An Approximation Algorithm for Constructing Error Detecting Prefix Codes

Artur Alves Pessoa
Universidade Federal Fluminense

Abstract

A -bit Hamming prefix code is a binary code with the following property: for any codeword and any prefix y of another codeword, both and y having the same length, the Hamming distance between and y is at least . Given an alphabet $A = [\ _1, \ldots, \ _n]$ with corresponding probabilities $[p_1, \ldots, p_n]$, the -bit Hamming prefix code problem is to find a -bit Hamming prefix code for A with minimum average codeword length $\sum_{i=1}^{n} p_i$, where $_i$ is the length of the codeword assigned to $_i$. In this paper, we propose an approximation algorithm for the 2-bit Hamming prefix code problem. Our algorithm spends $O(n \log^3 n)$ time to calculate a 2-bit Hamming prefix code with an additive error of at most $O(\log \log \log n)$ bits with respect to the entropy $= -\sum_{i=1}^{n} p_i \log_2 p_i$.

1 Introduction

A -bit Hamming prefix code is a binary code with the following property: for any codeword and any prefix y of another codeword, both and y having the same length, the Hamming distance between and y is at least . Given an alphabet $A = [\ _1, \ldots, \ _n]$ with corresponding probabilities $[p_1, \ldots, p_n]$, the -bit Hamming prefix code problem is to find a -bit Hamming prefix code for A with minimum average codeword length $\sum_{i=1}^{n} p_i$, where $_i$ is the length of the codeword assigned to $_i$. It is worth to mention that the well-known prefix code problem is a special case of the previous problem, where $= 1$. For the sake of simplicity, let us refer to 2-bit Hamming prefix codes only as Hamming prefix codes.

It is well known that the optimum solution for the prefix code problem can be obtained by the Huffman's algorithm [4] in $O(n \log n)$ time. A prefix code constructed by the Huffman's algorithm is usually referred to as a Huffman code. On the other hand, Hamming devised algorithms for the construction of fixed-length error detecting codes [3]. The Hamming prefix code problem has been proposed by Hamming in the same

1ª Versão Recebida em 10/0 /06 - Publicado em 22/11/06

book as a way to combine both the compression provided by Huffman codes and the error protection provided by Hamming codes. However, we found only one paper in the literature addressing the problem [10], where the authors propose a method for the 3-bit Hamming prefix code problem (refereed to as ECCC problem). The proposed method achieves a relatively small loss of compression with respect to the Huffman code in some practical experiments. However, the authors report that they have no good criterion to determine the codeword lengths used in the method. Moreover, no worst-case bound on the compression loss is provided. In [8], Pinto et. al. give a polynomial algorithm for finding an optimal prefix code where all codewords have even parities. The authors are motivated by the Hamming prefix code problem but the resulting code is much less effective for detecting errors. In a Hamming prefix code, one can detect one bit changed before finishing the decode of the first corrupted codeword. Codes with even codeword parities do not necessarily have this property. As far as we know, no polynomial (exact or approximate) algorithm and no hardness proof has been found for the Hamming prefix code problem.

Similarly to conventional prefix codes, any Hamming prefix code can be represented by a full binary tree. Such a tree has two types of leaves: the codeword leaves, which represent the codewords, and the error leaves, which represent binary prefixes that are forbidden and interpreted as errors. In this representation, each codeword length ℓ_i is given by the level of the corresponding codeword leaf in the tree. As a result, the average codeword length of a code is equal to the weighted external path length of the corresponding tree, where each leaf is weighted with the probability of the corresponding symbol (error leaves have zero weights). Throughout this paper, we refer to these trees as *Hamming trees*. Moreover, full binary trees that represent conventional prefix codes are referred to as *code trees*. Finally, we use the term *code-and-error tree* to denote any full binary tree with both codeword leaves and error leaves. Observe that a Hamming tree is a special case of a code-and-error tree.

In this paper, we propose an approximation algorithm for the Hamming prefix code problem. Our algorithm runs in $O(n \log^3 n)$ time. It obtains a Hamming prefix code whose average codeword length is at most $O(\log \log \log n)$ bits larger than the entropy. We observe that one must increase the codeword length of a fixed-length code by at least 1 bit to achieve a Hamming distance of 2 bits. For a Hamming distance of 3 bits, this increase must be at least $\lceil \log_2 \log_2 n \rceil$ bits. For variable-length codes, our result gives an upper bound on the average codeword length increase that is between

$O(1)$ and $O(\log \log n)$. Moreover, it is worth to mention that our algorithm is suitable for compression schemes where the alphabet can be arbitrarily large and the average codeword length is not too small such as the word-based text compression [7]. For example, if the average codeword length grows as any $\omega(\log \log \log n)$ function of n, then our approximation bound implies in an approximation ratio of $1 + o(1)$.

This paper is organized as follows. In section 2, we introduce a new technique to rearrange a code-and-error tree so that it becomes a Hamming tree. In section 3, we describe and analyze our approximation algorithm. In section 4, we summarize our conclusions. Throughout this paper, let us use $\log x$ to denote $\log_2 x$.

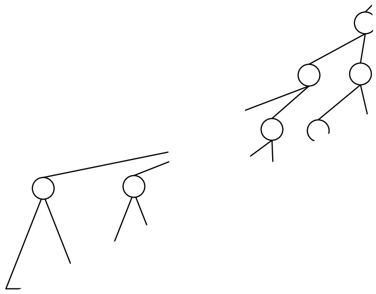
2 Spreading leaves

In this section, we introduce a general procedure for transforming a code-and-error tree into a Hamming tree. Basically, this procedure iterates for each level in the input code-and-error tree rearranging the codeword leaves, error leaves and internal nodes in such a way that the Hamming distance between a codeword leaf and an internal node or another codeword leaf is at least two. Let us refer to this procedure as *Spread*.

Figure 1 illustrates the Spread procedure. In Figure 1.(a), we represent a code-and-error tree, where codeword leaves are rectangles, error leaves are gray filled circles, and internal nodes are white filled circles. Moreover, the subtrees rooted at level four are represented by triangles.

Figure 1.(b) shows the tree of figure 1.(b) after the first iteration of the Spread procedure. In this iteration, the proc
codeword leaf x found at this level. He
level. Then, the procedure moves the th
to codewords whose Hamming distanc
codewords are 100, 010 and 001. As r

(a)



positions	codewords	signatures	nodes
1	0,11,0	000	internal
2	0,11,1	001	internal
3	1,01,0	110	error
4	1,01,1	111	codeword
5	1,10,0	110	error
6	1,10,1	111	codeword
7	1,11,0	100	internal
8	1,11,1	101	error

Table 1: The available codewords at level four in the tree of Figure 1.(c), the corresponding codeword sections and signatures.

After calculating the signatures, the Spread procedure chooses one signature for the codeword leaves. As a result, all codeword leaves are moved to the positions that have the chosen signature. In the example of table 1, the first and the last columns of this table shows a sequence number for the codeword positions and the type of node moved to this position, respectively. Observe that the chosen signature is 111 in this case.

Finally, the Spread procedure computes the Hamming distance between the chosen signature and the other signatures. The error leaves are moved to the positions that correspond to signatures whose Hamming distance to is one. The internal nodes are moved to the remaining positions. In the example of table 1, since = 111, the signatures 011, 101 and 110 shall be reserved to error leaves. These signatures correspond to the positions 3, 5 and 8.

Now, one can observe that the first iteration of the Spread procedure corresponds to dividing the available codewords into sections of one bit each. The criterion used to divide codewords is a subject of Section 3.

Next, we formalize the Spread procedure. Let be a code-and-error tree. At the ith iteration of the procedure, we denote by ℓ_i the level of being processed, for $i = 1, \ldots, L$. At this level, the available codewords are denoted by $c_1^i, \ldots, c_{\alpha_i}^i$. Each codeword c_j^i can be written as the concatenation of q_i sections of bits. For that, we use the notation $c_j^i = s_1^{i,j} + \cdots + s_{q_i}^{i,j}$, where $|s_k^{i,1}| = \cdots = |s_k^{i,\alpha_i}| = \ell_i^k$, for all $i = 1, \ldots, r$ and $= 1, \ldots, q_i$. Observe that $\ell_i = \sum_{k=1}^{q_i} \ell_i^k$. Finally, we use $\pi(c)$ and $d(c, y)$ to denote the parity of the bit string c and the Hamming distance between the two bit strings c

and y, respectively. At the ith iteration, the Spread procedure performs the following five steps:

Step 1: Choose the section lengths $\ell_i^1, \ldots, \ell_i^{q_i}$ for level ℓ_i;

Step 2: Choose a signature $^{i} = {}^{i}_{1} + \cdots + {}^{i}_{q_i}$ for the codeword leaves, where ${}^{i}_{k}$ is a single bit for $= 1, \ldots, q_i$.

Step 3: Move each codeword leaf to a distinct position that corresponds to a codeword c_j^i such that $\pi(s_k^{i,j}) = {}^{i}_{k}$ for $= 1, \ldots, q_i$.

Step 4: Move each error leaf to a distinct position that corresponds to a codeword c_j^i such that $\pi(s_k^{i,j}) = {}^{-}_{k}$ for $= 1, \ldots, q_i$, where ${}^{-} = {}^{-}_{1} + \cdots + {}^{-}_{q_i}$ is a signature such that $d({}^{i}, {}^{-}) = 1$.

Step 5: Move the internal nodes the positions that correspond to the remaining signatures.

For a given iteration i of the previous procedure, let us refer to the positions mentioned in Step 3 (resp. Step 4) as the codeword (resp. error) positions. The correctness of the Spread procedure is a consequence of the following proposition.

Proposition 1 *If the Hamming distance between two codewords of c_j^i and $c_{j'}^i$ is one, then the Hamming distance between the corresponding two signatures is also one.*

Proof: Since c_j^i and $c_{j'}^i$ differ by exactly one bit, exactly one section of c_j^i and one section of $c_{j'}^i$ have different parities. ■

Observe that, by the previous proposition, the Spread procedure places an error leaf in all positions whose Hamming distances from a codeword leaf is one.

2.1 Choosing the section lengths

A natural question that arises when looking at the Spread procedure is how to choose the sections lengths in each iteration so that codeword leaves, error leaves and internal nodes can be placed accordingly. Here, a remarkable observation is that the previous procedure can be more flexible than the previous description suggests. In fact, one can place some internal nodes at codeword positions if there are more such positions than codeword leaves. Moreover, error leaves can be placed anywhere provided that all error positions are already filled. As a result, a necessary and sufficient condition for an

1ª Versão Recebida em 10/0 /06 - Publicado em 22/11/06

iteration of the Spread procedure to succeed is having both enough error leaves to fill all error positions and enough codeword positions to accommodate all codeword leaves.

The following theorem gives an upper bound on the number of error leaves required to successfully apply the Spread procedure on a code-and-error tree. This upper bound will be useful when we analyze the additive error of our algorithm in the next section. Moreover, the theorem proof describes a method that may be used in the Step 1 of the Spread procedure to choose the section lengths.

Theorem 1 *At the ith iteration of the Spread procedure, let a_i and b_i be the number of codeword leaves and the number of error leaves at level ℓ_i of the input tree , respectively. If $b_i \geq (\ell_i - \lceil \log a_i \rceil)(2a_i - 1)$ then there is a valid choice of section lengths for this iteration.*

Proof: The valid choice of section lengths can be obtained through the following method. Start with only one section whose length is ℓ_i. On each iteration, choose for the codeword leaves the signature that gives more available positions. If the number of available positions is larger than $2a_i - 1$, than choose an arbitrary codeword section larger than 1 and split it into two sections of arbitrary lengths. Then, go to the next iteration. Otherwise, if the number of available positions is not larger than $2a_i - 1$ then stop. In this case, the current section length and the currently chosen signature are used.

Next, we show that we have both at least a_i codeword positions and at most $(\ell_i - \lceil \log a_i \rceil)(2a_i - 1)$ error positions, which is enough to prove this theorem.

First, let us consider the set of available positions that corresponds to each signature on each iteration of the previous method. Let $_j$ and * be the maximum set cardinality among all such sets before the jth iteration and after the last iteration, respectively. Observe that each such set is split into two subsets after each iteration. As a result, we have $_{j+1} \geq \ _j/2$ for each iteration j except the last one. Since the previous method only performs a split when $_j \geq 2a_i$, we obtain that $^* \geq a_i$.

Now, observe that $^* \leq 2^{\ell_i - q_i}$, where q_i is the number of chosen codeword sections. This is true because each available codeword has ℓ_i bits from which q_i bits are used to match the section parities. As a result, we must have $q_i \leq \ell_i - \lceil \log a_i \rceil$. Since we have exactly q_i sets of error positions, each one with cardinality not greater that $_* \leq 2a_i - 1$, we have at most (ℓ

3 The Approximation Algorithm

In this section, we describe our approximation algorithm for the Hamming prefix code problem. Basically, our approach is to raise the leaves of an usual prefix code tree so that sufficient error leaves can be inserted. Thus, let us refer to this algorithm as the Raising algorithm.

3.1 Description

The Raising algorithm performs the following three steps:

Step 1: Construct a full binary tree $_1$ with n leaves whose height is at most $L = 2\lceil \log n \rceil$.

Step 2: Obtain an expanded tree $_2$ from $_1$ with $n_2 \succ n$ leaves and set $n_2 - n$ leaves as error leaves.

Step 3: Apply the Spread procedure on $_2$ to obtain a Hamming tree $_3$.

In Step 1, the Raising algorithm uses a modified version of the BRCI algorithm [6] to construct an L-restricted binary tree $_1$ whose weighted external path length exceeds the entropy by at most $1 + 1/n$. In Step 2, for each level in $_1$, the Raising algorithm raises the leaves from level to a higher level in such a way there is enough room to place the necessary error leaves without affecting the other leaves of $_1$. Let $_2$ be the resulting tree. Also, let us refer to this step as the *Raising Step*. In the next subsection, we give more details on the Raising Step. Finally, in Step 3, we apply the Spread procedure on $_2$ to obtain a Hamming tree $_3$.

3.2 Constructing a Length-restricted Prefix Code

The construction of length restricted prefix codes has been addressed by several researchers [5, 9, 6]. However, none of the algorithms found in the literature serves for our purposes. In order to prove our approximation bound, we need to construct a height-restricted full binary tree where the probability of any node at a given level is bounded above by $/2^{\ell}$ for some constant . We devise such a construction by slightly modifying the BRCI algorithm proposed in [6]. The complete description of this method has been moved to Appendix A. However, the following two propositions are useful for our purposes.

1ª Versão Recebida em 10/0 /06 - Publicado em 22/11/06

Proposition 2 *The probability of a node at level of is bounded above by* $1/2^{\ell-2}$.

Proof: See Appendix A. ■

Proposition 3 *The weighted external path length of is at most* $+1+1/2^{L-\lceil \log n \rceil - 2}$.

Proof: See Appendix A. ■

3.3 The Raising Step

Here, we describe the method used by the Raising Step to move all leaves from a given level of and insert the corresponding error leaves. Basically, this step consists in replacing each leaf by complete binary tree with height . Let be the number of leaves at level in . After the previous transformation, leaves are available at level + . Then, the method sets of these leaves as codeword leaves and the remaining $(2^{\bar{\ell}}-1)$ leaves as error leaves. The following theorem gives a suitable value for as a function of both and that satisfies the condition of Theorem 1.

Theorem 2 *If* $\geq \lceil \log \rceil$ *and* $= \lceil \log(- \lceil \log \rceil + 2) \rceil + 2$, *then*

$$(2^{\bar{\ell}} - 1) \geq (+ - \lceil \log \rceil)(2 - 1). \quad (1)$$

Proof: We divide it into two cases:

Case 1: $- \lceil \log \rceil < 4$,

Case 2: $- \lceil \log \rceil \geq 4$,

We analyze Case 1 for all possible values of $- \lceil \log \rceil$. The table bellow summarizes this analysis, where the third and the fourth columns show the corresponding expressions for the left-hand side of (1) and the right-hand side of (1), respectively.

$- \lceil \log \rceil$		LHS of (1)	RHS of (1)
0	3	7	$6 - 3$
1	4	15	$10 - 5$
2	4	15	$12 - 6$
3	5	31	$16 - 8$

Observe in the previous table that (1) is satisfied for all rows.

For Case 2, we have that the left-hand side of (1) is bounded bellow by

$$2^{k_1}(\ell - \lceil \log k_1 \rceil) + 2^{k_2}(\ell - \lceil \log k_2 \rceil + 1.5) \tag{2}$$

and the right-hand side of (1) is bounded above by

$$2^{k_1}(\ell - \lceil \log k_1 \rceil) + 2^{k_2}\bar{\ell}. \tag{3}$$

Observe that (2) is not smaller than (3) whenever $\bar{\ell} \leq \ell - \lceil \log k_2 \rceil + 1.5$, which is true whenever $\ell - \lceil \log k_2 \rceil \geq 4$. From this it follows that (1) is satisfied for Case 2. ■

Observe that, depending on the value of k_2, we may have a leaf w_1 at a higher level than another left w_2 in T_1, and the same leaf w_1 at a lower level than w_2 in T_2. In this case, the Spread procedure applied in the Step 3 of the Raising algorithm will process w_1 before w_2. Moreover, exchanging w_1 and w_2 in T_2 might improve the external weighted path length of the resulting tree in this case. However, we do not consider this improvement in our analysis. Finally, we have the case where w_1 and w_2 are at different levels in T_1 but at the same level in T_2. For this case, we observe that, if the condition of Theorem 1 is true independently for two sets of leaves at the same level, then it is also true for the union of these two sets. Hence, it is safe to process both w_1 and w_2 at the same iteration of the Spread procedure.

3.4 Analysis

In this section, we analyze both the time complexity of the Raising algorithm and the compression loss of the Hamming prefix code generated by it.

The following proposition gives an upper bound on the time complexity of the Raising algorithm.

Proposition 4 *The Raising algorithm runs in $O(n \log^3 n)$ time.*

Proof: The Step 1 runs in $O(n)$ time. As a result of this step, we have a sorted list of leaf levels. Then, in Step 2, each sublist of k_2 equal leaf levels in the input generates $k_2 2^{\bar{\ell}}$ levels for both error and codeword leaves in the output, where $\bar{\ell} = O(\log \ell) = O(\log \log n)$ according to Theorem 2. Hence, the total number of leaves in T_2 is $O(n \log n)$. Since T_2 is a full binary tree, it also has $O(n \log n)$ internal nodes. As a result, T_2 can be constructed in $O(n \log n)$ time. Finally, in Step 3, we apply

the Spread procedure on 2. In each iteration of this procedure, we must choose the sections lengths for the nodes at the current level of 2 using the method described in the proof of Theorem 1. In each iteration of this method, we traverse the whole tree until level counting the number of available codewords that correspond to each signature for the current section lengths. This counting operation takes $O(n \log n)$ time in the worst case. Since the method given by Theorem 1 performs $O(\log n)$ iterations, it runs in $O(n \log^2 n)$ time. Moreover, the Spread procedure spends $O(n \log^3 n)$ time executing the previous method $O(\log n)$ times (once in each iteration). The remaining operations performed by the Spread procedure are dominated by the choice of section lengths since they spend only $O(n \log n)$ time. As a result, the overall time complexity of the Raising algorithm is $O(n \log^3 n)$. ■

Now, let us analyze the redundancy of the Hamming prefix code generated by the Raising algorithm.

Theorem 3 *The average codeword length of a Hamming prefix code generated by the Raising algorithm is at most*

$$\log(\log\lceil \log n \rceil + 5) + 4 + 4/n \tag{4}$$

bits larger than the entropy .

Proof: Let $\bar{p}_\ell$ be the sum of the probabilities of all leaves at level of 1. By Theorem 2, the compression loss of the code generated by the Raising algorithm with respect to the prefix code constructed by the modified BRCI algorithm is given by

$$\sum_{\ell=1}^{L} \bar{p}_\ell \left(\lceil \log(- \lceil \log \rceil + 2) \rceil + 2 \right).$$

Moreover, by Proposition 3, the compression loss of the modified BRCI with respect to is at most $1 + 4/n$, for $L = 2\lceil \log n \rceil$. This gives the following upper bound on the overall compression loss with respect to .

$$\leq \sum_{\ell=1}^{L} \bar{p}_\ell \left(\lceil \log(- \lceil \log \rceil + 2) \rceil \right) + 3 + 4/n.$$

On the other hand, by Proposition 2, we have $\bar{p}_\ell \leq / 2^{\ell-2}$. As a result, we obtain that $-\log \bar{p}_\ell \geq - \lceil \log \rceil - 2$. By combining this inequality with the previous upper bound on , we obtain that

$$\dot{\boldsymbol{\epsilon}} \leq \sum_{\ell=1}^{L} \bar{p}_\ell \log(-\log \bar{p}_\ell + 4) + 4 + 4/n. \tag{5}$$

Now, let us consider the function $f(\bar{p}_\ell) = \bar{p}_\ell \log(-\log \bar{p}_\ell + 4)$. From elementary calculus, we can conclude that this function has a decreasing derivative (for $\bar{p}_\ell \succ 0$). As a result, we have that $2f((\text{ } + y)/2) \succ f(\text{ }) + f(y)$. Hence, we maximize $\sum_{\ell=1}^{L} f(\bar{p}_\ell)$ subject to $\sum_{\ell=1}^{L} \bar{p}_\ell = 1$, when $\bar{p}_1 = \cdots = \bar{p}_L = 1/L$. By replacing each $\bar{p}_\ell$ by $1/(2\lceil \log n \rceil)$ in the right-hand side of (5), we obtain (4). ■

4 Conclusions

In this section, we give some conclusions on the results of this paper.

Table 2 shows the additional bits required to achieve each Hamming distance for each type of code. This table gives an intuition of how our result compares to other classical error detection codes.

Codeword length	**Hamming distance**	**Added bits**
Fixed	2	1
Variable	2	$O(\log \log \log n)$
Fixed	3	$\lceil \log \log n \rceil$

Table 2: Additional bits required to achieve each Hamming distance.

In addition, we remark that our theoretical upper bound on the additive error of the Raising algorithm is less than eight bits for any practical purpose. For example, we have an upper bound of 6.9993 bits for $n = 10^6$ and 7.0706 bits for $n = 10^9$. Moreover, an additive error equal to this upper bound can be acceptable in some applications. For example, for the word-based text compression scheme [7] mentioned in the introduction, the Zipf's law [2] is accepted as an estimation of the symbol probability distribution. This estimation leads to ≈ 18.9515 for $n = 10^9$, in which case the Raising algorithm achieves an approximation ratio of 1.3731.

Next, we point out that the $O(n \log^3 n)$ time complexity of the Raising algorithm is suitable to deal with large alphabets. For example, a $\Theta(n^2)$ algorithm would be prohibitive for $n = 10^9$.

Finally, we observe that this work may have interesting extensions. A similar approach might be applicable to devise approximation algorithms for the -bit Hamming

prefix code problem, for $\succ 2$. Moreover, an implementation of the Raising algorithm with some practical improvements might generate codes with additive errors much smaller than the worst-case upper bound.

References

[1] J. Brian Connell. A Huffman-Shannon-Fano code. In *Proceedings of the IEEE*, volume 61, pages 1046–1047, July 1973.

[2] A. S. Fraenkel and S. T. Klein. Bounding the depth of search trees. *The Computer Journal*, 36(7):668–678, 1993.

[3] R. W. Hamming. *Coding and Information Theory*. Prentice Hall, 1980.

[4] 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.

[5] Lawrence L. Larmore and Daniel S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473, July 1990.

[6] Ruy L. Milidiú and Eduardo S. Laber. Bounding the inefficiency of length-restricted prefix codes. *Algorithmica*, 31(4):513–529, 2001.

[7] Alistair Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, February 1989.

[8] Paulo E. D. Pinto, Fábio Protti, and Jayme L. Szwarcfiter. A huffman-based error detecting code. In *WEA'2004 - III International Workshop on Efficient and Experimental Algorithms*, pages 446–457, Angra dos Reis, Brazil, may 2004.

[9] Baruch Schieber. Computing a minimum-weight -link path in graphs with the concave Monge property. *Journal of Algorithms*, 29(2):204–222, November 1998.

[10] Thomas Wenisch, Peter F. Swaszek, and Augustus K. Uht. Combined error correcting and compressing codes. In *IEEE International Symposium on Information Theory*, page 238, Washington, DC, USA, june 2001.

A The Modified BRCI Algorithm

Before describing the modified BRCI algorithm, we shall discuss some details of the original one. This algorithm constructs an L-restricted binary tree by rearranging some selected nodes in a Huffman tree . First, it removes from all leaves at levels not smaller than L. Then, it raises the subtree rooted at the node with the smallest probability at level $= L - \lceil \log n \rceil - 1$ to level $+ 1$. Finally, it inserts in a complete binary tree $'$ containing all removed leaves so that the root of $'$ becomes sibling of . Since the height of $'$ is bounded above by $\lceil \log n \rceil$, the level of each inserted leaf in the resulting tree $^{-}$ is at most $+ 1 + \lceil \log n \rceil = L$.

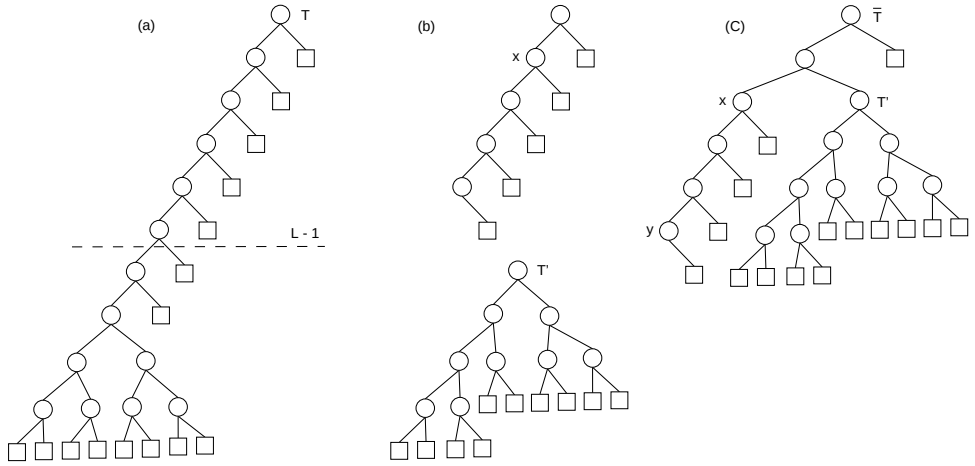


Figure 2: The BRCI algorithm.

Figure 2 illustrates a BRCI construction for $L = 6$ and $n = 15$. Figure 2.(a) shows an example of a Huffman tree . Figure 2.(b) represents both the tree after the removal of the leaves at levels higher than $L - 1$ and the complete binary tree $'$ built with the removed leaves. Observe in this figure that the node is chosen at level one and that the height of $'$ is exactly $\lceil \log n \rceil = 4$. Finally, Figure 2.(c) shows the resulting tree $^{-}$ obtained after raising the node and inserting $'$. Observe in this figure the $^{-}$ is not a full binary tree because the node y has only one child. In this case, we shall remove the node y and connected its child as a child of its parent. Let us assume that the BRCI algorithm always transforms the resulting binary tree into a full binary tree by removing internal nodes with single children as described before.

In order to describe our construction, we recall some well-known results of the information theory field. The prefix code problem can be written in terms of the Kraft-

McMillen inequality $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$ as follows. Find integer codeword lengths $\ell_1, \ldots, \ell_n$ that minimize $\sum_{i=1}^{n} p_i \ell_i$ subject to the Kraft-McMillen inequality. Given the optimal solution for the previous problem, a corresponding prefix code (or full binary tree) can be constructed in a linear time [1]. Observe that the Kraft-McMillen inequality is satisfied with equality in this case. Moreover, if we relax the integrality constraint in the previous problem, then the optimal solution is given by $\ell_i = -\log p_i$, for $i = 1, \ldots, n$. In this case, observe that the average codeword length is the entropy $= -\sum_{i=1}^{n} p_i \log p_i$. Hence, an immediate way to find integer codeword lengths whose average is at most one bit larger than is setting $\ell_i = \lceil -\log p_i \rceil$. Our construction uses the previous idea to replace the Huffman tree used by the BRCI algorithm. Instead, we construct a full binary tree based on the previous codeword lengths. In order to satisfy the Kraft-McMillen inequality with equality, we use the following procedure. While there is a value of ℓ_j such that $\Delta = 1 - \sum_{i=1}^{n} 2^{-\ell_i} \geq 2^{-\ell_j}$, decrease ℓ_j by $\lfloor \log(\Delta / 2^{-\ell_j} + 1) \rfloor$. After that, we use the method proposed in [1] to construct a full binary tree whose leaves are at the levels $\ell_1, \ldots, \ell_n$. Let $\hat{}$ be this tree.

Now, let us assume that the tree $\bar{}$ has been constructed by the BRCI algorithm by rearranging the nodes of $\hat{}$ instead of . Observe that the overall construction runs in $O(n)$ time. Next, we give the proofs for both Propositions 2 and 3.

Proof of Proposition 2: By construction, the level ℓ of a leaf of $\hat{}$ with probability p is not greater than $-\log p + 1$. Hence, we have $\ell \leq 1/2^{\ell-1}$. Now, we claim that the previous bound is valid for both leaves and internal nodes of $\hat{}$. Next, we prove this claim by induction on n. For $n = 1$, the claim is true because $\hat{}$ has no internal node. Now, let us assume that the claim is true for $n < n'$. For $n = n'$, choose an internal node y at a maximum level ℓ among all internal nodes of $\hat{}$. Since the two children of y are necessarily leaves, their probabilities are bounded above by $1/2^{\ell}$ each. As a result, the probability of y is bounded above by $1/2^{\ell-1}$. Now, remove the two children of y from $\hat{}$. In this case, y becomes a leaf with the same probability as before, and we have $n = n' - 1$. Moreover, the probabilities of all other nodes remain unchanged. Hence, by inductive hypothesis, the probability bound is also valid for all other internal nodes of $\hat{}$.

Since the BRCI algorithm increases the level of each node in $\hat{}$ by at most one, we obtain that the probability of a node at level ℓ in $\bar{}$ is bounded above by $1/2^{\ell-2}$ ■

Proof of Proposition 3: By the previous discussion, the weighted external path

length of $\hat{}$ is bounded above by $+1$. The BRCI algorithm increases by one the level some the leaves in the subtree of $\hat{}$ rooted at $\checkmark$, which is at level $\ell = L - \lceil \log n \rceil - 1$. All other leaves have their levels either maintained or reduced. Hence, the increase in the weighted external path length of $\hat{}$ due to the BRCI algorithm is at most the probability of $\checkmark$. By the proof of proposition 2, this probability is bounded above by $1/2^{\ell-1} = 1/2^{L-\lceil \log n \rceil - 2}$. ■

1ª Versão Recebida em 10/0 /06 - Publicado em 22/11/06