An A^*-Based Algorithm for Constructing Reversible Variable Length Codes with Minimum Average Codeword Length

Yuh-Ming Huang, *Member, IEEE*, Ting-Yi Wu, and Yunghsiang S. Han, *Senior Member, IEEE*

***Abstract*—Variable length codes (VLCs) are widely adopted in many compression standards due to their good coding efficiency on average codeword length. However, an inherent problem with a VLC is that an error of even one bit can cause serious error propagation and thus loss of synchronization at the receiver, which would lead to a series of non-correctly decoded symbols. Reversible variable length codes (RVLCs) were introduced to significantly mitigate this phenomenon. In this work, a method to find an optimal RVLC in terms of the minimum average codeword length is first formulated as a tree-searching problem, and then, instead of performing an exhaustive search, an A^*-based construction algorithm is proposed to find an optimal RVLC. The proposed algorithm has been applied to several benchmarks for sources and has found respective optimal symmetric and asymmetric RVLCs.**

***Index Terms*—Source coding, reversible variable length codes, A* algorithm.**

I. Introduction

THE Huffman class of Variable Length Codes (VLCs) have been shown to be highly efficient entropy codes and are therefore widely adopted in many well-known compression standards, including JPEG and MPEG-2. However, a Huffman-encoded bitstream is vulnerable to bit errors. When transmitted over a noisy channel, even one bit error in the bitstream may cause serious error propagation, and thus synchronization loss at the receiver. This phenomenon leads to decoding failure and produces a series of non-correctable symbols. Several reversible variable length codes (RVLCs) [1]–[12], which were modified from VLCs, have been presented to mitigate this phenomenon by admitting instantaneous decoding in either direction (forward and backward), but at the cost of compression loss. In recent years, more compression standards, such as JPEG-2000, H.263+, and MPEG-4, have adopted RVLCs to enhance their error-resilient capabilities. Furthermore, as shown elsewhere [13]–[17], although an RVLC increases code redundancy, which represents the difference between the entropy and the average codeword length, this redundancy can be regarded as a means of implicit channel protection and can be used by the decoder to provide the error-correcting capacity of the RVLC.

An RVLC is a fix-free code in which no codeword is a prefix or a suffix of any other codeword. Hence, an RVLC-encoded bitstream can be instantaneously decoded in both the forward and backward directions. RVLCs can be divided into two categories - *symmetric* and *asymmetric*. A symmetric RVLC has an additional constraint - that the two bitstreams respectively obtained by parsing a codeword in the forward and backward directions are identical. In contrast, an asymmetric RVLC does not have this property. Because of the additional constraint, symmetric RVLCs are associated with smaller search spaces than asymmetric RVLCs. Hence, this constraint simplifies the design of optimal codes. As expected, an asymmetric RVLC never provides less coding efficiency than a symmetric one, since codewords can be selected more flexibly in the construction of an asymmetric RVLC.

Algorithms for constructing RVLCs typically follow two methodologies. One constructs an RVLC with its average codeword length as close to that of the Huffman code (with the best coding efficiency) as possible. The other reduces the size of the search tree to easily locate an optimal (or a nearly optimal) solution. [1]

Based on a given Huffman code and its codeword length distribution, Takishima *et al.* [1] first specified how to construct both asymmetric and symmetric RVLCs. Later, Tsai *et al.* [2], [3] improved methods that were proposed elsewhere [1] using heuristic codeword selection mechanisms. In two other works, [4] and [6], further improvements were achieved using another heuristic codeword selection mechanism for asymmetric RVLCs. At each level of the code tree, these

Paper approved by J. Kliewer, the Editor for Iterative Methods and Cross-Layer Design of the IEEE Communications Society. Manuscript received April 6, 2009; revised January 6, 2010, April 14, 2010, and June 25, 2010.

This work was partly supported by the National Science Council, Taiwan, R.O.C., under the projects (No. NSC95-2221-E-260-029-MY2 and No. NSC97-2221-E-260-014).

Y. M. Huang is with the Department of Computer Science and Information Engineering, National Chi Nan University, Puli, Taiwan, R.O.C. (e-mail: ymhuang@csie.ncnu.edu.tw).

T. Y. Wu was with the Department of Computer Science and Information Engineering, National Chi Nan University, Puli, and is now with the Department of Electrical Engineering, National Chiao-Tung University, Hsinchu, Taiwan, R.O.C. (e-mail: mavericktywu@gmail.com).

Y. S. Han is with the Department of Electrical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, R.O.C. He was with the Graduate Institute of Communication Engineering, National Taipei University, Taipei County, and with the Department of Computer Science and Information Engineering, National Chi Nan University, Puli Taiwan, R.O.C. (e-mail: yshan@mail.ntust.edu.tw).

Digital Object Identifier 10.1109/TCOMM.2010.091710.0901872

[1] In Section II, the problem of constructing an RVLC is converted into a tree-searching problem, in which all of the asymmetric RVLCs can be represented in binary tree $\mathcal{T}$.

The process of implementing each existing construction algorithm can be described by its own search tree, for example Fig. 3 of [9]. Essentially, this search tree can be seen as a subset of binary tree $\mathcal{T}$ if each branch of Fig. 3 is decomposed into a series of smaller branches.

algorithms assign a number of RVLC codewords that is as close to the number of Huffman codewords as possible. While more available RVLC codewords can be chosen, a heuristic codeword selection mechanism is applied to guide the selection procedure. When fewer available RVLC codewords can be chosen, the length of some RVLC codewords must be increased, and so the coding efficiency of the constructed RVLC is worse than that of a Huffman code. Basically, a trade off exists between the coding efficiency and the time-complexity of the codeword selection mechanism. Since a heuristic codeword selection mechanism with higher time-complexity may explore more information, it is more likely to find an optimal code. Among the algorithms in [3], [4], and [6], that in [4] has the highest time-complexity and that in [3] has the lowest time-complexity. For example, for the Canterbury Corpus file set, the search results given in [6] are almost better than those given in [3]. Owing to its exponential time-complexity, Lin *et al.* [6] showed that the algorithm given in [4] can only be applied to some source benchmarks taken from the Canterbury Corpus file set. Moreover, the codeword selection mechanism in [4] is not an optimal scheme that always results in an optimal code. Using the algorithms in [3] and [4], Wang *et al.* [5] improved the search results given in those works, in relation to the coding efficiency, by iteratively optimizing the distribution of the codeword lengths.

Rather than utilizing a fixed distribution of codeword lengths associated with a given Huffman code, Tseng *et al.* [7] proposed a new approach for constructing symmetric RVLCs by employing a backtracking technique. This technique eliminates one codeword from a currently constructed RVLC codeword list, such that more available shorter RVLC codewords can be selected. Accordingly, a new codeword list of the same size, which has a shorter average codeword length, can be generated. Later, Lin *et al.* [8] proposed two backtracking-based construction algorithms for asymmetric RVLCs, whose results improved those presented in [3], with either smaller average codeword lengths or shorter maximum codeword lengths. In 2004, Tseng *et al.* [9] extended the algorithm that they had proposed in [7] to construct asymmetric RVLCs. However, some of their constructed codes given in [9] were still worse than those found by the Huffman code-based algorithm [5] in terms of coding efficiency. By exploiting the property of a convex function, Jeong *et al.* [10] constructed asymmetric RVLCs with greater coding efficiency than those given in [8] and [5]; however, some constructed asymmetric RVLCs have a larger average codeword length than those given in [9]. Many of the above algorithms have the phenomenon that different initial values must be assigned heuristically according to the source statistics. Moreover, none of the aforementioned methods is superior to the others for all search results.

None of the above construction algorithms can be guaranteed to find an optimal RVLC because the corresponding search tree of each algorithm is not always large enough to capture an optimal solution. Although an exhaustive search on binary tree $\mathcal{T}$ can find an optimal RVLC, its use is impractical because of the extremely high time-complexity.

Recently, Savari [11] proposed two approaches for finding optimal symmetric RVLCs. First, she extended the idea given in another work [18] to RVLCs, and found that, a small binary symmetric RVLC has few dominant length vectors. After all of these dominant length vectors are found, they can be tested to determine which is the best in terms of the minimum average codeword length for a given source probability vector. This approach just requires a one-time computational cost to find all of the dominant length vectors and can easily construct a new optimal symmetric RVLC when the source probability vector is changed. Although Savari also pointed out two properties that may be utilized to quickly generate all of the dominant length vectors and their corresponding codes, no explicit algorithm was given in her original work. An explicit algorithm was given in her later work [19]. Unlike our proposed algorithm, which can find optimal symmetric RVLCs with large sizes, only optimal symmetric RVLCs with small sizes were given in [11]. Second, this work proposed a searching scheme that traverses the set of various partial length vectors in order to locate the full length vector associated with an optimal code. Given a source probability vector, a threshold can be calculated that is equal to the average codeword length of a known best symmetric RVLC. Apparently, this threshold is an upper bound on the average codeword length of an optimal code. According to the source probability vector and the threshold, a partial length vector (and all of its possible extensions) is eliminated to speed up the search after it is determined to not have an extension corresponding to an optimal code. However, unlike our proposed algorithm, which can find optimal symmetric RVLCs with large sizes, only an optimal symmetric symmetric RVLC for the English alphabet was given in [11].

In [12], Savari first proposed two low-complexity heuristic approaches for constructing sub-optimal asymmetric RVLCs for the English alphabet. Although the resultant RVLCs were not better than those given in [9], Savari pointed out that the Kraft-sum of a code is an important heuristic parameter for the design of an optimal code. She then proposed two "bounding" schemes that refined the searching approach given in [11] and applied them to find optimal asymmetric RVLCs. She also showed that the decision and assignment problem "to determine whether or not there is a fix-free code for a given set of codeword lengths and a procedure to generate such a code if it exists" could be converted into an equivalent Boolean satisfiability problem (SAT), which is an NP-complete problem [20]. As Savari pointed out, the open source SAT solver (http://een.se/niklas/Satzoo) was always able to quickly determine satisfiability and provide a corresponding bit assignment for a given length vector. However, even though it could sometimes quickly determine unsatisfiability, it was often extremely slow in doing so. Moreover, the author didn't explicitly give a systematic and efficient algorithm to determine the order of the length vectors checked through the SAT solver in order to quickly eliminate dominated length vectors and easily find all of the dominant length vectors. An explicit algorithm was given in her later work [19]. How to extend and traverse the partial length vectors more efficiently in order to quickly find the optimal code was also not provided. Hence, only an optimal RVLC for the English alphabet was found in [12] and no optimal RVLCs with large sizes for the Canterbury Corpus file set were presented.

In this work, all of the asymmetric RVLCs of size N are represented by a binary tree, and then, an A^*-based sequential algorithm is proposed to search for an asymmetric RVLC with a minimum average codeword length, as an alternate to an exhaustive search. The proposed algorithm can also be applied to find an optimal symmetric RVLC with far lower time-complexity by considering the additional "symmetry" constraint. It is the first searching approach described in complete detail and where the algorithm solves larger problems than those discussed in Savari's papers.

This paper is organized as follows. Section II presents a binary-tree representation of all of the RVLCs and converts the problem of constructing an optimal RVLC into a tree-searching problem. Section III presents the proposed A^*-based sequential algorithm along with its search tree and proves that the proposed algorithm will always find an optimal RVLC. Section IV compares the search results obtained using the proposed method with those presented in previous works [5], [7], [9], [11], [12]. The time-complexity and memory requirement of the proposed algorithm are also evaluated. Conclusions are finally drawn in Section V, which also includes recommendations for future research.

II. Binary-tree Representation of RVLCs

Let $\mathcal{S}$ be a set of N independent source symbols $\{s_1, s_2, s_3, \cdots, s_N\}$ with respective to occurrence probabilities $\{p_1, p_2, p_3, \cdots, p_N\}$ in decreasing order. Let Ψ be the set of all RVLCs such that each RVLC in Ψ has N codewords. Let $\mathcal{C} = \{c_1, c_2, c_3, \cdots, c_N\} \in \Psi$ and the length of each codeword in $\mathcal{C}$ is respectively denoted as $\ell(c_i)$, $1 \le i \le N$. The goal of our construction algorithm is to find a code $\hat{\mathcal{C}}$ corresponding to $\mathcal{S}$ with the minimum average codeword length, that is:

$$\hat{\mathcal{C}} = arg \min_{\mathcal{C} \in \Psi} \sum_{i=1}^{N} p_i \times \ell(c_i). \tag{1}$$

All of the RVLCs in Ψ can be represented in binary tree $\mathcal{T}$ where each leaf node of the tree denotes an RVLC. In other words, each path of the tree explicitly depicts the process of constructing an RVLC. Fig. 1 shows a simple example for N=3, where codeword list $\mathcal{C}$ and candidate list $\mathcal{A}$ are used to describe the process. $\mathcal{C}$ records the currently selected codewords and $\mathcal{A}$ is a list of legitimate codeword candidates in lexicographic order. There are two branches leaving from each node. The lower branch shows that the first available candidate in $\mathcal{A}$ is selected, while the upper branch shows that this candidate is non-selected. Let us consider the thicken path shown in Fig. 1. Initially, $\mathcal{C}$ and $\mathcal{A}$ are respectively assigned to be $\emptyset$ and $\{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \cdots\}$. In the first branch of the path, the first codeword "0" in $\mathcal{A}$ is selected and hence codeword "0" is eliminated from $\mathcal{A}$ and included in $\mathcal{C}$. $\mathcal{A}$ is further updated by removing the codewords $\{00, 01, 10, 000, \cdots\}$ that are not prefix-free or suffix-free with respect to codeword "0." In the second branch of the path, the first codeword "1" in $\mathcal{A}$ is not selected. So the content of list $\mathcal{C}$ is unchanged and "1" is eliminated from $\mathcal{A}$. In the third branch of the path, the first codeword "11" in $\mathcal{A}$ is selected and the codeword "11" is included in $\mathcal{C}$.

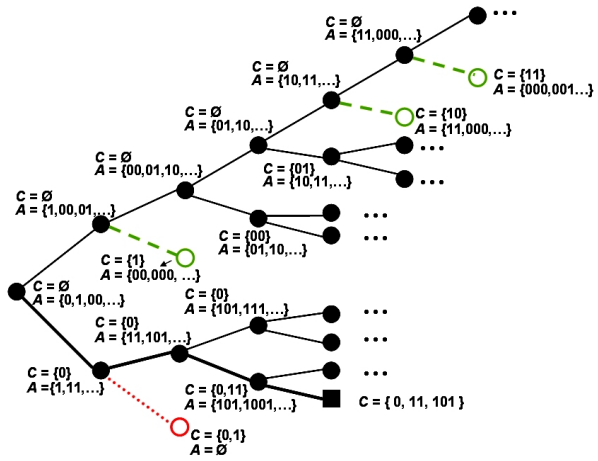

Fig. 1. All of the asymmetric RVLCs of size 3 are represented in binary tree $\mathcal{T}$. ■ shows a leaf in $\mathcal{T}$.

The codewords $\{111, 1011, \cdots\}$ are also excluded from $\mathcal{A}$ to maintain the fix-free property for each codeword in $\mathcal{A}$ with respect to codeword "11." In the final branch of the path, after selecting the codeword "101" from $\mathcal{A}$, we obtain an RVLC as $\{0, 11, 101\}$. The dotted branch shown in Fig. 1 indicates that it will not be expanded since list $\mathcal{A}$ will become empty after the codeword "1" is selected.

A complement of a code (a set) is the code that contains complements of all of the elements of this code (set). Since an RVLC and its complement have the same average codeword length, the complement of each RVLC is disregarded here. Therefore, for each newly expanded node, one can simply stipulate that the first selected codeword in C is required to start with a zero-valued bit. Therefore, as shown in Fig. 1, the dashed branches are pruned to further simplify the binary tree $\mathcal{T}$. In fact, in our implementation, only N-$|C|$ candidates[2] are kept in list $\mathcal{A}$, where $|C|$ denotes the number of selected codewords in C. Once binary tree $\mathcal{T}$ is constructed, finding the desired $\hat{\mathcal{C}}$ becomes a tree-search problem in $\mathcal{T}$.

The A^* algorithm [21] is a very famous search algorithm on a graph in artificial intelligence. In the graph, a cost is associated with each branch and the algorithm finds the shortest path from the starting node to a goal node with minimum path cost, which is the accumulated branch costs along the path. A heuristic function is imposed to the cost of each path in the graph to reduce the search space. When a better heuristic function is used, the search visits fewer nodes. The cost of each path (partial or full) is composed of two parts: the accumulated cost from the starting node to the ending node of the path and the heuristic function that estimates the cost from the ending node of the path to a goal node. In Section III, an A^*-based algorithm for constructing optimal RVLCs will be presented after the cost of each node is specified. It will be proved that the proposed algorithm always finds an optimal RVLC as a shortest path is found by the A^* algorithm.

Without loss of generality, the relationship between a parent node and its two children nodes in $\mathcal{T}$ is illustrated in Fig. 2,

[2]The actual contents of $\mathcal{A}$ change adaptively according to the codewords selected in C. Our goal is to find a code of size N. Hence, it is just enough to keep the first N-$|C|$ candidates during the implementation.

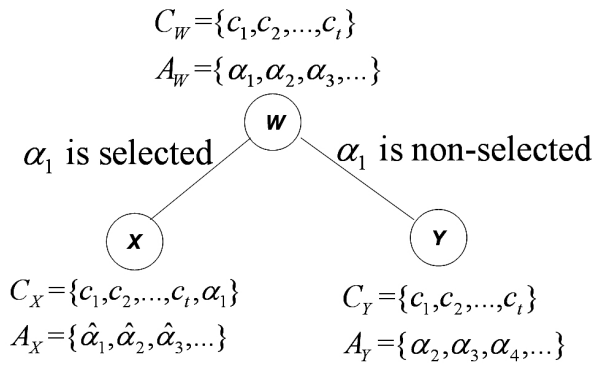

Fig. 2. Relationship between a parent node and its two children nodes in $\mathcal{T}$.

where the costs of nodes W, X, and Y are specified.[3] Since our goal is to find an optimal RVLC, we need to specify the cost of each node in the binary tree according to (1). By constructing $\mathcal{T}$, it is easy to see that the branch cost assigned to each branch should be $p_i \times l(c_i)$, if c_i is selected in this branch, or 0, if no codeword is selected. Let $m(W)$ denote the cost of node W and define it as:

$$m(W) = \underbrace{\sum_{i=1}^{t} p_i \times l(c_i)}_{g(W)} + \underbrace{\sum_{i=t+1}^{N} p_i \times l(\alpha_{i-t})}_{h(W)}, \quad (2)$$

where $g(W)$ denotes the accumulated branch cost from the root of $\mathcal{T}$ (the starting node) to node W (i.e., the accumulated average codeword length over the codewords c_i's with respect to their occurrence probabilities p_i's, $1 \le i \le t$), and $h(W)$ denotes an estimate on the cost of the path from node W to any goal (leaf) node. Note that we design $h(W)$ as the average codeword length over the $N-t$ shortest codewords among all of the remaining codewords that can be legally selected while traversing any path starting from node W. Let $h^*(W)$ denote the minimum cost among all of the possible paths from node W to a goal node. It can be easily observed that:

$$h(W) \le h^*(W)$$

and

$$g(W) + h(W) \le g(W) + h^*(W), \quad (3)$$

i.e., $h(W)$ is a lower bound of $h^*(W)$.

As shown in Fig. 2, node W is associated with two lists, $\mathcal{C}_W$ and $\mathcal{A}_W$. The child node X on the left hand side adds the first codeword α_1 in $\mathcal{A}_W$ into $\mathcal{C}_W$. Thus, the codeword list is now:

$$\begin{aligned} \mathcal{C}_X &= \mathcal{C}_W \cup \{\alpha_1\} \\ &= \{c_1, c_2, \cdots, c_t, \alpha_1\}. \end{aligned} \quad (4)$$

[3]Since we deal with a tree in our work, the ending node is used to represent a path. Hence, the cost associated with the path is the same as that associated with its ending node.

Next, the candidate list $\mathcal{A}_W$ is updated as:

$$\mathcal{A}_X = \{\hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha}_3, \cdots\} \quad (5)$$

by removing all non-fix-free codewords in $\mathcal{A}_W$ with respect to α_1. Hence, the cost of node X is equal to:

$$m(X) = \underbrace{[\sum_{i=1}^{t} p_i \times l(c_i)] + p_{t+1} \times l(\alpha_1) +}_{g(X)} \underbrace{\sum_{i=t+2}^{N} p_i \times l(\hat{\alpha}_{i-t-1})}_{h(X)}. \quad (6)$$

For child node Y on the right hand side, the content of the codeword list is unchanged, i.e., $\mathcal{C}_Y = \mathcal{C}_W$. $\mathcal{A}_Y$ is obtained by just eliminating the codeword α_1 from $\mathcal{A}_W$, i.e.,

$$\begin{aligned} \mathcal{A}_Y &= \mathcal{A}_W \setminus \{\alpha_1\} \\ &= \{\alpha_2, \alpha_3, \alpha_4, \cdots\}. \end{aligned} \quad (7)$$

Then, the cost of node Y is calculated as:

$$m(Y) = \underbrace{\sum_{i=1}^{t} p_i \times l(c_i)}_{g(Y)} + \underbrace{\sum_{i=t+1}^{N} p_i \times l(\alpha_{i-t+1})}_{h(Y)}. \quad (8)$$

III. A^*-BASED ALGORITHM FOR CONSTRUCTING OPTIMAL RVLCS

Equipped with the definitions and notations given in Section II, an A^*-based construction algorithm for designing optimal RVLCs with respect to the minimum average codeword length is presented as follows:

$< Algorithm\ 1 >$

Step 1. *Set $Ubound$ equal to the average codeword length of an existing RVLC.*
Initially, the Stack has only one root node R with

$$\mathcal{C}_R = \emptyset, \mathcal{A}_R = \{\alpha_1 = 0, \alpha_2 = 1, \alpha_3 = 00, \alpha_4 = 01, \alpha_5 = 10, \alpha_6 = 11, \cdots\}$$

and

$$g(R) = 0, m(R) = h(R)(= \sum_{i=1}^{N} p_i \times l(\alpha_i)).$$

Step 2. *Pop the top node W (for the first time $W = R$) from the Stack.*
If $\mathcal{C}_W \cup \{\alpha_1, \alpha_2, \cdots, \alpha_{N-t}\}$ is a valid RVLC, then the algorithm stops and we obtain an optimal RVLC.

Step 3. *Generate two children nodes X and Y of node W and construct the lists $\mathcal{C}_X$, $\mathcal{A}_X$ and $\mathcal{C}_Y$, $\mathcal{A}_Y$.*
If $\mathcal{A}_X$ is empty and the size of $\mathcal{C}_X$ is less than N, then discard node X.
If the first selected codeword in $\mathcal{C}_X$ does not start with a zero-valued bit, then discard node X.
Calculate the cost values of the remaining children nodes.

If the cost value of X (or Y) is greater than $Ubound$, then discard node X (or Y).

Step 4. Insert the remaining children nodes into the Stack and reorder the Stack according to ascending cost values. Go to Step 2.

We next prove that the proposed algorithm will always construct an optimal RVLC. That is, sequential-type searching in the above algorithm can locate a leaf node with minimum cost in $\mathcal{T}$.

Lemma 1: The cost at each node is a non-decreasing function along any path in $\mathcal{T}$.

Proof: As shown in Fig. 2, we need to show that $m(W) = g(W) + h(W) \leq m(X) = g(X) + h(X)$ and $m(W) \leq m(Y) = g(Y) + h(Y)$. Note that the candidate list is in lexicographic order. Then, we have $l(\alpha_2) \leq l(\hat{\alpha}_1)$, $l(\alpha_3) \leq l(\hat{\alpha}_2), \cdots, l(\alpha_{N-t}) \leq l(\hat{\alpha}_{N-t-1})$ due to the fact that A_X is a subset of A_W. Hence,

$$\begin{aligned} m(W) &= \sum_{i=1}^{t} p_i \times l(c_i) + \sum_{i=t+1}^{N} p_i \times l(\alpha_{i-t}) \\ &\leq \left(\sum_{i=1}^{t} p_i \times l(c_i) + p_{t+1} \times l(\alpha_1) \right) + \\ &\quad \sum_{i=t+2}^{N} p_i \times l(\hat{\alpha}_{i-t-1}) \\ &= g(X) + h(X) = m(X). \end{aligned}$$

Similarly we have $m(W) \leq m(Y)$. ■

By Lemma 1, the cost along any path is non-decreasing. Then, any average codeword length of an existing RVLC can be an upper bound on the cost of an optimal path in the binary tree [11], [12], [22]. This upper bound can be used to reduce the size of the stack since any newly generated node with a cost that is not less than this upper bound can be eliminated from the stack. By using this technique, the size of the stack can be shortened and the search can be sped up at the same time. This speed-up is benefited from the fact that a shorter stack has a faster sorting procedure than a longer one.

Theorem 1: The proposed A^*-based Algorithm always finds an optimal RVLC.

Proof: As indicated in Step 2 of the above algorithm, if $\mathcal{C}_W \cup \{\alpha_1, \alpha_2, \cdots, \alpha_{N-t}\}$ is a valid RVLC, the algorithm stops and reports that an optimal RVLC has been found. From the definition of the cost for a node, $m(W)$ is not only the cost of node W but also the average codeword length of this RVLC. The costs of other nodes are not less than $m(W)$ and, by Lemma 1, the costs of their successors are also not less than $m(W)$. Hence, any further search in the tree will not result in an RVLC with a lower minimum average codeword length. This proves that the proposed algorithm always finds an optimal RVLC. ■

Example 1: Consider a set of four independent source symbols $\{s_1, s_2, s_3, s_4\}$ with respective occurrence probabilities $\{p_1, p_2, p_3, p_4\} = \{0.68, 0.13, 0.11, 0.08\}$. The search tree and the stack contents for this example are respectively shown in Fig. 3 and Fig. 4. Initially, the stack has only one node 0 with cost $m(0) = 1.19$ ($g(0) = 0$ and $h(0) = 0.68 \times 1 + 0.13 \times 1 + 0.11 \times 2 + 0.08 \times 2 = 1.19$). In round 1, we pop out node 0 from the stack. Two children nodes (node 1 and node 2) of node 0 are generated with $m(1) = 1.27$ ($g(1) = 0.68 \times 1 = 0.68$, $h(1) = 0.13 \times 1 + 0.11 \times 2 + 0.08 \times 3 = 0.59$) and $m(2) = 1.32$ ($g(2) = 0, h(2) = 0.68 \times 1 + 0.13 \times 2 + 0.11 \times 2 + 0.08 \times 2 = 1.32$). They then are pushed into the stack in the order of ascending cost values. In round 2, node 1 is popped out from the stack and only its child on the right hand side (node 4) with $m(4) = 1.51$ ($g(4) = 0.68 \times 1$, $h(4) = 0.13 \times 2 + 0.11 \times 3 + 0.08 \times 3 = 0.83$) is generated and inserted into the stack. Since the size of $\mathcal{C}_3$ is less than 4 and list $\mathcal{A}_3$ becomes empty, we discard node 3. In round 3, node 2 is popped out from the stack and only its child on the right hand side (node 6) with $m(6) = 2$ ($g(6) = 0$, $h(6) = 0.68 \times 2 + 0.13 \times 2 + 0.11 \times 2 + 0.08 \times 2 = 2$) is generated and inserted into the stack. Since $\mathcal{C}_5 (= \{1\})$ is complemented with list $\mathcal{C}_4 (= \{0\})$ kept by node 4 in the current stack, we discard node 5. In round 4, we pop out node 4 from the stack, and then generate its two children (node 7 and node 8) with $m(7) = 1.59$ ($g(7) = 0.68 \times 1 + 0.13 \times 2 = 0.94$, $h(7) = 0.11 \times 3 + 0.08 \times 4 = 0.65$), and $m(8) = 1.72$ ($g(8) = 0.68 \times 1 = 0.68$, $h(8) = 0.13 \times 3 + 0.11 \times 3 + 0.08 \times 4 = 1.04$). Since $\mathcal{C}_7 \cup \{101, 1001\}$ is a valid RVLC, an asymmetric RVLC $\{0, 11, 101, 1001\}$ with the minimum average codeword length is obtained immediately after node 7 is popped out.

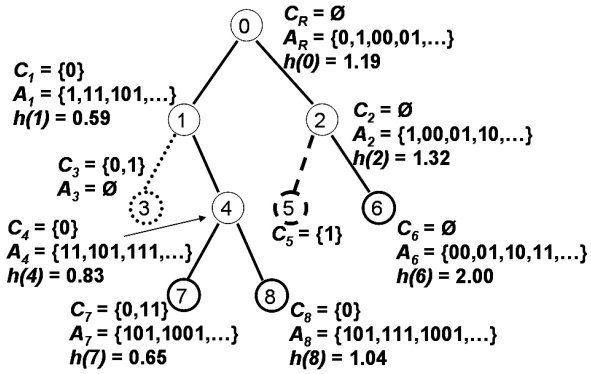

Fig. 3. The search tree in Example 1.

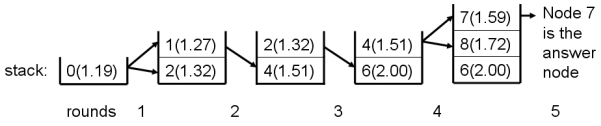

Fig. 4. Stack contents in Example 1.

It has been shown that with a larger estimated h, fewer nodes are visited in the A^* algorithm search [22], [23]. However, in order to guarantee that an optimal solution will be found in the search, the value of $h(Z)$ must be no more than $h^*(Z)$ for each node Z in $\mathcal{T}$, where $h^*(Z)$ denotes the minimum cost among all possible paths from node Z to a goal node. Increasing the value of $h(Z)$ can be easily achieved by applying the following $h-estimate$ subroutine to the subtree rooted at node Z in $\mathcal{T}$. A temporary local stack is created for node Z to store the generated nodes in the subtree while calculating $h(Z)$. A node is called *expanded* if its two children are generated in the subroutine. When the number of nodes

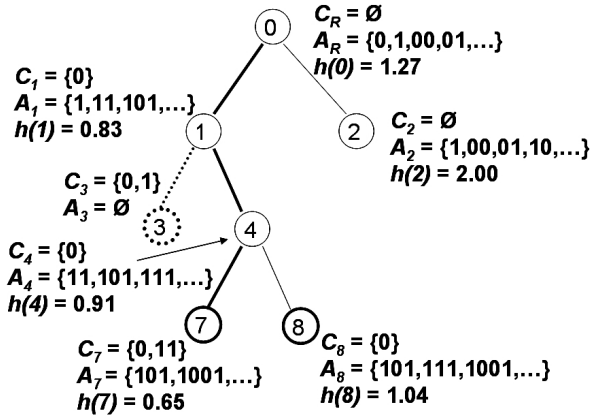

Fig. 5. The search tree in Example 1 with larger estimate.

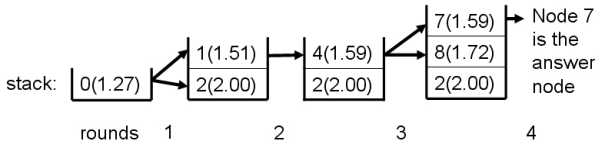

Fig. 6. Stack contents in Example 1 with larger estimate.

expanded by the subroutine, which is denoted as EN, is larger in the subtree, fewer rounds are required by $Algorithm\ 1$. In addition, this technique can be used to efficiently reduce the memory requirement. Even though the computational cost for $h(Z)$ increases when the number of expanded nodes is larger in the subtree, the overall speed of the proposed algorithm will not degrade significantly. Sometimes the overall speed even becomes faster when the EN value increases. This is due to the reduction in the number of rounds. Moreover, when EN is sufficiently large, an RVLC can be found immediately and its average codeword length is a lower bound of all of the possible RVLCs in this subtree rooted at node Z. That is, we don't need to extend node Z anymore. A detailed $h{-}estimate$ subroutine is presented as follows:

$< h{-}estimate(Z, EN) >$

// Z may represent node R, node X, or node Y in $Algorithm\ 1$.

Step 1. *Copy node Z and its $\mathcal{C}_Z$, $\mathcal{A}_Z$, and g(Z), which are calculated in Algorithm 1, into the temporary Stack. $i = 0$. //* Initially, the temporary Stack is empty.

Step 2. *Pop the top node W (for the first time $W = Z$) from the temporary Stack.*
If the size of $\mathcal{C}_W$ is equal to N-1, then $\mathcal{C}_Z = \mathcal{C}_W \cup \{\alpha_1\}$ and go to Step 5. // I.e. $m(W)$ is a lower bound of all of the possible RVLCs in this subtree rooted at node Z.
If $i = EN$, then go to Step 5.

Step 3. *Generate two children nodes, X and Y, of node W and construct the lists $\mathcal{C}_X$, $\mathcal{A}_X$ and $\mathcal{C}_Y$, $\mathcal{A}_Y$.*
If $\mathcal{A}_X$ is empty and the size of $\mathcal{C}_X$ is less than N, then discard node X.
If the first selected codeword in $\mathcal{C}_X$ does not start with a zero-valued bit, then discard node X.
If the size of $\mathcal{C}_X$ is equal to N-1 and $m(X) < Ubound$, then $Ubound = m(X)$. // Note that $\mathcal{C}_X \cup \{\alpha_1\}$ is a valid RVLC, hence $m(X)$ is a new upper bound.
If the cost value of X (or Y) is greater than $Ubound$, then discard node X (or Y).

Step 4. *Insert the remaining children nodes into the temporary Stack and reorder the temporary Stack according to ascending cost values.*
$i = i + 1$.
If the temporary Stack is empty, then return ∞. // I.e. a subtree rooted at node Z in $\mathcal{T}$ is pruned.
else go to Step 2.

Step 5. *Return m(W). //* I.e., the new estimated $h(Z)$ is equal to m(W)-g(Z).

In Section IV, different setups for EN will be simulated. For an easy illustration, the value of EN is set to one here. Hence, except for node 1 and node 2 in Fig. 3, the estimated $h(Z)$ of node Z is now replaced with $\min\{m(P), m(Q)\}$[4]-$g(Z)$, where P and Q are the two children of node Z. For example, $h(0) = \min\{m(1), m(2)\} - g(0) = 1.27$ and $h(4) = \min\{m(7), m(8)\} - g(4) = 0.91$. As for node 1 and node 2, since node 3 and node 5 don't exist, $h(1) = m(4) - g(1) = 0.83$ and $h(2) = m(6) - g(2) = 2$. By increasing the estimate, the new resultant search tree and the stack contents in Example 1 can be obtained as shown in Fig. 5 and Fig. 6. Notice that the number of rounds in Fig. 6 is reduced by one in comparison to that in Fig. 4. For the English alphabet, the number of rounds required in our proposed algorithm is reduced from 15702 to 7438 if the number of expanded nodes is set to two. Moreover, the maximum number of nodes in the stack is also reduced from 6222 to 2940.

IV. Experimental Results

The proposed algorithm has been tested on the English alphabet and various source benchmarks taken from the Canterbury Corpus file set (available at http://corpus.canterbury.ac.nz). These benchmarks were designed specifically for testing new compression algorithms. The achievable performance of the proposed algorithm and that of the algorithms presented by [7], [5], [9], and [11], [12] are listed and assessed in Table I−Table V. The Huffman code [24] is also given in all of the tables for reference. Our experiments were done using Bloodshed Dev-C++ run on an Acer Travel/Mate 6293 Notebook with an Intel Core 2 Duo P8600 2.4 GHz CPU and 3G bytes of RAM. There are two kinds of primary storage in our program. One is an array used for storing the codewords and corresponding indices, where all of the codewords in a full binary code tree are arranged in lexicographic order and indexed incrementally from zero. For example, the index of the codeword 11 is represented by 5. The other type of storage is for the information of each node W in the stack that is kept in a record, which stores the costs, $g(W)$ and $m(W)$ and the lists, $\mathcal{C}_W$ and $\mathcal{A}_W$. List $\mathcal{C}_W$ and $\mathcal{A}_W$ respectively record the indices of selected codewords and candidate codewords. The indices are chained together by pointers. Notice that the total number of indices kept in $\mathcal{C}_W$ and $\mathcal{A}_W$ is equal to N.

[4] $\min\{a, b\}$ denotes the smaller one between a and b.

TABLE I
HUFFMAN CODE AND SYMMETRIC/ASYMMETRIC RVLCS CONSTRUCTED BY THE PROPOSED ALGORITHM AND THOSE OF [7], [9], [11], AND [12] FOR THE ENGLISH ALPHABET.

English Alphabet	Probability	Huffman [24]	symmetric RVLCs		asymmetric RVLCs	
			Tseng *et al.* [7] (Savari [11])	Ours	Tseng *et al.* [9]	Ours (Savari [12])
E	0.14878570 (0.14878610)	001	000	000	101	000
T	0.09354149	110	010	010	111	011
A	0.08833733	0000	101	101	000	110
O	0.07245769	0100	111	111	0110	0010
R	0.06872164	0110	0110	0110	1001	0100
N	0.06498532	1000	1001	1001	0010	0101
H	0.05831331	1010	00100	00100	0011	1001
I	0.05644515	1110	01110	01110	0100	1010
S	0.05537763	0101	10001	10001	1100	1111
D	0.04376834	00010	11011	11011	01010	00111
L	0.04123298	10110	001100	001100	11010	10001
U	0.02762209	10010	011110	011110	01110	10111
P	0.02575393	11110	100001	100001	10001	11100
F	0.02455297	01111	110011	110011	01011	11101
M	0.02361889	10111	0010100	0010100	11011	001100
C	0.02081665	11111	0011100	0011100	011110	001101
W	0.01868161	000111	0111110	0111110	100001	100001
G	0.01521216	011100	1000001	1000001	0111110	101100
Y	0.01521216	100110	1100011	1100011	1000001	101101
B	0.01267680	011101	1101011	1101011	01111110	1000001
V	0.01160928	100111	00111100	00111100	10000001	10000001
K	0.00867360	0001100	01111110	01111110	011111110	100000001
X	0.00146784	00011011	10000001	10000001	100000001	1000000001
J	0.00080064	000110101	11000011	11000011	0111111110	10000000001
Q	0.00080064	0001101001	011111110	001010100	1000000001	100000000001
Z	0.00053376	0001101000	100000001	001101100	01111111110	1000000000001
Average codeword length		4.15572284	4.46463681 (4.46463801)	4.46463681	4.18734808	**4.17280313** (4.17280433)

TABLE II
HUFFMAN CODE AND SYMMETRIC/ASYMMETRIC RVLCS CONSTRUCTED BY THE PROPOSED ALGORITHM AND THOSE OF [7], [5], AND [9] FOR THE CANTERBURY CORPUS FILE SET.

		Symmetric RVLCs		
		Huffman [24]	Tseng *et al.* [7]	Ours
File	Number of codewords	Average codeword length		
asyoulik.txt	68	4.84464646	5.21025119	5.21025119
alice29.txt	74	4.61244402	4.93155363	4.93155363
xargs.1	74	4.92382304	5.33995697	5.33995697
grammar.lsp	76	4.66433754	5.01773571	5.01773571
plrabn12.txt	81	4.57524019	4.89526695	4.89526695
lcet10.txt	84	4.6971159	5.01682473	5.01682473
cp.html	86	5.26716254	5.81172993	5.81172993
fields.c	90	5.04089686	5.46331826	5.46331826
ptt5	159	1.66091275	1.75991768	1.75991768
sum	255	5.36503661	6.03917365	6.03917365
kennedy.xls	256	3.59337466	4.27209103	**4.21507667**

		Asymmetric RVLCs			
		Huffman [24]	Wang *et al.* [5]	Tseng *et al.* [9]	Ours
File	Number of codewords	Average codeword length			
asyoulik.txt	68	4.84464646	4.92273	4.86816504	**4.85172460**
alice29.txt	74	4.61244402	4.70569	4.65799667	**4.65015261**
xargs.1	74	4.92382304	5.00166	4.94511430	**4.93494159**
grammar.lsp	76	4.66433754	4.80247	4.72104143	**4.71539776**
plrabn12.txt	81	4.57524019	4.71036	4.63937150	**4.62281489**
lcet10.txt	84	4.69711590	4.80024	4.73408573	**4.72663178**
ptt5	159	1.66091275	1.69580	1.69717583	**1.67596446**

The codewords and average codeword lengths of symmetric/asymmetric RVLCs for the English alphabet found by the algorithm of Tseng *et al.* [7], [9] and our proposed algorithm are given in Table I. Table I shows that both our algorithm and the algorithm of [7] can find an optimal symmetric RVLC with minimum average codeword length. Our algorithm could also find an optimal asymmetric RVLC, while the algorithm of [9] did not. Note that although the Huffman code has the least average codeword length since it is an optimal code without reversible and symmetrical constraints, the average codeword length of the asymmetric RVLC found by our proposed algorithm is very close to that of the Huffman code. In [11] and [12], optimal symmetric and asymmetric RVLCs for the English alphabet were also provided. However, the average codeword lengths of these RVLCs are different from those presented in Table I. This is due to the fact that the probability of the first source symbol shown in Table I of [11], [12] was incremented by 0.0000004 to make the total sum of the source probabilities equal to one. In order to compare our results with those given in [5], [7], and [9], we still keep the

TABLE III
THE MAXIMUM CODEWORD LENGTHS (MAX.) AND THE CODEWORD LENGTH VECTORS OF HUFFMAN CODE AND THE SYMMETRIC RVLCS CONSTRUCTED BY THE PROPOSED ALGORITHM AND THAT OF [7] FOR THE CANTERBURY CORPUS FILE SET.

File	Algorithm	Max.	Codeword length vector
asyoulik.txt	Huffman[24]	15	(0,0,1,6,8,7,11,8,5,11,5,1,0,3,2)
(68 codewords)	Tseng et al.[7]	12	(0,0,3,2,5,5,7,6,10,9,16,5)
	Ours	12	(0,0,3,2,5,5,7,6,10,9,16,5)
alice29.txt	Huffman[24]	16	(0,1,0,4,9,8,6,5,4,13,12,3,0,1,4,4)
(74 codewords)	Tseng et al.[7]	12	(0,0,3,2,5,5,7,6,10,9,16,11)
	Ours	12	(0,0,3,2,5,5,7,6,10,9,16,11)
xargs.1	Huffman[24]	12	(0,0,1,6,9,6,8,9,6,8,11,10)
(74 codewords)	Tseng et al.[7]	12	(0,0,3,2,5,5,7,6,10,9,16,11)
	Ours	12	(0,0,3,2,5,5,7,6,10,9,16,11)
grammar.lsp	Huffman[24]	12	(0,1,0,4,8,8,8,8,5,15,9,10)
(76 codewords)	Tseng et al.[7]	12	(0,1,1,2,5,5,6,7,9,10,15,15)
	Ours	12	(0,1,1,2,5,5,6,7,9,10,15,15)
plrabn12.txt	Huffman[24]	19	(0,0,2,7,3,9,4,4,7,14,5,2,6,1,1,7,4,3,2)
(81 codewords)	Tseng et al.[7]	14	(0,0,4,2,4,4,6,4,8,6,12,8,18,5)
	Ours	14	(0,0,4,2,4,4,6,4,8,6,12,8,18,5)
lcet10.txt	Huffman[24]	16	(0,0,1,8,4,9,5,7,12,13,10,6,2,3,0,4)
(84 codewords)	Tseng et al.[7]	13	(0,0,3,3,4,4,7,5,10,8,15,11,14)
	Ours	13	(0,0,3,3,4,4,7,5,10,8,15,11,14)
cp.html	Huffman[24]	14	(0,0,0,6,12,8,5,11,13,13,6,5,5,2)
(86 codewords)	Tseng et al.[7]	12	(0,0,2,2,6,6,8,8,12,12,20,10)
	Ours	12	(0,0,2,2,6,6,8,8,12,12,20,10)
fields.c	Huffman[24]	13	(0,1,0,3,7,11,7,17,16,17,3,4,4)
(90 codewords)	Tseng et al.[7]	12	(0,0,2,2,6,6,8,8,12,12,20,14)
	Ours	12	(0,0,2,2,6,6,8,8,12,12,20,14)
ptt5	Huffman[24]	17	(1,0,0,1,2,14,11,9,9,7,10,9,13,20,16,11,26)
(159 codewords)	Tseng et al.[7]	16	(1,0,0,2,3,3,5,5,8,8,13,12,22,23,41,13)
	Ours	16	(1,0,0,2,3,3,5,5,8,8,13,12,22,23,41,13)
sum	Huffman[24]	14	(0,1,0,0,3,20,16,24,25,45,30,56,29,6)
(255 codewords)	Tseng et al.[7]	15	(0,1,0,1,5,5,9,10,15,14,26,24,44,43,58)
	Ours	15	(0,1,0,1,5,5,9,10,15,14,26,24,44,43,58)
kennedy.xls	Huffman[24]	12	(1,0,1,3,1,0,0,1,1,74,146,28)
(256 codewords)	Tseng et al.[7]	17	(1,0,1,0,3,1,1,4,8,8,17,16,31,29,59,56,21)
	Ours	17	(1,0,1,2,1,1,1,3,7,7,13,13,23,22,43,41,77)

TABLE IV
THE MAXIMUM CODEWORD LENGTHS (MAX.) AND THE CODEWORD LENGTH VECTORS OF HUFFMAN CODE AND THE ASYMMETRIC RVLCS CONSTRUCTED BY THE PROPOSED ALGORITHM AND THAT OF [9] FOR THE CANTERBURY CORPUS FILE SET

File	Algorithm	Max.	Codeword length vector
asyoulik.txt	Huffman[24]	15	(0,0,1,6,8,7,11,8,5,11,5,1,0,3,2)
(68 codewords)	Tseng et al.[9]	11	(0,0,1,6,7,8,9,10,11,13,3)
	Ours	12	(0,0,1,5,9,8,12,8,9,8,3,5)
alice29.txt	Huffman[24]	16	(0,1,0,4,9,8,6,5,4,13,12,3,0,1,4,4)
(74 codewords)	Tseng et al.[9]	11	(0,0,1,6,7,8,9,10,11,12,10)
	Ours	12	(0,0,1,6,8,8,7,5,8,10,14,7)
xargs.1	Huffman[24]	12	(0,0,1,6,9,6,8,9,6,8,11,10)
(74 codewords)	Tseng et al.[9]	11	(0,0,1,6,7,8,9,10,12,10,11)
	Ours	12	(0,0,1,5,9,9,8,11,10,10,7,4)
grammar.lsp	Huffman[24]	12	(0,1,0,4,8,8,8,8,5,15,9,10)
(76 codewords)	Tseng et al.[9]	11	(0,0,1,6,7,8,9,10,11,12,12)
	Ours	12	(0,0,1,5,9,9,9,7,14,10,7,5)
plrabn12.txt	Huffman[24]	19	(0,0,2,7,3,9,4,4,7,14,5,2,6,1,1,7,4,3,2)
(81 codewords)	Tseng et al.[9]	12	(0,0,1,6,7,8,9,10,11,12,13,4)
	Ours	12	(0,0,1,7,7,7,4,5,7,10,15,18)
lcet10.txt	Huffman[24]	16	(0,0,1,8,4,9,5,7,12,13,10,6,2,3,0,4)
(84 codewords)	Tseng et al.[9]	12	(0,0,1,6,7,8,9,10,11,12,13,7)
	Ours	12	(0,0,1,7,5,8,7,10,16,16,5,9)
ptt5	Huffman[24]	17	(1,0,0,1,2,14,11,9,9,7,10,9,13,20,16,11,26)
(159 codewords)	Tseng et al.[9]	13	(1,0,1,0,3,3,11,9,16,15,24,35,41)
	Ours	13	(1,0,0,1,3,9,9,12,12,12,22,38,40)

TABLE V
THE EXECUTION TIME AND MEMORY REQUIREMENT OF OUR PROPOSED ALGORITHM WHILE CONSTRUCTING THE SYMMETRIC RVLCS FOR THE ENGLISH ALPHABET AND THE CANTERBURY CORPUS FILE SET.

		Execution Time			Number of rounds (Number of g functions executed) {Number of h functions executed}			Maximum stack size		
Source	Number of codewords	0 nodes	2 nodes	4 nodes	0 nodes	2 nodes	4 nodes	0 nodes	2 nodes	4 nodes
English alphabet	26	0.036	0.038	0.039	46 (88) {88}	26 (198) {151}	21 (249) {212}	16	5	3
asyoulik.txt	68	0.068	0.115	0.120	223 (441) {441}	141 (1169) {893}	98 (1312) {1121}	50	23	16
alice29.txt	74	0.070	0.127	0.136	205 (405) {405}	124 (1041) {798}	90 (1206) {1031}	47	21	18
xargs.1	74	0.081	0.119	0.132	218 (431) {431}	133 (1107) {847}	97 (1294) {1105}	51	21	16
grammar.lsp	76	0.072	0.123	0.127	228 (451) {451}	129 (1083) {830}	84 (1156) {993}	56	26	19
plrabn12.txt	81	0.082	0.128	0.128	144 (283) {283}	86 (713) {546}	59 (788) {675}	31	16	11
lcet10.txt	84	0.079	0.156	0.182	195 (385) {385}	121 (1003) {766}	95 (1236) {1051}	45	21	17
cp.html	86	0.073	0.128	0.139	253 (500) {500}	149 (1229) {938}	107 (1417) {1210}	67	26	21
fields.c	90	0.130	0.259	0.268	525 (1045) {1045}	288 (2458) {1889}	175 (2501) {2158}	132	75	42
ptt5	159	0.135	0.240	0.285	98 (194) {194}	63 (508) {386}	51 (638) {540}	26	10	9
sum	255	4.100	7.400	7.600	3195 (6384) {6384}	1693 (14682) {11303}	1075 (15541) {13398}	880	445	275
kennedy.xls	256	7.300	15.300	16.400	4881 (9759) {9759}	2543 (25402) {20321}	1520 (27308) {24272}	4854	2533	1512

original source probability in our experiments.

The number of codewords and the average codeword lengths of the symmetric/asymmetric RVLCs for the Canterbury Corpus file set found by the algorithm of Tseng *et al.* [7], [9] and our proposed algorithm are given in Table II. As given in Table II, all of the symmetric RVLCs constructed by our proposed algorithm are no worse than those constructed by the algorithm of [7]. Since the algorithm of [7] is suboptimal, it failed to find an optimal symmetric RVLC for "kennedy.xls." In Table II, we also add the results found in [5] since there are three better codes found by the algorithm of [5] than that by the algorithm of [9]. All asymmetric RVLCs constructed by our proposed algorithm are better than those constructed by the algorithms of [9] and [5]. Since the algorithms of [9] and [5] are suboptimal, they failed to find optimal asymmetric RVLCs for all of the test files. Unlike when constructing the symmetric RVLCs, our proposed algorithm has better performance in relation to lowering the average codeword length for asymmetric RVLCs. Since no results other than those for the English alphabet were given in [11] and [12], no comparison with [11] and [12] are presented here for source benchmarks taken from the Canterbury Corpus file.

Table III and Table IV respectively list the codeword length vectors and the maximum codeword lengths of the Huffman code, the symmetric/asymmetric RVLCs constructed by the proposed algorithm, and those of [7] and [9]. The value of the i-th entry of each codeword length vector given in this table denotes the number of codewords of length i. It can be observed that the symmetric RVLCs found by our proposed algorithm have the same codeword length vectors as those found in [7] except the last one for "kennedy.xls," where our algorithm found an optimal RVLC but the algorithm of [7] did not. In contrast, the asymmetric RVLCs found by our proposed algorithm have different codeword length vectors from those found in [9] since our proposed algorithm found optimal asymmetric RVLCs but that of [9] did not.

Table V and Table VI respectively list the execution time (in seconds), the number of rounds (and the number of g/h functions executed), and the maximum stack size (in number of nodes) required in our proposed algorithm while constructing symmetric and asymmetric RVLCs for the English alphabet and the Canterbury Corpus File Set. The values inside the parentheses and brackets shown in Table V and Table VI respectively denote the number of g functions executed and the number of h functions executed. The maximum temporal local stack size is equal to EN+1. The results were simulated

TABLE VI
THE EXECUTION TIME AND MEMORY REQUIREMENT OF OUR PROPOSED ALGORITHM WHILE CONSTRUCTING THE ASYMMETRIC RVLCS FOR THE ENGLISH ALPHABET AND THE CANTERBURY CORPUS FILE SET.

		Execution Time			Number of rounds (Number of g functions executed) {Number of h functions executed}			Maximum stack size		
Source	Number of codewords	50 nodes	300 nodes	500 nodes	50 nodes	300 nodes	500 nodes	50 nodes	300 nodes	500 nodes
English alphabet	26	1.3	1.4	1.3	537 (84935) {83868}	110 (99251) {99035}	62 (90542) {90423}	158	36	17
asyoulik.txt	68	9620	11094	11554	2531735 (410148601) {405085137}	449589 (426159962) {425260781}	271571 (427445134) {426901994}	722584	123564	74532
alice29.txt	74	3770	4194	4445	842429 (135985087) {134300239}	141866 (136860099) {136564069}	84695 (136436434) {136249582}	247498	43452	26118
xargs.1	74	–	15942	16841	– (–) {–}	525620 (504862186) {503810906}	318797 (506282063) {505644425}	–	172211	97293
grammar.lsp	76	–	22119	23968	– (–) {–}	699801 (659572509) {658172907}	424933 (661516574) {660666713}	–	217029	122187
plrabn12.txt	81	2143	2025	2773	322391 (61808597) {61163825}	54934 (54411908) {54302040}	33487 (56357032) {56290048}	200335	21934	14106
lcet10.txt	84	6641	6573	8165	1109105 (199637376) {197419174}	189078 (183668882) {183290726}	112323 (181573546) {181348902}	465093	62426	36607
ptt5	159	41	52	74	3780 (762914) {755358}	844 (1010178) {1008494}	604 (1203218) {1202014}	3775	839	601

when the number of expanded nodes in a subtree were set to 0, 2, and 4 and 50, 300, and 500 for symmetric RVLCs and asymmetric RVLCs, respectively. From the results given in Table V and Table VI, one can see that when the number of expanded nodes in a subtree is set to be larger, the number of rounds and the memory requirement decrease, which is due to better estimation (or a larger value of h). Since a larger number of expanded nodes in a subtree results in higher computation complexity when calculating values of h, the overall execution time increases, except for "alphabet," "plrabn12.txt," and "lcet10.txt." The fact that the overall execution times for 300 expanded nodes were slightly smaller than those for 50 expanded nodes for "plrabn12.txt" and "lcet10.txt" indicates that increasing the number of expanded nodes does not always increase the overall execution time. In Table VI, the results denoted by "$-$" are those that could not be obtained successfully due to a memory limitation. Our proposed algorithm was not able to find optimal asymmetric RVLCs for "cp.html," "fields.c," "sum," and "kennedy.xls" in a limited time. Without the "symmetry" constraint, the codeword selection can be more flexible in constructing an asymmetric RVLC. As a result, the cost values of the nodes in the corresponding search tree are closer to each other than those when constructing a symmetric RVLC. Hence, for a given source, it is harder to find an optimal asymmetric RVLC than an optimal symmetric RVLC because we have to search back and forth and the number of rounds performed by our proposed algorithm increases drastically when N becomes large. However, it is possible that the search will still be fast. For example, since the probability of the most occurring symbol in the "ptt5" file is relatively high (up to 0.87124914), the resultant search tree associated with our proposed algorithm will remain within a reasonable size. Hence, our algorithm can easily locate the path corresponding to the optimal code. It only needs 41 seconds to find the optimal asymmetric RVLC, whereas the algorithm proposed in [4] needs about 10^{18} years (This data was shown in Table III of [6]) just to find a sub-optimal asymmetric RVLC.

V. CONCLUSIONS AND FUTURE WORK

Based on the A^* algorithm concept, a unified approach for constructing symmetric RVLCs and asymmetric RVLCs with the minimum average codeword length was presented. Some of optimal asymmetric RVLCs for the Canterbury Corpus file set, which were not found in the past, were constructed systematically in a limited time. Even though our proposed algorithm is still exponential in the worst case, it successfully found optimal asymmetric RVLCs for all of the benchmarks except for four test files. A method to further improve the search speed of our proposed algorithm will certainly be investigated in our future work.

There are still many interesting research areas about RVLCs, such as the balance of $0/1$ bits [25], free distance [13], error detection/synchronization [26], and error resiliency measurement [27]. Therefore, instead of just considering coding efficiency, in the future, we expect to find more efficient RVLCs in regard to possessing the additional features of higher balance in 0/1 bits, larger free distance, or stronger synchronization capability.

ACKNOWLEDGMENT

The authors would like to express their special thanks to the associate editor Professor J. Kliewer and the three anonymous reviewers for their valuable comments.

REFERENCES

[1] Y. Takishima, M. Wade, and H. Murakami, "Reversible variable length codes," *IEEE Trans. Commun.*, vol. 43, no. 2/3/4, pp. 158-162, Feb. 1995.

[2] C.-W. Tsai and J.-L. Wu, "Modified symmetrical reversible variable-length code and its theoretical bounds," *IEEE Trans. Inf. Theory*, vol. 47, no. 6, pp. 2543-2548, Sep. 2001.

[3] ——, "On constructing the Huffman-code-based reversible variable-length codes," *IEEE Trans. Commun.*, vol. 49, no. 9, pp. 1506-1509, Sep. 2001.

[4] K. Lakovic and J. Villasenor, "An algorithm for construction of efficient fix-free codes," *IEEE Commun. Lett.*, vol. 7, no. 8, pp. 391-393, Aug. 2003.

[5] J. Wang, L.-L. Yang, and L. Hanzo, "Iterative construction of reversible variable-length codes and variable-length error-correcting codes," *IEEE Commun. Lett.*, vol. 8, no. 11, pp. 671-673, Nov. 2004.

[6] C.-W. Lin, J.-L. Wu, and Y.-J. Chuang, "Two algorithms for constructing efficient Huffman-code based reversible variable length codes," *IEEE Trans. Commun.*, vol. 56, no. 1, pp. 81-89, Jan. 2008.

[7] H.-W. Tseng and C.-C. Chang, "Construction of symmetrical reversible variable length codes using backtracking," *Computer J.*, vol. 46, no. 1, Jan. 2003.

[8] C.-W. Lin, Y.-J. Chuang, and J.-L. Wu, "Generic construction algorithm for symmetric and asymmetric RVLCs," in *Proc. IEEE International Conf. Commun. Syst.*, Singapore, Nov. 2002, pp. 968-972.

[9] H.-W. Tseng and C.-C. Chang, "A branch-and-bound algorithm for construction of reversible variable length codes," *Computer J.*, vol. 47, no. 6, Nov. 2004.

[10] W.-H. Jeong, Y.-S. Yoon, and Y.-S. Ho, "Design of reversible variable-length codes using properties of the Huffman code and average length function," in *Proc. IEEE International Conf. Image Process.*, Singapore, Oct. 2004, pp. 817-820.

[11] S. A. Savari, "On optimal reversible-variable-length codes," in *Proc. IEEE Int. Workshop Inf. Theory Appl.*, San Diego, CA, Feb. 2009, pp. 311-317.

[12] ——, "On minimum-redundancy fix-free codes," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 2009, pp. 3-12.

[13] K. Lakovic and J. Villasenor, "On design of error-correcting reversible variable length codes," *IEEE Commun. Lett.*, vol. 6, no. 8, pp. 337-339, Jan. 2002.

[14] V. Buttigieg and R. Deguara, "Using variable-length error-correcting codes in MPEG-4 video," in *Proc. Int. Symp. Inf. Theory*, Sep. 2005, pp. 2379-2383.

[15] Y.-M. Huang, Y. S. Han, and T.-Y. Wu, "Soft-decision priority-first decoding algorithms for variable-length error-correcting codes," *IEEE Commun. Lett.*, pp. 572-574, Aug. 2008.

[16] C. Guillemot and P. Christ, "Joint source-channel coding as an element of qos framework for 4G wireless multimedia," *Comput. Commun.*, vol. 27, pp. 762-779, 2004.

[17] R. Thobaben and J. Kliewer, "An efficient variable-length code construction for iterative source-channel decoding," *IEEE Trans. Commun.*, vol. 57, no. 7, pp. 2005-2013, July 2009.

[18] E. N. Gilbert, "Codes based on inaccurate source probabilities," *IEEE Trans. Inf. Theory*, vol. 17, pp. 304-314, May 1971.

[19] N. Abedini, S. P. Khatri, and S. A. Savari, "A SAT-based scheme to determine optimal fix-free codes," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 2010, pp. 169-178.

[20] G. K. Palshikar, "Satisfying the satisfiability problem," *Dr. Dobb's J.*, http://www.ddj.com/cpp/184402009, Sep. 1 2005.

[21] E. Rich and K. Knight, *Artificial Intelligence*. New York: McGraw-Hill, 1991.

[22] Y. S. Han, C. R. P. Hartmann, and C.-C. Chen, "Efficient priority-first search maximum-likelihood soft-decision decoding of linear block codes," *IEEE Trans. Inf. Theory*, vol. 39, no. 5, pp. 1514-1523, Sep. 1993.

[23] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley Publishing Company, 1984.

[24] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, pp. 1098-1101, Sep. 1952.

[25] J. Y. Lin, Y. Liu, and K. C. Yi, "Balance of 0,1 bits for Huffman and reversible variable-length coding," *IEEE Trans. Commun.*, vol. 52, no. 3, pp. 359-361, Mar. 2004.

[26] C.-W. Lin and J.-L. Wu, "On error detection and error synchronization of reversible variable-length codes," *IEEE Trans. Commun.*, vol. 53, no. 5, pp. 826-832, May. 2005.

[27] L. Xu and S. Kumar, "Error resiliency measure for RVLC codes," *IEEE Signal Process. Lett.*, vol. 13, no. 2, pp. 84-87, Feb. 2006.

Yuh-Ming Huang received the B.Sc. degree in mathematics from National Tsing Hua University, Hsinchu, Taiwan, in 1987 and the M.Sc. and Ph.D. degrees in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1989 and 1999, respectively.

During his military service (from 1989 to 1991), he was an information instructor of the Training and Command Department, Naval Fleet, Kaohsiung, Taiwan. From 1991 on, he worked as an associate engineer in the High-Speed Network Department of the Communication System Group, ITRI, Hsinchu, Taiwan. He worked as a lecturer at Yuan Ze Institute of Technology, Taoyuan, Taiwan. He worked as a research assistant at the Communications and Multimedia Lab. of the Computer Science and Information Engineering Department, National Taiwan University, Taipei, Taiwan. He worked as a network team leader at the computer center of National Chi Nan University, Nantou, Taiwan.

He is now an Assistant Professor with the department of computer science and information engineering, National Chi Nan University, Nantou, Taiwan. His current research interests include data compression, error correction coding, joint source/channel coding, video encryption, and key agreement in mobile wireless communication.

Ting-Yi Wu received the B.Sc. and M.Sc. degrees in Computer Science and Information Engineering from National Chi-Nan University, Nantou, Taiwan, in 2005 and 2007, respectively. From 2007 to 2009, he was a research assistant of the Graduate Institute of Communication Engineering, National Taipei University, Taipei, Taiwan.

He is currently pursuing the Ph.D. degree in Institute of Communications Engineering, National Chiao-Tung University, Hsinchu, Taiwan. His current research interests include error-control coding and information theory.

Yunghsiang S. Han was born in Taipei, Taiwan, on April 24, 1962. He received B.Sc. and M.Sc. degrees in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan, in 1984 and 1986, respectively, and a Ph.D. degree from the School of Computer and Information Science, Syracuse University, Syracuse, NY, in 1993.

He was from 1986 to 1988 a lecturer at Ming-Hsin Engineering College, Hsinchu, Taiwan. He was a teaching assistant from 1989 to 1992, and a research associate in the School of Computer and Information Science, Syracuse University from 1992 to 1993. He was, from 1993 to 1997, an Associate Professor in the Department of Electronic Engineering at Hua Fan College of Humanities and Technology, Taipei Hsien, Taiwan. He was with the Department of Computer Science and Information Engineering at National Chi Nan University, Nantou, Taiwan from 1997 to 2004. He was promoted to Professor in 1998. He was a visiting scholar in the Department of Electrical Engineering at University of Hawaii at Manoa, HI from June to October 2001, the SUPRIA visiting research scholar in the Department of Electrical Engineering and Computer Science and CASE center at Syracuse University, NY from September 2002 to January 2004, and the visiting scholar in the Department of Electrical and Computer Engineering at University of Texas at Austin, TX from August 2008 to June 2009. He was with the Graduate Institute of Communication Engineering at National Taipei University, Taipei, Taiwan from August 2004 to July 2010. From August 2010, he is with the Department of Electrical Engineering at National Taiwan University of Science and Technology. His research interests are in error-control coding, wireless networks, and security. Dr. Han was a winner of the 1994 Syracuse University Doctoral Prize.