An Approximation Algorithm for Constructing Error Detecting Prefix Codes

Artur Alves Pessoa
artur@producao.uff.br
Production Engineering Department
Universidade Federal Fluminense, Brazil

September 2, 2006

Abstract

A k-bit Hamming prefix code is a binary code with the following property: for any codeword x and any prefix y of another codeword, both x and y having the same length, the Hamming distance between x and y is at least k. Given an alphabet $A = [a_1, \ldots, a_n]$ with corresponding probabilities $[p_1, \ldots, p_n]$, the k-bit Hamming prefix code problem is to find a k-bit Hamming prefix code for A with minimum average codeword length $\sum_{i=1}^{n} p_i \ell_i$, where ℓ_i is the length of the codeword assigned to a_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 $H = -\sum_{i=1}^{n} p_i \log_2 p_i$.

1 Introduction

An encoding scheme designed for efficient data communication through an unreliable channel shall provide at least two basic features: compression and error detection/correction. While the provided compression shall be as efficient as possible (for the given computational resource constraints), error correction requirements depend on the reliability of the communication channel. One possible error correction scheme consists of adding error detection capabilities to the code and then retransmitting every data package where an error is found. In this case, the earlier detection of an error reduces the amount of data that must be retransmitted.

Typically, compression and error detection capabilities are added to an encoded message in two independent steps [3]. On the other hand, the combinatorial optimization problem that arises when designing a code with maximum compression efficiency subject to some error detection/correction requirement has been suggested by Hamming many years ago [3]. Nevertheless, we found only a few papers on this topic in the literature [10, 8].

In this context, we define a k-bit Hamming prefix code as a binary code with the following property: for any codeword x and any prefix y of another codeword, both x and y having the same length, the Hamming distance between x and y is at least k. Given an alphabet $A = [a_1, \ldots, a_n]$ with corresponding probabilities $[p_1, \ldots, p_n]$, the k-bit Hamming prefix code problem is to find a k-bit Hamming prefix code for A with minimum average codeword length $\sum_{i=1}^{n} p_i \ell_i$, where ℓ_i is the length of the codeword assigned to a_i. It is worth to mention that the well-known prefix code problem is a special case of the previous problem, where $k = 1$. 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 is based on an example given by Hamming in the same book that combines the compression provided by Huffman codes and the error protection provided by Hamming codes. However, we found only one paper in the literature addressing this specific 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 (say *Even prefix codes*). Although the authors have the same motivation, the resulting code only ensure the detection of (an even number of) errors at the end of the encoded message. On the other hand, Hamming prefix codes allow one to detect one bit changed before finishing the decode of the first corrupted codeword. As mentioned before, the earlier detection of errors may be desirable in some applications.

Next, we give an example to illustrate the difference between the Even prefix codes and Hamming prefix codes. Table 1 gives both an Even prefix code and a Hamming prefix code for the letters of the word "BANANA". When using the Even prefix code, "BANANA" is encoded as 101011011. In this case, if the first bit is changed to zero, then the resulting message (excluding the last bit) is erroneously interpreted as "AABB". In this (worst) case, the error is detected only when the last bit is decoded. On the other hand, "BANANA" is encoded as 1111001100001100 using the (suboptimal) Hamming prefix code of Table 1. In this case, if the first bit is changed to zero then the minimum Hamming distance between codewords and prefixes ensure that the prefix 01 of the resulting message is immediately interpreted as an error.

Symbol	**Even prefix code**	**Hamming prefix code**
A	0	00
N	11	1100
B	101	1111

Table 1: An Even prefix code and a Hamming prefix code for the letters of the word "BANANA".

As far as we know, no polynomial (exact or approximate) algorithm and no hardness proof has been found for the Hamming prefix code problem.

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 $1 + o(1)$.

A strictly binary tree is a tree where every internal node has exactly two children. Similarly to

conventional prefix codes, any Hamming prefix code can be represented by a strictly 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, strictly 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 strictly binary tree with both codeword leaves and error leaves. Observe that a Hamming tree is a special case of a code-and-error tree.

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 on 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 procedure processes level 3. It chooses the codeword 000 for the only codeword leaf x found at this level. Hence, it moves x to the leftmost position at this level. Then, the procedure moves the 3 error leaves to the positions that correspond to codewords whose Hamming distances with respect to x are one. These forbidden codewords are 100, 010 and 001. The 3 internal nodes of level 3 are arranged at the remaining positions.

For the sake of simplicity, let us refer to the codeword that corresponds to a given position p as the *label* of p. Moreover, we say that two positions in a given level ℓ are *neighbors* when the Hamming distance between the two corresponding labels is one. At level 3 of our example, the two positions with labels 000 and 100 are neighbors.

In the second iteration, the Spread procedure performs a similar step for level 4. However, this level introduces the following difficulties: not all codewords are available, and more than one codeword leaf shall be arranged at the same level. More specifically, 2 codeword leaves, 3 error leaves and 3 internal nodes shall be arranged into the 8 available positions with labels 0110, 0111, 1010, 1011, 1100, 1101, 1110, and 1111. Such an arrangement must satisfy the condition that every position having at least one neighbor occupied by a codeword leaf is occupied by an error leaf. In our example, the 2 codeword leaves are placed in the two positions with labels 1011 and 1101. Hence, the available neighbors of the position with label 1011 (the positions with labels 1010 and 1111) and the available neighbors of the position with label 1101 (the positions with labels 1100 and 1111) must be occupied by the 3 error leaves. The 4 internal nodes are placed in the remaining positions. Observe that the position with label 1111 is a common neighbor of the positions with

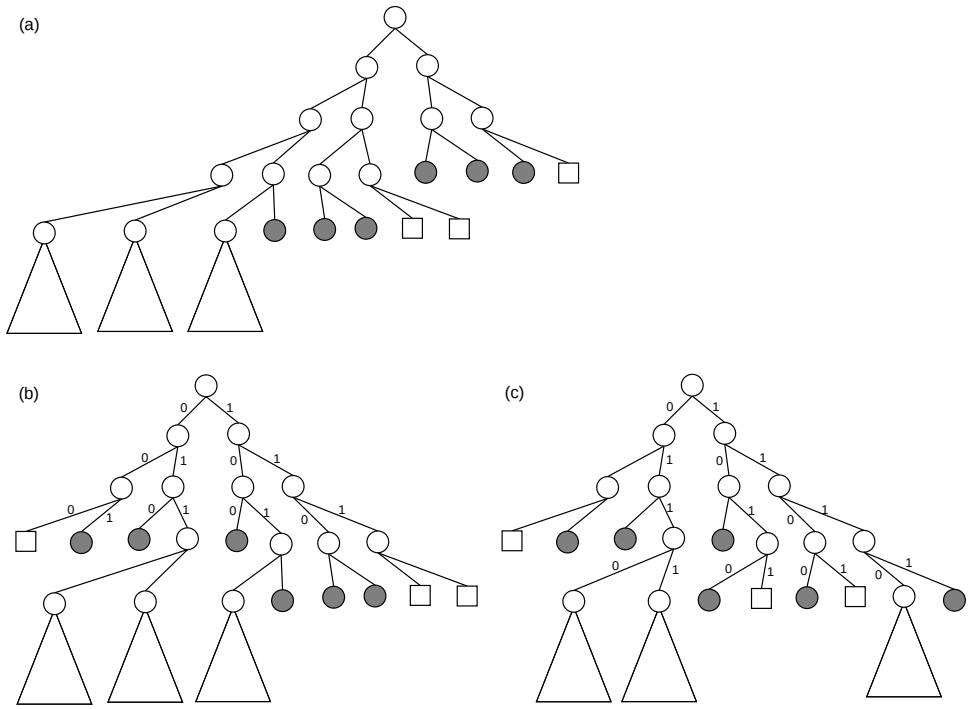

Figure 1: (a) A code-and-error tree; (b) the same tree after rearranging the nodes of level three; (c) the corresponding Hamming tree.

labels 1011 and 1101. In fact, for an efficient node placement, we shall try to generate as many common neighbors for the positions occupied by codeword leaves as possible in order to reduce the total number of error leaves. The Spread procedure has been developed with this intent.

Our approach to efficiently assign nodes to positions as described before is to split the codeword bits into *sections* and group the available codewords according to the parities of these sections. We remark that the sections for all codewords must follow that same pattern of lengths, that is, the length of the ith section is the same for all codewords, for $i = 1, \ldots, r$, where r is the number of sections. For example, table 2 shows the available codewords at level four in the tree of Figure 1.(b). These codewords appear at the second column with the sections separated by commas. In this case, the first section consists of the first codeword bit, the second section consists of the second and the third codeword bits, and the last section consists of the last codeword bit. The third column in this table shows the corresponding parity of each section of bits found in the second column. The ith bit gives the parity of the ith section, for $i = 1, \ldots, r$. Let us refer to each sequence of section parities as a *signature*.

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 2, the first and the last columns of this table show 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 g and the other signatures. The error leaves are moved to the positions that correspond to signatures whose Hamming distance to g is one. The internal nodes are moved to the remaining positions. In

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

the example of table 2, since $g = 111$, the signatures 011, 101 and 110 shall be reserved to error leaves. These signatures correspond to the positions 3, 5 and 8.

Next, we formalize the Spread procedure. Let T be a code-and-error tree. At the ith iteration of the procedure, we denote by ℓ_i the level of T being processed, for $i = 1, \ldots, I$. At this level, the available codewords are denoted by $x_1^i, \ldots, x_{\alpha_i}^i$. Each codeword x_j^i can be written as the concatenation of r_i sections. For that, we use the notation $x_j^i = (s_1^{i,j}, \ldots, s_{r_i}^{i,j})$, where $s_k^{i,j}$ denotes the kth section of codeword x_j^i, for $k = 1, \ldots, r_i$. The length of the kth section of any codeword in the ith iteration is denoted by $\ell_k^i = |s_k^{i,1}| = \cdots = |s_k^{i,\alpha_i}|$, for $i = 1, \ldots, I$ and $k = 1, \ldots, r_i$. Observe that $\ell_i = \sum_{k=1}^{r_i} \ell_k^i$. Let us use $\pi(x)$ and $d(x, y)$ to denote the parity of the bit string x and the Hamming distance between the two bit strings x and y, respectively. The signature of a given position label x_j^i at the ith iteration is denoted by $g(x_j^i) = (\pi(s_1^{i,j}), \ldots, \pi(s_{r_i}^{i,j}))$. At the ith iteration, the Spread procedure performs the following five steps:

Step 1: Choose the section lengths $\ell_1^i, \ldots, \ell_{r_i}^i$ for level ℓ_i;

Step 2: Choose a signature g^i for the codeword leaves.

Step 3: Move each codeword leaf to a distinct position whose label x_j^i satisfies $g(x_j^i) = g^i$. There must be enough such positions (say codeword positions).

Step 4: Move each error leaf to a distinct position whose label x_j^i satisfies $d(g(x_j^i), g^i) = 1$. There must be enough error leaves to fill all such positions (say error positions).

Step 5: Move the remaining error leaves and all internal nodes to the remaining positions.

The criterion used to choose the section lengths in Step 1 is a subject of Section 2.1.

The correctness of the Spread procedure is a consequence of the following key proposition.

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

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

Observe that the Spread procedure places all codeword leaves in codeword positions. Moreover, by the previous proposition, it places error leaves in all positions that are neighbors of at least one codeword position. This is enough to ensure that the tree obtained from T after the execution of the Spread procedure is a Hamming tree.

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. Observe that each iteration of the Spread procedure is guaranteed to succeed if there are both enough error leaves to fill all error positions and enough codeword positions to accommodate all codeword leaves. For the example, the codeword positions and signatures given by Table 2 may also be used to rearrange a different configuration of nodes: 1 codeword leaf, 4 error leaves and 3 internal nodes. In this case, the Spread procedure does the following:

1. it places the only codeword leaf in 1 of the 2 codeword positions (1011 ou 1101);
2. it places 3 of the 4 error leaves in the 3 error positions (1010, 1100 and 1111);
3. it places the remaining nodes (1 error leaf and 3 internal nodes) arbitrarily in the remaining positions.

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 the method 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 m_i and b_i be the number of codeword leaves and the number of error leaves at level ℓ_i of the input tree T, respectively. If $b_i \geq (\ell_i - \lceil \log m_i \rceil)(2m_i - 1)$ 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 a signature that maximizes the number of corresponding available positions. If the number of available positions is larger than $2m_i - 1$, then choose an arbitrary codeword section larger than one 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 $2m_i - 1$ then stop. In this case, the current section lengths and the currently chosen signature are used.

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

First, for each signature g, let us denote by $S_j(g)$ the set of all available positions whose signature is g after the jth iteration of the previous method. Let us also define $S_0(0)$ (resp. $S_0(1)$) as the set of all available positions whose labels have even (resp. odd) parity. Finally, let η_j be the maximum cardinality of $S_j(g)$ among all possible signatures for g after the jth iteration. Observe that, when a codeword section is split at a given iteration $j+1$, every set $S_j(g)$ is partitioned into two subsets $S_{j+1}(g_1)$ and $S_{j+1}(g_2)$. As a result, we have $\eta_{j+1} \geq \eta_j/2$ for each iteration j except the last one. Since the previous method only splits a codeword section when $\eta_j \geq 2m_i$, we obtain that $\eta_{j^*} \geq m_i$, where j^* is the number of iterations performed.

Now, observe that $\eta_{j^*} \leq 2^{\ell_i - r_i}$, where r_i is the number of chosen codeword sections. This is true because each available codeword has ℓ_i bits from which $\ell_i - r_i$ codeword bits may assume any configuration without changing the signature provided the other r_i bits are updated accordingly. As a result, we must have $r_i \leq \ell_i - \lceil \log m_i \rceil$. Since we have exactly r_i sets of error positions, each one with cardinality not greater that $\eta_{j^*} \leq 2m_i - 1$, we have at most $(\ell_i - \lceil \log m_i \rceil)(2m_i - 1)$ error positions. ■

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 which will be detailed later:

Step 1: Construct a strictly binary tree T_1 with n leaves whose height is at most $L = 2\lceil \log n \rceil$.

Step 2: Obtain an expanded tree T_2 from T_1 with $n_2 > n$ leaves and set $n_2 - n$ leaves as error leaves.

Step 3: Apply the Spread procedure on T_2 to obtain a Hamming tree T_3.

In Step 1, the Raising algorithm uses a modified version of the BRCI algorithm [6] to construct an L-restricted binary tree T_1 whose weighted external path length exceeds the entropy by at most $1 + 1/n$. This modified method is described in Section 3.2. In Step 2, for each level ℓ in T_1, the Raising algorithm raises the leaves from level ℓ to a higher level in such a way that there is enough room to place the necessary error leaves without affecting the other leaves of T_1. Let T_2 be the resulting code-and-error tree. Also, let us refer to this step as the *Raising Step*, which is detailed in Section 3.3. Finally, in Step 3, we apply the Spread procedure (described in Section 2) on T_2 to obtain a Hamming tree T_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 strictly binary tree where the probability of any node at a given level ℓ is bounded above by $K/2^\ell$ for some constant K. 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.

Proposition 2 *The probability of a node at level ℓ of T_1 is bounded above by $1/2^{\ell-2}$.*

Proof: See Appendix A. ■

Proposition 3 *The weighted external path length of T_1 is at most $H + 1 + 1/2^{L-\lceil \log n \rceil - 2}$.*

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 ℓ of T_1 and insert the corresponding error leaves. Basically, this step consists of replacing each leaf of T_1 by a complete binary tree with height $\bar{\ell}$. Let m_ℓ be the number of leaves at level ℓ in T_1. After the previous transformation, $m_\ell 2^{\bar{\ell}}$ leaves are available at level $\ell + \bar{\ell}$. Then, the method sets m_ℓ of

these leaves as codeword leaves and the remaining $m_\ell(2^{\bar{\ell}} - 1)$ leaves as error leaves. The following theorem gives a suitable value for $\bar{\ell}$ as a function of both m_ℓ and ℓ that satisfies the condition of Theorem 1.

Theorem 2 *If* $\ell \geq \lceil \log m_\ell \rceil$ *and* $\bar{\ell} = \lceil \log(\ell - \lceil \log m_\ell \rceil + 2) \rceil + 2$, *then*

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

Proof: We divide it into two cases:

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

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

We analyze Case 1 for all possible values of $\ell - \lceil \log m_\ell \rceil$. 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.

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

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

$$2m_\ell(\ell - \lceil \log m_\ell \rceil) + 2m_\ell(\ell - \lceil \log m_\ell \rceil + 1.5) \tag{2}$$

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

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

Observe that (2) is not smaller than (3) whenever $\bar{\ell} \leq \ell - \lceil \log m_\ell \rceil + 1.5$, which is true whenever $\ell - \lceil \log m_\ell \rceil \geq 4$. As a result, we have that (1) is satisfied for Case 2. ■

Observe that, depending on the value of m_ℓ, we may have a leaf x_1 at a higher level than another leaf x_2 in T_1, while the same leaf x_1 is at a lower level than x_2 in T_2. In this case, the Spread procedure applied in the Step 3 of the Raising algorithm will process x_1 before x_2. Moreover, exchanging x_1 and x_2 in T_2 might improve the weighted external 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 x_1 and x_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 x_1 and x_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 $O(n \log n)$ time. As a result of this step, we have a sorted list of leaf levels. Then, in Step 2, each sublist of m_ℓ equal leaf levels in the input generates $m_\ell 2^{\bar{\ell}}$ 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 strictly 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 T_2. In each iteration of this procedure, we must choose the sections lengths for the nodes at the current level ℓ of T_2 using the method described in the proof of Theorem 1. In each iteration of this method, we traverse the whole tree until level ℓ 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 H.

Proof: Let $\bar{p}_\ell$ be the sum of the probabilities of all leaves at level ℓ of T_1. By the 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

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

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

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

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

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

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 $\bar{p}_\ell > 0$). As a result, we have that $2f((x+y)/2) > f(x) + f(y)$. 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 3 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 3: 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 $H \approx 18.9515$ 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 k-bit Hamming prefix code problem, for $k > 2$. 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 k-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 T. First, it removes from T all leaves at levels not smaller than L. Then, it raises the subtree rooted at the node x with the smallest probability at level $\ell = L - \lceil \log n \rceil - 1$ to level $\ell + 1$. Finally, it inserts in T a complete binary tree T' containing all removed leaves so that the root of T' becomes sibling of x. Since the height of T' is bounded above by $\lceil \log n \rceil$, the level of each inserted leaf in the resulting tree $\bar{T}$ is at most $\ell + 1 + \lceil \log n \rceil = L$.

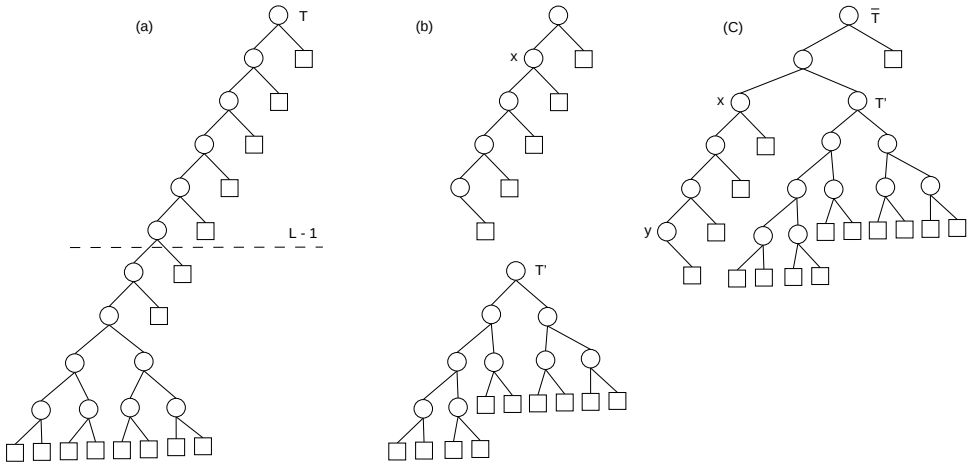

Figure 2: The BRCI algorithm.

Figure 2 illustrates a BRCI construction for $L = 6$ and $n = 15$. Figure 2.(a) shows an example of a Huffman tree T. Figure 2.(b) represents both the tree T after the removal of the leaves at levels higher than $L-1$ and the complete binary tree T' built with the removed leaves. Observe in this figure that the node x is chosen at level one and that the height of T' is exactly $\lceil \log n \rceil = 4$. Finally, Figure 2.(c) shows the resulting tree $\bar{T}$ obtained after raising the node x and inserting T'. Observe in this figure the $\bar{T}$ is not a strictly binary tree because the node y has only one child. In this case, we shall remove the node y and connect its child as a child of its parent. Let us assume that the BRCI algorithm always transforms the resulting binary tree into a strictly 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 strictly 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 $H = -\sum_{i=1}^{n} p_i \log p_i$. Hence, an immediate way to find integer codeword lengths whose average is at most one bit larger than H is setting $\ell_i = \lceil -\log p_i \rceil$. Our construction uses the previous idea to replace the Huffman tree T used by the BRCI algorithm. Instead, we construct a strictly 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 strictly binary tree whose leaves are at the levels $\ell_1, \ldots, \ell_n$. Let $\hat{T}$ be this tree.

Now, let T_1 be the tree constructed by the BRCI algorithm by rearranging the nodes of $\hat{T}$ instead of T. Observe that the overall construction runs in $O(n \log n)$ 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 $\hat{T}$ 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 $\hat{T}$. Next, we prove this claim by induction on n. For $n = 1$, the claim is true because $\hat{T}$ 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 $\hat{T}$. 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 $\hat{T}$. 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 $\hat{T}$.

Since the BRCI algorithm increases the level of each node in $\hat{T}$ by at most one, we obtain that the probability of a node at level ℓ in T_1 is bounded above by $1/2^{\ell-2}$ ■

Proof of Proposition 3: By the previous discussion, the weighted external path length of $\hat{T}$ is bounded above by $H + 1$. The BRCI algorithm increases by one the level of the leaves in the subtree of $\hat{T}$ rooted at x, 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 $\hat{T}$ due to the BRCI algorithm is at most the probability of x. By the proof of proposition 2, this probability is bounded above by $1/2^{\ell-1} = 1/2^{L-\lceil \log n \rceil - 2}$. ■