= q.parent ;
Phase 2:
    q:=leaf node corresponding to a_{i_{t+1}};
    while q is not the root of the Huffman tree do
        q.weight := q.weight + 1 ;
        q:= q.parent ;
```

Figure 2: The FGK algorithm.

T_{t-1}: The Huffman tree stored by FGK before a_{i_t} is processed.

T_t : The Huffman tree stored by FGK after a_{i_t} is processed.

y_{i_t}: Leaf that corresponds to a_{i_t}.

l_{i_t}: Level of y_{i_t} in T_{t-1}.

l'_{i_t}: Level of y_{i_t} in T_t.

h_t: l_{i_t} - l'_{i_t}.

w_{i_t}: The weight of y_{i_t} in T_{t-1}.

In figure 3, we illustrate the previous concepts. Figure 3.(a) shows the Huffman tree T_{34} stored by the FGK algorithm before the processing of $a_{i_{35}}$. Figure 3.(b) shows the Huffman tree T_{35} obtained after the processing of $a_{i_{35}}$. The number inside each node indicates its weight and the number outside indicates the ordering number related with the sibling property of the current tree. The leaf $y_{i_{35}}$ that corresponds to the 35-th symbol in the message is shaded in the two trees. In the tree of figure 3.(a), $y_{i_{35}}$ is on level $l_{i_{35}} = 5$ and in the tree of figure 3.(b), $y_{i_{35}}$ is on level $l'_{i_{35}} = 3$. Hence, $h_{35} = 5 - 3 = 2$ for this case.

It must be observed that if $k < n$ and the 0-node is accessed in the tree T_{t-1}, then l_{i_t} corresponds to the level of the 0-node in the tree T_{t-1} and l'_{i_t} corresponds to the level of the parent of the 0-node in T_t.

Let us use S_t to denote the total number of bits used by a static Huffman encoding of M_t. Observe that $S_0 = 0$. The *incremental* cost s_t is defined by the following equation

$$s_t = S_t - S_{t-1} \tag{1}$$

From the above definitions, it follows immediately that

$$S_t = \sum_{j=1}^{t} s_j \tag{2}$$

In a similar way, we use d_t to denote the number of bits used for encoding a_{i_t} through the FGK algorithm. Clearly, we have that $d_t = l_{i_t}$, where l_{i_t} is the level of the leaf that corresponds to a_{i_t} in T_{t-1}. If a_{i_t} has not appeared in M_{t-1}, then a_{i_t} is associated to the 0-node of T_{t-1}. Furthermore, we use D_t to denote the total number of bits used by the FGK algorithm to encode the message M_t.

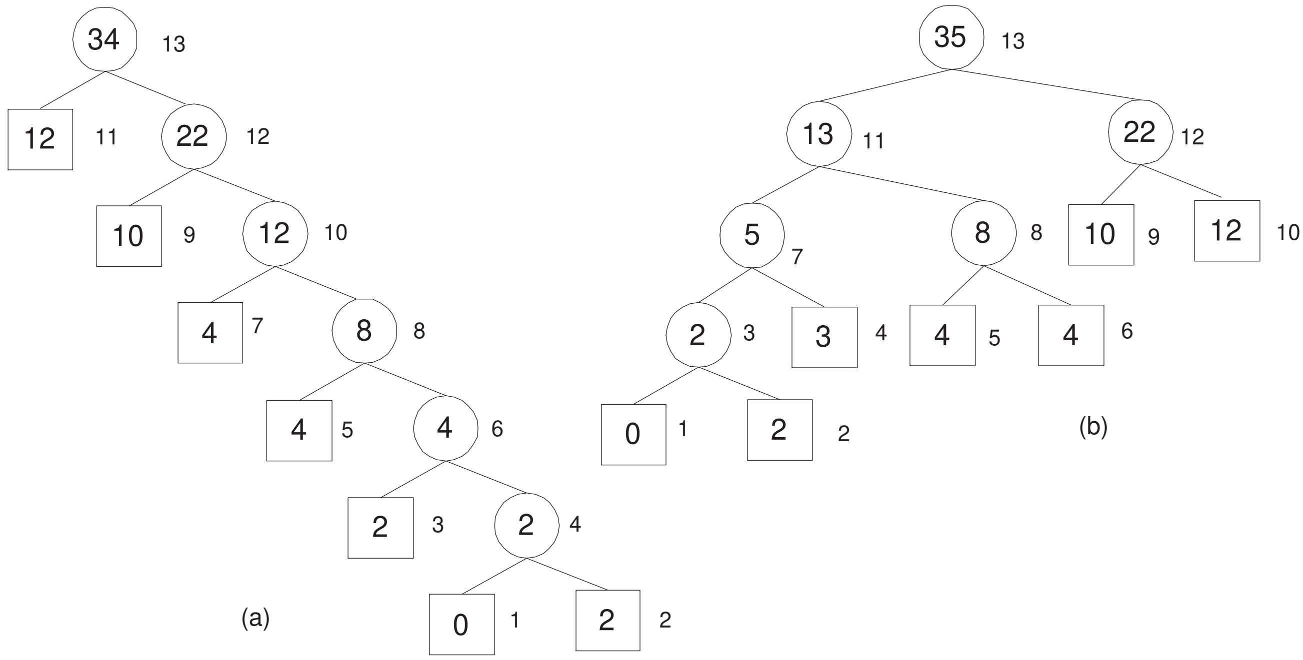

Figure 3: For $t = 35$, figure (a) shows the tree T_{t-1}, whereas figure (b) shows the tree T_t. The node y_t is shaded in both trees. In T_{t-1} it is at level 5 and in T_t it is at level 3.

Observe that

$$D_t = \sum_{j=1}^{t} d_j \tag{3}$$

Now, our interest is on bounding the difference between D_t and S_t. In order to get such a bound, we must analyze the difference between d_i and s_i for $i = 1, \dots, t$, since

$$D_t - S_t = \sum_{i=1}^{t} (d_i - s_i) \tag{4}$$

A fundamental result for our analysis is the theorem 3.2 proved by Vitter [12]. This theorem shows how to evaluate $d_t - s_t$. The formulation that we present below is slightly weaker than that of Vitter.

Theorem 1 (Vitter's Theorem) *The following four statements describe the behavior of the difference $d_t - s_t$:*

(i) if $t = 1$, then $d_t - s_t = h_t$;

(ii) if $t > 1$ and y_{i_t} is the 0-node in T_{t-1}, then $d_t - s_t \leq h_t - 1$;

(iii) if T_{t-1} has a 0-node and w_{i_t} is the minimum strictly positive weight in T_{t-1}, then $d_t - s_t = h_t + 1$;

(iv) if either T_{t-1} does not have a 0-node or if w_{i_t} is not the minimum strictly positive weight in T_{t-1}, then $d_t - s_t = h_t$.

The last theorem suggests the study of the value of $\sum_{j=1}^{t} h_j$. Another useful result stated by Vitter [12] is the following proposition.

Proposition 1 *Two nodes with the same weight cannot be more than one level apart in the tree, except if one is the sibling of the* 0*-node. In this last case, the nodes can be at most two levels apart.*

See Vitter [12]. □

3.2 Height variation in FGK

In this subsection, we relate the height variation of the trees constructed by the FGK algorithm with the level variation h_j. The height variation must be understood as the difference between the height of the Huffman tree obtained by the FGK algorithm after and before the execution of procedure Update. The relation proved in this section is the key for bounding the value of $\sum_{j=1}^{t} h_j$.

Here, we assume that either T_{t-1} does not have a 0-node or w_{i_t} is greater than the weight of the 0-node sibling. Let us define K_t as a subset of nodes of T_{t-1}. A node y in T_{t-1} belongs to K_t if and only if it satisfies at least one of the conditions below:

(i) y is the sibling of y_{i_t};

(ii) the weight of y is greater than or equal to w_{i_t} and smaller than the minimum between $2w_{i_t}$ and the weight of y_{i_t}'s parent in T_{t-1}.

Figure 4 corresponds to $t = 132$. Observe that the black arrow points to node y_{i_t}. Observe also that the nodes of K_t are shaded.

Let us define the concepts of ancestor and descendant in a tree. Every node is an ancestor of itself. If s is a node, then the parent of any ancestor of s is also an ancestor of s. A node s_1 is a descendent of another node s_2 if and only if s_2 is an ancestor of s_1. We say that a node s_1 is a strict descendant (ancestor) of a node s_2 if and only if s_2 is an ancestor (descendant) of s_1 and s_2 is different from s_1.

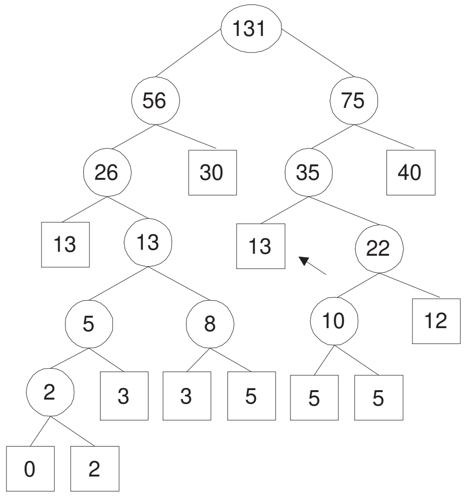

Figure 4: The figure shows a Huffman tree T_{t-1} for $t = 132$. The black arrow points to y_t. The nodes of the set K_t are shaded.

Now, choose y^s as one of the nodes of T_{t-1} with minimum weight and maximum level. If T_{t-1} has a 0-node, then y^s is this node. Let us define y^m as the node with maximum weight, among the ancestors of y^s with weight smaller than w_{i_t}. In figure 4, y^s is the 0-node and y^m is the internal node with weight equal to 5.

Proposition 2 *y^s is a descendant of some node in K_t.*

If y^m is the sibling of y_{i_t}, we are done since y^m satisfies (i). If y^m is not the sibling of y_{i_t}, then its parent satisfies condition (ii). We can prove this by contradiction. In effect, assume that the weight of y^m's parent is smaller than w_{i_t}. In this case, we contradict the maximality assumption of y^m among the ancestors of y^s. Hence, the weight of y^m's parent is greater than or equal to w_{i_t} Now, assume that the weight of y^m's parent is greater than or equal to the minimum between $2w_{i_t}$ and the weight of y_{i_t}'s parent. If the weight of y^m's parent is greater than or equal to $2w_{i_t}$, then the sibling of y^m must have weight greater than w_{i_t}. This contradicts the sibling property since either the number of y_{i_t} is greater than that of y^m's sibling or it is smaller than the number of y^m. If the weight of y^m's parent is greater than or equal to the weight of y_{i_t}'s parent and y_{i_t}'s parent weight is smaller than $2w_{i_t}$, then the sibling property is also contradicted. That happens since either the weight of y^m or the weight of y^m's sibling would be greater than the weight of y_{i_t}'s sibling. $\square$

Proposition 3 *During the tth call to procedure Update, there is no interchange between the strict descendants of the nodes in K_t.*

Let y be a descendant of some node in K_t. It is enough to prove that y's number is smaller than y_{i_t}'s number. In this case, the procedure Update does not interchange the node y. We divide the proof into two cases.

First, let us examine the case where y does not belong to K_t. In this case, the weight of y is smaller than w_{i_t}.

Secondly, let us examine the case where y belongs to K_t. Since y is a strict descendant of some node in the set K_t, then its parent also belongs to K_t. Hence, the weight of y must be equal to w_{i_t} and its sibling must have weight smaller than w_{i_t}, otherwise its parent would not belongs to K_t. Furthermore, y can not be y_{i_t} itself, otherwise its parent would not belong to K_t. Hence, the number of y_{i_t} must be greater than the number of y. □

In the next proposition, we relate the level of y_t with the levels of the nodes that belong to K_t.

Proposition 4 *All the nodes of K_t are at levels greater than or equal to $l_{i_t} - 1$ in T_{t-1} and are at levels smaller than or equal to $l'_{i_t} + 1$ in the tree T_t.*

First, we prove the result concerning T_{t-1}. By definition, the weight of any node of K_t is smaller than the weight of y_{i_t}'s parent. Since T_{t-1} is a Huffman tree, then all the nodes of K_t must be at levels greater than or equal to y_{i_t}'s parent level, that is, level $l_{i_t} - 1$. Therefore, the result holds.

Now, we prove the result concerning the tree T_t. For the nodes of K_t with weights greater than w_{i_t} the result holds, otherwise T_t would not be a Huffman tree for M_{t-1}. For the nodes with weights equal to w_{i_t}, proposition 1 assures the correctness since T_t is a Huffman tree for M_{t-1}. A third case can occur if the sibling of y_{i_t} in T_{t-1} has weight smaller than y_{i_t}. However, the FGK algorithm assures that this node will be the sibling of a node with weight w_{i_t} in T_t, what lead us to previous case. □

Figure 5 shows an example where the bounds of proposition 4 are tight. The tree in figure 5.(a) is a T_{t-1} tree for $t = 132$. The black arrow in that figure points to node y_{i_t}. The node y_{i_t} is at level 5, that is $l_{i_t} = 5$. The nodes of set K_t are shaded and its node with smallest level is at level 4. Figure 5.(b) shows the tree T_t. In that tree, the node y_t is at level 4 and the node of K_t with the greatest level is at level 5.

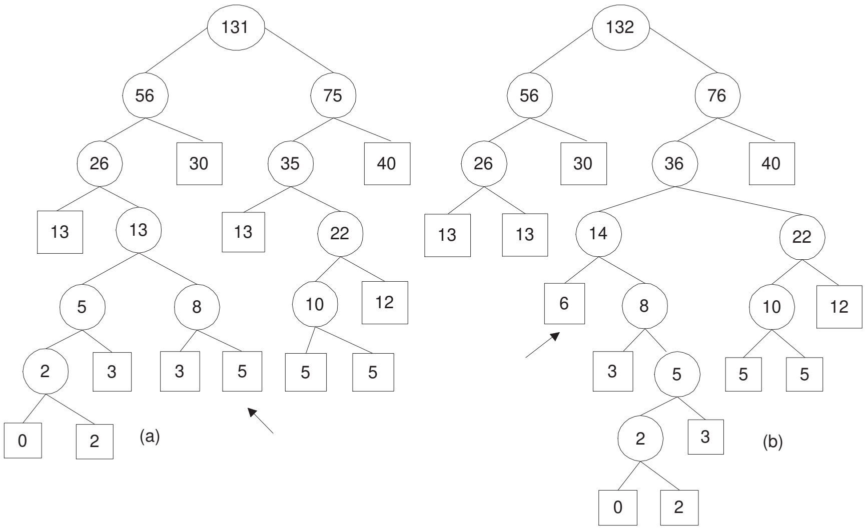

Figure 5: Figure (a) shows the Huffman tree T_{t-1} and figure (b) shows the Huffman tree T_t. The nodes that belong to K_t are shaded in figure (a). The shaded nodes in figure (b) correspond to the shaded nodes in figure (a).

In the next proposition, we show the major result of this subsection. It relates h_t with the height variation $H(T_t) - H(T_{t-1})$, where $H(T_t)$ denotes the height of the Huffman tree T_t.

Proposition 5 *If either T_{t-1} does not have a 0-node or w_{i_t} is greater than the weight of the 0-node sibling, then $H(T_t) - H(T_{t-1}) + h_t \leq 2$.*

By proposition 2, we can assume that there is an ancestor y of the node y^s that belongs to K_t in the tree T_{t-1}. Let l and l' be, respectively, the levels of y in T_{t-1} and in T_t. Furthermore, let $H(y)$ be the height of the subtree of T_{t-1} rooted at y. Clearly, we have that

$$H(T_{t-1}) = H(y) + l \tag{5}$$

On the other hand, proposition 4 assures that

$$l_{i_t} - l \leq 1 \tag{6}$$

Since the subtrees rooted at the nodes of K_t remain the same after executing procedure Update, then we have that

$$H(T_t) = H(y) + l' \tag{7}$$

By proposition 4, we have that

$$l'_{i_t} - l' \geq -1 \tag{8}$$

By subtracting (5) from (7), we obtain

$$H(T_t) - H(T_{t-1}) = l' - l \tag{9}$$

By performing some algebra with inequalities (8) and (6), we have $l' - l \leq l'_{i_t} - l_{i_t} + 2$, and hence $l' - l \leq 2 - h_t$. Replacing this last inequality in (9), we obtain

$$H(T_t) - H(T_{t-1}) \leq 2 - h_t$$

□

3.3 The Proof of Vitter's Conjecture

In order to bound the value of $\sum_{j=1}^{t} h_j$, we use an amortized analysis [11]. The potential function Φ is defined as the height of the Huffman tree maintained by the FGK algorithm. Our choice is strongly motivated by the results presented in the previous section. The amortized cost $c(t)$ is given by

$$c(t) = \Phi(t) - \Phi(t-1) + h_t$$

where $\Phi(t)$ is the height of T_t and $\Phi(t-1)$ is the height of T_{t-1}. Now, we give some upper bounds for $c(t)$. Let us consider the following four exclusive cases:

Case 1: y_{i_t} is the 0-node in T_{t-1};

Case 2: T_{t-1} has a 0-node and w_{i_t} is the minimum strictly positive weight in T_{t-1};

Case 3: T_{t-1} has a 0-node and w_{i_t} is the minimum positive weight in T_{t-1} but not strictly;

Case 4: Either T_{t-1} does not have a 0-node or w_{i_t} is greater than the weight of the 0-node sibling.

Observe that case 1 covers cases (i) and (ii) of theorem 1, case 2 covers case (iii), whereas cases 3 and 4 together cover case (iv). First, let us examine case 1.

Lemma 1 *If y_{i_t} is the 0-node in T_{t-1}, then $c(t) = 1$.*

In T_{t-1}, we have $\Phi(t-1) = l_{i_t}$. Since the 0-node is splitted to form two other nodes, we have that $\Phi(t) = l'_{i_t} + 1$. Hence, it follows that

$$c(t) = \Phi(t) - \Phi(t-1) + l_{i_t} - l'_{i_t} = 1$$

□

Now, we examine case 2.

Lemma 2 *If T_{t-1} has a 0-node and w_{i_t} is the minimum strictly positive weight in T_{t-1}, then $c(t) = 0$.*

Since T_{t-1} has a 0-node and w_{i_t} is the minimum strictly positive weight in T_{t-1}, then y_{i_t} must be the 0-node sibling in T_{t-1}, otherwise we contradict the sibling property. Hence,

$$l_{i_t} = \Phi(t-1) \tag{10}$$

Since T_t is also a Huffman tree for the alphabet of the message M_{t-1}, then we must have

$$l'_{i_t} = \Phi(t) \tag{11}$$

Hence, by subtracting (10) from (11) we obtain that $c(t) = 0$. □

The bound for case 3 is presented in the following lemma.

Lemma 3 *If T_{t-1} has a 0-node and w_{i_t} is the minimum positive weight in T_{t-1} but not strictly, then $c(t) \leq 2$.*

The tree height definition assures that

$$\Phi(t-1) \geq l_{i_t} \tag{12}$$

Since the weight of the 0-node sibling is equal to w_{i_t} and $\Phi(t)$ is equal to the level of the 0-node's sibling, then it follows by proposition 1 that

$$\Phi(t) \leq l'_{i_t} + 2 \tag{13}$$

Hence, by subtracting (12) from (13) we obtain that

$$\Phi(t) - \Phi(t-1) \leq l'_{i_t} + 2 - l_{i_t} = -h_t + 2$$

Rearranging this last inequality, we get that $c(t) \leq 2$. □

The next lemma shows that $c(t) \leq 2$ for case (iv).

Lemma 4 *If either T_{t-1} does not have a 0-node or w_{i_t} is greater than the weight of the 0-node sibling, then $c(t) \leq 2$.*

This result follows immediately from proposition 5, since $\Phi(t) = H(T_t)$, $\Phi(t-1) = H(T_{t-1})$ and $c(t) = \Phi(t) - \Phi(t-1) + h_t$. $\Box$

The following theorem establishes Vitter's conjecture.

Theorem 2 $D_t \leq S_t + 2t - 2k + 1 - \lceil \log Min(k+1, n) \rceil$.

The condition (i) of theorem 1 is satisfied only once, that is when $t = 1$. Let n_2, n_3 and n_4 be respectively the number of times where conditions (ii), (iii) and (iv) of theorem 1 are satisfied. Observe that for each $t > 1$, one and only one of these conditions must be satisfied. Theorem 1 assures that

$$\sum_{i=1}^{t}(d_i - s_i) \leq \sum_{i=1}^{t} h_i + n_3 - n_2 \tag{14}$$

On the other hand, we have

$$\sum_{i=1}^{t} h_i = \sum_{i=1}^{t} c(t) + \Phi(0) - \Phi(t)$$

Replacing the value of $c(t)$ by its upper bounds given by lemmas 1, 2, 3 and 4, we obtain

$$\sum_{i=1}^{t} h_i \leq 2t - 1 - n_2 - 2n_3 + \Phi(0) - \Phi(t) \tag{15}$$

Since T_t is full binary tree with $Min(k+1, n)$ leaves, it follows that $\Phi(t) \geq \lceil \log Min(n, k+1) \rceil$. We also have that $\Phi(0) = 0$ and $n_2 = k - 1$. Replacing these values in (15), we obtain that

$$\sum_{i=1}^{t} h_i \leq 2t - k - 2n_3 - \lceil \log Min(k+1, n) \rceil \tag{16}$$

Hence, it follows from (14) and (16) that

$$\sum_{i=1}^{t}(d_i - s_i) \leq 2t - 2k + 1 - \lceil \log Min(k+1, n) \rceil$$

Since $D_t - S_t = \sum_{i=1}^{t}(d_i - s_i)$, then the proof is concluded. $\Box$

The next corollary shows that $\epsilon < 2$ for the FGK algorithm.

Corollary 1 *For the FGK algorithm we have* $\epsilon < 2$

It follows from theorem 2 that

$$D_t \leq S_t + 2t - 2k + 1 - \lceil \log Min(k+1, n) \rceil$$

Hence, we have that

$$\epsilon \leq \frac{2t - 2k + 1 - \lceil \log Min(k+1, n) \rceil}{t}$$

and as a consequence $\epsilon < 2$. $\square$

4 The sharpness of the bound

The maximum of the potential function $c(t)$ can be reached only under the conditions of lemmas 3 and 4. In order to prove the sharpness of our bound, we present a simple example that explores the conditions stated in lemma 3. A similar example was used by Vitter [12] to prove the sharpness of the bound on the Λ algorithm. We assume that $\Sigma = \{a_1, \ldots, a_{2^j+2}\}$ and that $a_{i_m} = a_m$ for $m = 1, \ldots, 2^j + 1$, that is, the symbol a_m is the m'th transmitted symbol. The resulting Huffman tree for $j = 3$ is shown in figure 6. By counting the number of bits, we obtain that $D_{2^j+1} = j.2^j + 1$ and $S_{2^j+1} = j.2^j + j + 2$, what implies that $D_t = S_t + 2t - 2k - \lceil \log(k+1) \rceil$, for $t = 2^j + 1$.

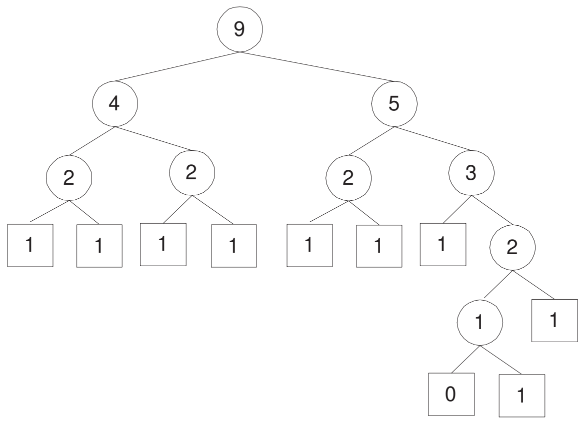

Figure 6: The picture shows a Huffman tree obtained by the FGK algorithm after the processing of 9 distinct symbols.

After that, the symbol corresponding to the 0-node sibling in the current Huffman tree is always the symbol to be transmitted. This last rule forces the level of the current node to decrease by two units in all situations, except when either its weight is the minimum strictly positive weight or when there is exactly one more symbol with equal weight. In the first case, its level remains the same and in the second case it decreases by one unity. Hence, for each set of k new transmitted symbols , the difference between D_t and S_t increases by $2k-3$. Therefore, for such a set of symbols, we obtain that $D_t = S_t + 2t - 2k - 3\lfloor (t-k)/k \rfloor - \lceil \log(k+1) \rceil$.

If we make t and k go to $+\infty$ in the last example, with t growing asymptotically faster than k, we obtain that $\epsilon \rightarrow 2$. Hence, $\epsilon < 2$ is an asymptotically tight bound.

5 Final Comments

In this paper, we have proved a sharp bound on the compression loss of the FGK algorithm. This bound improves the previous bound due to Vitter [12]. However, practical experiments suggest that the usual compression loss is much smaller than the value given by our bound. Although our bound is tight, we believe that an improved analysis is also possible for the cases where the symbol distribution is skew.

In [6], Larmore asks if it is possible to update an Adaptative length restricted Huffman Code with $O(l)$ time effort. For future research, we intend to address this question by using some new results concerning the length restricted prefix codes [8, 9].

References

[1] Faller, N. *An adaptive system for data compression.* In Record of the 7th Asilomar Conferences on Circuits, Systems and Computers (Pacific Grove, Calif., Nov.), Naval Postgraduate School, Monterrey, Calif, 593-597, 1973.

[2] Gallager, R. G., *Variations on a theme by Huffman*, IEEE Transactions on Information Theory, IT-24(1978), 668-674.

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

[4] Katajainen, J., Moffat, A., *Inplace calculation of minimum-redundancy codes*, In Proc. Workshop an Algorithms and Data Structures, Kingston University, Canada, August 1995. Springer-Verlag.

[5] Huffman, D.A. *A Method for the Construction of the Minimum Redundancy Codes*, Proc. Inst. Radio Engineers, Vol 40,No 10(1952), 1098-1101.

[6] Larmore, L.L. and Hirschberg, D.S. *A Fast Algorithm for Optimal Lenght-Limited Huffman Codes*, Journal of the ACM, July (1990), 464-473.

[7] Leeuwen, J. van, On the construction of Huffman trees. In proc. 3rd Int. Coll. On Automata, Languages, and Programming(1976), 382-410.

[8] Milidiú, R.L. and Laber, E.S., *The WARM-UP ALGORITHM: A Lagrangean Construction of Length Restricted Huffman Codes*, Monografias em Ciência da Computação 15/96, Departamento de Informática, PUC-RJ, 1996.

[9] Milidiú, R.L., and Laber, E.S., *Bounding the Inefficiency Introduced by Length-Restricted Prefix Codes* Monografias em Ciência da Computação 33/97, Departamento de Informática, PUC-RJ, 1997.

[10] Moffat A. and Turpin A., *Efficient Construction of Minimum-Redundancy Codes For Large Alphabets*, To appear in IEEE Transactions on Information Theory.

[11] Tarjan, R. E. "Amortized Computational Complexity", SIAM Journal on Applied and Discrete Methods, vol. 6, no 2(1985), 306-318.

[12] Vitter, J. S., *Design and Analysis of Dynamic Huffman codes*, Journal of ACM 34, 4(1987).

[13] Vitter, J. S., *Algorithm 673: Dynamic Huffman coding*, ACM Transactions on Mathematical Software, vol 15, no 2(1989), 158-167.

[14] Zobel, J. and Moffat, A. *Adding Compression to full-text retriweval system*, Software Practice and Experience, 25(8) (1995), 891-903.