Automatic Generation of C++/Java Code for Efficient VLC Decoding

Danny Hong, *Student Member, IEEE,* Alexandros Eleftheriadis, *Senior Member, IEEE*

Abstract

We present an efficient method for software decoding of variable-length codes used in multimedia compression standards such as MPEG and H.26x. We also define a new problem called "VLC partitioning", whose solution enables VLC decoding methods to perform optimally without going over a given memory upper limit. We have modified Flavor, a software tool that automatically generates C++ or Java VLC decoding code, so that the Flavor-generated VLC decoder uses our proposed method. We show that the generated VLC decoder is comparable to or faster than other known, manually designed software VLC decoders.

Index Terms

Variable length codes, software tools, algorithms, complexity theory

This material is based upon work supported in part by the National Science Foundation under Grant MIPS-9703163.

Automatic Generation of C++/Java Code for Efficient VLC Decoding

I. Introduction

VARIABLE-length codes (VLCs), particularly Huffman codes (or minimum redundancy codes) [1], are widely used in media representation due to their decoding speed, simplicity in realization and efficiency in compression. For example, they are used in numerous international standards including the Group 3 and 4 facsimile standards [2], [3], JPEG [4], MPEG [5], [6], [7], and so on. Accordingly, methods for efficient decoding of the coded data have been proposed for both hardware [8], [9], [10] and software [11], [12], [13], [14], [15], [16], [17]. We focus on software decoding of VLCs, and the main objective is to maximize decoding speed without consuming too much memory.

Among minimum redundancy codes, canonical codes (CCs) [18], [19] have gained a wide interest because they can be very efficiently processed [12], [14], [17]. However, the above-mentioned standards specify a set of fixed VLC tables[1], most of which are not canonical. This is especially true for reversible VLCs (RVLCs) [20] that have been utilized both in MPEG-4 and H.263+ for better error resiliency, and there is no known technique to design reversible canonical codes with the same performance as reversible only codes. Hence, it is highly desirable to be able to (automatically) generate an efficient VLC decoding method for *any given* VLC table: our interest is in general software decoding methods that can be used with any type of VLCs.

Traditional ways of decoding VLCs normally follow one of two methods, or a mix thereof. The first is to use a binary tree, which is traversed one bit at a time. This binary tree approach, although conceptually simple, is extremely slow, requiring at most L stages of bitstream input and lookup, where L is the length of the longest codeword. The second method, the fastest decoding method, is to use just one lookup table with 2^L entries. In this case, L bits are read in a look-ahead mode from the bitstream, and they are used as an index to the table. Each entry of the table contains two values: the value of the codeword length and the symbol value associated with the first codeword contained in the L bits. Once an entry corresponding to the L bits is found, the number of bits equal to the looked-up codeword length will be skipped from the bitstream.

It is possible that L bits contain more than one codeword and to increase the throughput, the values of the symbols corresponding to the L bits can be stored in the table. This leads to a state-based decoder [11], [21], [22], [23], and much more memory than the traditional lookup table-based approach of decoding one codeword at a time is required, as each state requires a different table. Another drawback of decoding multiple codewords at a time is that a typical multimedia bitstream, e.g., an MPEG-2 video stream, is comprised of a mixture of codewords from multiple VLC tables. Additionally, in the bitstream, whether and how to decode a codeword may depend on a previously decoded symbol value. Consequently, it may not be desirable to decode multiple codewords at a time, and we only concentrate on decoding methods that output one symbol at a time. Note that the throughput of such decoding methods that ouput one symbol at a time can be improved by applying parallel processing, and more information about parallel huffman coding can be found in [24], [25].

The drawback of the binary tree approach is the slow decoding speed, whereas the drawback of the approach using just one lookup table is the high memory requirement. A compromise between these two extremes is a combination of smaller lookup tables, organized in a hierarchical fashion. This combines the efficiency of multi-bit lookups and the lower memory requirements of a hierarchical lookup scheme.

We define a "VLC partition", or simply a "partition", as a set of one or more lookup tables used for VLC decoding. Then, we call the problem of coming up with a partition, in such a way that the memory requirement is under a given bound and the average decoding time is minimized, a "VLC partitioning" or "VP" problem. If the tables must be organized in a hierarchical fashion, then we call such special case of the problem a "hierarchical VP" or "HVP" problem. Additionally, we call a set of hierarchically organized lookup tables a "hierarchical VLC partition" or "hierarchical partition".

The next section describes some of the already-proposed software decoding methods for VLCs. All the methods can be divided into two parts. The first part provides a way to generate a partition from a given set of codewords, which allows fast decoding without wasting too much memory, and the second part provides a way to efficiently access the generated lookup tables. Then in Section III, we provide a novel VLC decoding method that has the following three advantages over the table-based methods: 1) it generally requires less memory, 2) it requires less average decoding time, 3) it is easier to automatically generate. The third advantage is very important because all the methods should be customized to each VLC table for optimal performance. Today, most of the decoding methods are manually customized, consuming much time and effort. Note that we are not trying to design VLCs such as CCs for efficient processing; rather, the VLCs are given.

Our new method allows flexible partitioning of given codewords, enabling a wide range of memory constraints to be satisfied. The method works optimally (under a given memory constraint), if a solution to the HVP problem is provided. In Section IV, we formally state the HVP problem and prove that it is NP-complete. Then we provide a simple greedy algorithm

[1] The JPEG standard supports custom tables for each image, but generally the typical tables specified in Annex K of the standard are used.

for finding a hierarchical partition that can be used to customize the proposed method for efficient VLC decoding. We have modified Flavor [26], [27], [28], [29] so that the proposed decoding method and the greedy algorithm are incorporated. Using Flavor, efficient VLC decoding code that is customized to each set of codewords can be automatically generated. In Section V we analyze the methods described in Section II as well as the one used by the Flavor-generated decoder. We also describe some of the popular methods [14], [17] used for decoding CCs, and show that the Flavor-generated decoder is comparable in terms of both space and time complexity to the special CC decoders. Section VI shows and discusses some experimental results of using the methods analyzed in Section V, and finally we conclude with Section VII, summarizing the work described in this paper.

II. Background

In this section, we introduce some of the already-proposed software VLC decoding methods that can be used with any type of VLCs.

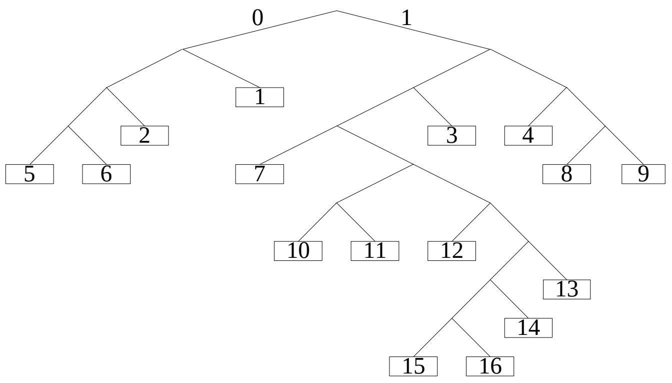

Fig. 1. A sample Huffman tree representing a set of variable-length codewords.

A. Hashemian Method

The Hashemian method [13] breaks a Huffman tree into smaller clusters (subtrees) by cutting the tree at every K^{th} level. Given a sample tree in Figure 1, if $K = 3$, then we get the lookup tables (LTs) and a super table (ST) as shown in Figure 2; each entry of the table is comprised of a codeword length (CL) and a symbol value (SV). For decoding, first we perform a K-bit (3-bit) lookup using LT1. If the value of the CL is 0, then no valid codeword has been detected and further bitstream lookup is required. For example, if the bits are “`000`”, then the CL is 0 and since the SV is 1, we use the first entry of the ST to determine the table for the next lookup. From the entry of the ST, we get the size of the next lookup (CL = 1) as well as the corresponding table (LT2). Similarly, if the bits are “`100`” or “`111`”, then LT3 or LT4 is used for the next lookup, respectively. Any other bits include a valid codeword and the corresponding CL and SV can be obtained from the table.

Generally, as the value of K increases, the decoding time decreases. However, the memory requirement increases with

LT1

Index	CL	SV
`000`	0	0
`001`	3	2
`010`	2	1
`011`	2	1
`100`	0	1
`101`	3	3
`110`	3	4
`111`	0	2

LT3

Index	CL	SV
`000`	1	7
`001`	1	7
`010`	1	7
`011`	1	7
`100`	3	10
`101`	3	11
`110`	3	12
`111`	0	3

LT5

Index	CL	SV
`000`	3	15
`001`	3	16
`010`	2	14
`011`	2	14
`100`	1	13
`101`	1	13
`110`	1	13
`111`	1	13

LT2

Index	CL	SV
`0`	1	5
`1`	1	6

LT4

Index	CL	SV
`0`	1	8
`1`	1	9

ST

Index	CL	SV
0	1	LT2
1	3	LT3
2	1	LT4
3	3	LT5

Fig. 2. Lookup tables derived from the Huffman tree shown in Figure 1 using the Hashemian method with $K = 3$.

K, and given a memory upper bound, the largest possible value of K can be selected for an optimal performance using this method.

B. Jiang Method

Jiang et al. [15] proposed an efficient decoding method based on pattern partition and LTs. The method is derived from the Hashemian method, where the tables are formed based on leading bit patterns. This method works well only for single-side growing Huffman trees (SGH-trees); however, we can have a VLC table that cannot be represented as a SGH-tree (e.g., Table B37 of the MPEG-4 Video specification), and for the codewords given in Figure 1, this method will produce a LT of size 2^7 for the bit pattern “`10`”. Note that half of the table entries will be used to represent the codeword “`101`”, resulting in significant memory waste.

This method generally requires less lookups than the Hashemian method as we only need to determine the leading bit pattern and then perform at most one lookup from the corresponding LT, regardless of the codeword length. However, this method generally results in higher memory requirement. Note that using this method, controlling the memory requirement will be extremely difficult, if not impossible. In addition, this method depends on a special function to recognize certain bit patterns from a bitstream, and such a function may not be available in general-purpose computers.

C. Aggarwal Method

The Aggarwal method [16] also uses leading bit patterns to partition a Huffman tree into small LTs. With this method, a Huffman tree can be partitioned both from the left and right branches, whereas in the Jiang method, only the right (or left) branch is considered for partitioning. Again, for the codewords given in Figure 1, a very large table is needed for the bit pattern “`10`”, and controlling the memory requirement will be difficult as well. This method generally requires less memory than the Jiang method, but it takes longer decoding time as it needs to know which side of the tree each bit pattern belongs to.

D. MPEG Method

Similar to the Jiang and Aggarwal methods, the VLC decoding method used in the MPEG-2/4 reference software uses only one lookup for each codeword. However, the LTs are created based on the integer value of the codewords. For example, given $L = 9$, the range of integer values of the codewords is [0, 2^9-1]. Then, one possible partition is a LT for the codewords with their integer value $\in$ [0, 2^4], another LT for the codewords with their integer value $\in$ [2^4+1, 2^6], and finally, one more LT for the codewords with their integer value $\in$ [2^6+1, 2^9-1]. Figure 3 shows the code segment used in the MPEG-2 reference software for decoding the AC coefficients of the intra-coded MPEG-2 block.

```
if (code >= 1024)
  tab = &DCTtab0a[(code>>8)-4];
else if (code >= 512)
  tab = &DCTtab1a[(code>>6)-8];
else if (code >= 256)
  tab = &DCTtab2a[(code>>4)-16];
else if (code >= 128)
  tab = &DCTtab3a[(code>>3)-16];
else if (code >= 64)
  tab = &DCTtab4a[(code>>2)-16];
else if (code >= 32)
  tab = &DCTtab5a[(code>>1)-16];
else if (code >= 16)
  tab = &DCTtab6a[code-16];
```

Fig. 3. A code segment used in the MPEG-2 reference software for decoding the codewords defined in Table B15 of the MPEG-2 Video specification. Here, the total number of table entries used is 340.

Using this approach the codewords given in Figure 1 would be first sorted in decreasing order of their integer value. Note that we use "left-justified" integer values for codewords whose length is less than 9. For example, since $L = 9$, the integer value for the codeword "`1111`" is 480, not 15. Once sorted, the codewords can be grouped into a set of LTs based on the memory constraint. For example, grouping the first four codewords {"`1111`", "`1110`", "`110`", "`101`"} together requires a LT of size 6 because of the following reason. After look-ahead reading 9 input bits, if the integer value of the 9-bits is greater than or equal to 320, then the table can be used with the index calculated as (x>>5)-10, where x is the integer value of the 9 bits. As a result, the range of the index becomes 0 to 5. Note that the Jiang and Aggarwal methods also group codewords according to their integer representation; but the MPEG method is more general, as it uses nested `if-else` statements to determine which table to use, rather than using a special function.

The MPEG method can also be interpreted as a generalized version of Algorithm ONE-SHIFT described in [14] for decoding CCs. Algorithm ONE-SHIFT uses integer values of codewords to group the codewords according to their lengths, and this is possible because a canonical tree, when scanning its leaves from left to right, they appear in non-decreasing (or non-increasing) order of their depth [17]. However, for general VLCs, such a property does not hold. As the length of each codeword cannot be derived from its integer value, each `if` or `else` condition does not have to correspond to a single codeword length, but a range of lengths. Additionally, general VLCs do not follow the numerical sequence property [14], and the codeword values cannot be derived via simple arithmetic.

There are two main disadvantages of using this method. First, as described above, customization is very cumbersome. Second, a longer codeword (representing a less probable symbol) can be decoded in less time than a shorter codeword (representing a more probable symbol). This is due to the fact that the codewords are not hierarchically partitioned. In the next section, we introduce a novel decoding method that uses hierarchical partition.

III. A NEW VLC DECODING METHOD

We present a novel method (Flavor5 method) for software VLC decoding, which is based on the hybrid method (we refer to it as the Flavor4 method) described in [28].

The Flavor4 method uses nested `switch` statements, instead of lookup tables (LTs). The benefit of using this method is that only complete matches require `case` statements, while all partial matches can be grouped into a single `default` statement (that, in turn, introduces another `switch` statement). The method is further optimized by ordering the `case` statements in terms of the length of their codewords. As shorter lengths correspond to higher probabilities, this minimizes the average number of comparisons per codeword.

The downside of the Flavor4 method is that if the comparing values of the `case` statements are scattered, the compiler (e.g., `gcc` or Visual C++) usually generates binary search code; however, if they are nearly sequential, a jump table (JT) is used. Binary search operations are generally much slower than jump operations, and additionally, even though the `case` statements are ordered in terms of their codeword length, the compiled machine code may not necessarily go through comparisons in that order. For example, given the `switch` statements in Figure 4, the left `switch` statement generates a machine code that checks `data` with the value 100 first, and if `data` is greater than 100, then it is checked with the value 150, otherwise it is checked with the value 50, and so on. On the other hand, for the right `switch` statement given in the figure, a JT is created, and for each `case` statement, there is at least one corresponding entry in the table.

```
switch (data) {      |    switch (data} {
case 0: ...          |    case 100: ...
case 50: ...         |    case 101: ...
case 100: ...        |    case 102: ...
case 150: ...        |    case 103: ...
case 200: ...        |    case 105: ...
}                    |    }
```

Fig. 4. The left `switch` statement behaves differently from the right one.

The Flavor5 method has been derived from the Flavor4 method so that it can use any given hierarchical partition, and for each LT in the partition, a corresponding `switch` statement is generated. This fixes the above-mentioned problem as the `case` values of each `switch` statement will be sequential.

Index	CL	SV1	SV2	SV3
000	1	1	22	100
001	1	1	22	100
010	1	1	22	100
011	1	1	22	100
100	2	2	37	101
101	2	2	37	101
110	3	3	57	122
111	3	4	63	139

$\xrightarrow{\text{Flavor5}}$

```
switch (Index) {
case 0: case 1:
case 2: case 3:
 SV1=1; SV2=22; SV3=100;
 CL=1; break;
case 4: case 5:
 SV1=2; SV2=37; SV3=101;
 CL=2; break;
case 6:
 SV1=3; SV2=57; SV3=122;
 CL=3; break;
case 7:
 SV1=4; SV2=63; SV3=139;
 CL=3; break;
}
```

$\xrightarrow{\text{Compiler}}$

```
 jmp *L0(Index)
L1:
 SV1=1; SV2=22; SV3=100;
 CL=1;
L5:
 SV1=2; SV2=37; SV3=101;
 CL=2;
L7:
 SV1=3; SV2=57; SV3=122;
 CL=3;
L8:
 SV1=4; SV2=63; SV3=139;
 CL=3;
```

Fig. 5. A lookup table, a corresponding `switch` statement, and a compiler-generated assembly code. `L0` is a jump table with 8 entries where the first four entries have the value `L1`; the next two entries, `L5`; and the last two entries, `L7` and `L8`, respectively.

In addition, using the `switch` statements generally requires less memory than using the LTs. Note that in the VLC tables specified by the standards like JPEG and MPEG, there are usually more than one symbol value corresponding to each codeword. For example, the quantized discrete cosine transform (DCT) coefficients [30] are first run-length coded before applying variable-length coding, and consequently, each codeword corresponds to a group of values (e.g., the run of zeroes, the level of the subsequent non-zero coefficient, as well as the last symbol indicator). Therefore, using an LT, each entry needs to have several values stored. Figure 5 shows an LT with four values (CL, SV1, SV2 and SV3), its corresponding `switch` statement and the compiler-generated assembly code. As can be seen from the figure, using the `switch` statement generates a JT, where each entry only contains one value corresponding to a destination address. `L0` is the JT created where its first four entries have the value `L1`; the next two, `L5`; and the last two, `L7` and `L8`, respectively

In addition to the less memory requirement, there are three major benefits of using `switch` statements. First, there is no need to keep track of the LTs. For example, in the Hashemian method, there was the need for one extra table (the ST) to keep track of the order of the LTs. Using our proposed method, the order of the `switch` statements are explicitly coded. Thus the extra table is not needed, and the extra step required to lookup the extra table is avoided. Second, by explicitely assigning symbol values as part of the code, rather than looking up the values from the table, it is less likely to get a cache miss. Third, our new method is very easy to implement for automatic code generation. Given any set of codewords and a hierarchical partition, we can directly create a set of `switch` statements in the given hierarchical order.

Note that the proposed method does not assume availability of special functions such as the ones used by the Jiang and Aggarwal methods for detecting certain bit patterns, and it adjusts well to various memory requirements. It also works well with all sorts of instantaneously decodable VLCs, not just canonical codes. Finally, by using a hierarchical partition, it makes sure that longer codewords are never decoded faster than shorter ones. Next we formally state the HVP problem and describe a simple way to obtain a hierarchical partition, which can be used to customize the proposed method to any given set of codewords.

IV. Hierarchical VLC Partitioning

As described in the previous two sections, most of the fast software VLC decoding methods exploit LTs or JTs. For such methods, a customized partition must be derived for each set of codewords. Note that by definition, the methods will perform optimally for the given set of codewords, under a given memory constraint, if the partition is an optimal partition. An optimal partition is formally defined in the following subsection. Unlike Jiang and Aggarwal methods, the Hashemian, MPEG and our proposed methods can work with different partitions to accommodate different memory requirements. To optimize such methods, the VP problem can be solved for each set of codewords.

Manually solving the VP problem is tedious and time-consuming and in this section, we propose a way to automate such process. Particularly, we focus on solving the HVP problem, as the proposed method works with hierarchical partitions. First, we formally state the HVP problem and prove that it is NP-complete, suggesting that finding an efficient algorithm to solve the problem would be extremely difficult (if not impossible). Then we provide a simple greedy algorithm for finding a hierarchical partition. Hereinafter, we simply refer to a hierarchical partition as a partition.

A. The HVP Problem

Given: Set $\{c_i\}$ of codewords where $i = 1, \ldots, N$, set $\{p_i\}$ of probabilities where p_i is the probability of the codeword c_i, set $\{l_i\}$ of codeword lengths where l_i is the length of the codeword c_i, and the memory upper bound U.

Definitions: Let F be the average number of table lookups per codeword during decoding. Let P be a partition of the codewords: P defines a set of LTs, organized in a hierarchical fashion. Let P be comprised of $n \in \mathcal{Z}^+$ LTs, and let $\{t_i\}$ denote the set of tables in the partition, where $i = 1, 2, \ldots, n$. Let f_i denote the number of tables looked-up to decode the codewords in t_i. We say a codeword is *in* a table t_i, if t_i is the last table needed to decode the codeword. Let S_i be the sum of the probabilities of the codewords in t_i.

Problem: We want to find a partition of the codewords that minimizes F where,

$$F = f_1 S_1 + f_2 S_2 + \cdots + f_n S_n, \text{ and} \tag{1}$$

$$\texttt{size}(P) = \sum_{i=1}^{n} \texttt{size}(t_i) \le U. \tag{2}$$

For example, consider the partition given in Section II-A and assume that the probabilities of the symbols 1, 2, ..., 16 are 0.3, 0.1, 0.1, 0.1, 0.05, 0.05, 0.05, 0.05, 0.05, 0.03, 0.03, 0.03, 0.02, 0.02, 0.01, 0.01, respectively. In this case we have $n = 5$, $f_1 = 1$, $f_2 = 2$, $f_3 = 2$, $f_4 = 2$, $f_5 = 3$, $S_1 = 0.6$, $S_2 = 0.1$, $S_3 = 0.14$, $S_4 = 0.1$, and $S_5 = 0.06$. Therefore, we have $F = 1(0.6)+2(0.1)+2(0.14)+2(0.1)+3(0.06) = 1.46$. Note that for our proposed method, we do not need to consider the ST.

An optimal partition is a partition that minimizes Equation (1) while satisfying Equation (2). The function $\texttt{size}(t_i)$ in Equation (2) returns the number of entries in t_i, and for the partition discussed in the above paragraph, the size is $8 + 2 + 8 + 2 + 8 = 28$. We want to find a partition that minimizes the average number of lookups without going over the given memory limit, and in the next subsection, we show that the HVP problem is NP-complete.

B. The HVP Problem is NP-complete

We prove that the HVP problem is NP-complete by restricting the problem to the Capacity Assigment (CA) problem [31], [32], which is a known NP-complete problem. First, we state the HVP problem (an optimization problem) as a decision problem: if an upper bound for the expected lookup delay is given (F_{max}), is there a partition with which the expected lookup delay is no larger than F_{max}? We can easily show that the HVP decision problem is an NP problem, as follows. Given a partition, we can derive the expected lookup delay, in polynomial time, by using Equation (1). Note that S_i is just the sum of the probabilities of the codewords in t_i, and f_i can easily be obtained by counting the total number of tables looked-up to decode the codeword in t_i. Then we can compare the calculated, expected lookup delay with F_{max}. We can also easily check to see if the total memory used by the given partition is less than U, in polynomial time. For this, we can simply add the size of each LT defined by the partition. Now, we can complete the proof that the HVP problem is NP-complete by showing that it is NP-hard.

We can restrict the HVP problem by only allowing the partitions formed by horizontally cutting across the whole Huffman tree. For example, say a tree is first cut at level k_1, then we are left with at most 2^{k_1} subtrees. Now the next cut, at level $k_1 + k_2$, will be across all the subtrees as shown in Figure 6(a). This kind of partitioning is very similar to the partitioning used by the Hashemian method, but in that case, $k_1 = k_2 = \cdots = k_m$. Basically, we are restricting the HVP problem by disallowing different cuts for different subtrees. Hence, the cuts shown in Figure 6(b) is not allowed in the restricted HVP (RHVP) problem, whereas it is allowed in the HVP problem. With this restriction, we get the following:

$$F = \hat{S}_1 + 2\hat{S}_2 + \cdots + m\hat{S}_m, \text{ where}$$

$$\hat{S}_j = \sum_{\mathcal{K}_{j-1} < l_i \le \mathcal{K}_j} p_i,\ j \in [1, m], \text{ and } \ \mathcal{K}_j = \sum_{i=0}^{j} k_i,\ k_0 = 0.$$

Note that with the RHVP problem, we have $k_1 + k_2 + \cdots + k_m = L$ and $m \in [1, L]$. This means $k_1 \in [1, L]$, $k_2 \in [1, L-k_1]$, $k_3 \in [1, L-\mathcal{K}_2]$, $k_4 \in [1, L-\mathcal{K}_3]$, and so on. For an arbitrary value $m \in [1, L]$, where $L \in \mathcal{Z}^+$, we can form two mxL matrices $\mathcal{F}$ and $\mathcal{U}$ with the rows corresponding to the cuts (k_i's) and the columns corresponding to the possible set of values the cuts can have. Then, the elements $\mathcal{F}_{ab}$ of the matrix $\mathcal{F}$ represent the additional expected number of table lookups resulting from the cut $k_a = b$, where $a \in [1, m]$ and $b \in [1, L]$. Similarly, the elements $\mathcal{U}_{ab}$ of the matrix $\mathcal{U}$ represent the accrued memory requirement due to the cut $k_a = b$.

The value of the elements of the matrices $\mathcal{F}$ and $\mathcal{U}$ can be easily obtained from the given set of codewords $\{c_i\}$, the corresponding probabilities $\{p_i\}$ and the codeword lengths $\{l_i\}$. More specifically, we get the following expressions:

$$\mathcal{F}_{1b} = 1 + \sum_{l_i > b} p_i \tag{3}$$

$$\mathcal{F}_{ab} = \sum_{l_i > \mathcal{K}_{a-1}+b} p_i,\ a > 1 \tag{4}$$

$$\mathcal{U}_{ab} = \sum_{t \in T_b} 2^{\min(b, h(t))},$$

where T_b is the set of subtrees resulting from the cut $k_a = b$, and $h(t)$ is the height of the tree t. Equation (3) is derived from the fact that by having $k_1 = b$, all the codewords with length greater than b require more than one lookup for decoding. Equation (4) is also derived from the same idea. Note that the values of the elements in the row a of the matrices $\mathcal{F}$ and $\mathcal{U}$ depend on the values of previous cuts k_i, where $i < a$. We can further simplify the RHVP problem so that the values of the elements of the matrices are fixed. So we have $\mathcal{U}_{ab} \in \mathcal{Z}^+$, and we can also make $\mathcal{F}_{ab}$ to take positive integer values by converting the probabilities $\{p_i\}$ into frequencies. Finally, we can restate our simplified RHVP problem as follows.

Given: A set of cuts $\{k_1, k_2, \ldots, k_m\}$, a set of corresponding values $\{1, 2, \ldots, L\}$, memory cost function $\mathcal{U}$: mx$L \to \mathcal{Z}^+$, lookup delay function $\mathcal{F}$: mx$L \to \mathcal{Z}^+$ such that, for all $a \in [1, m]$ and $i < j \in [1, L]$, $\mathcal{U}_{ai} \le \mathcal{U}_{aj}$ and $\mathcal{F}_{ai} \ge \mathcal{F}_{aj}$, and positive integers K and J.

Problem: Is there an assignment σ: $k_a \to \{1, 2, \ldots, L\}$ such that the total memory cost $\sum_{k_a} \mathcal{U}_{a\sigma(k_a)}$ does not exceed K and such that the total lookup delay penalty $\sum_{k_a} \mathcal{F}_{a\sigma(k_a)}$ does not exceed J?

As can be compared with the CA problem stated in [32], there is an obvious 1-to-1 correspondence between the instances of the simplified RHVP problem and those of the CA problem. Therefore, the RHVP problem is NP-hard and the HVP problem is NP-complete.

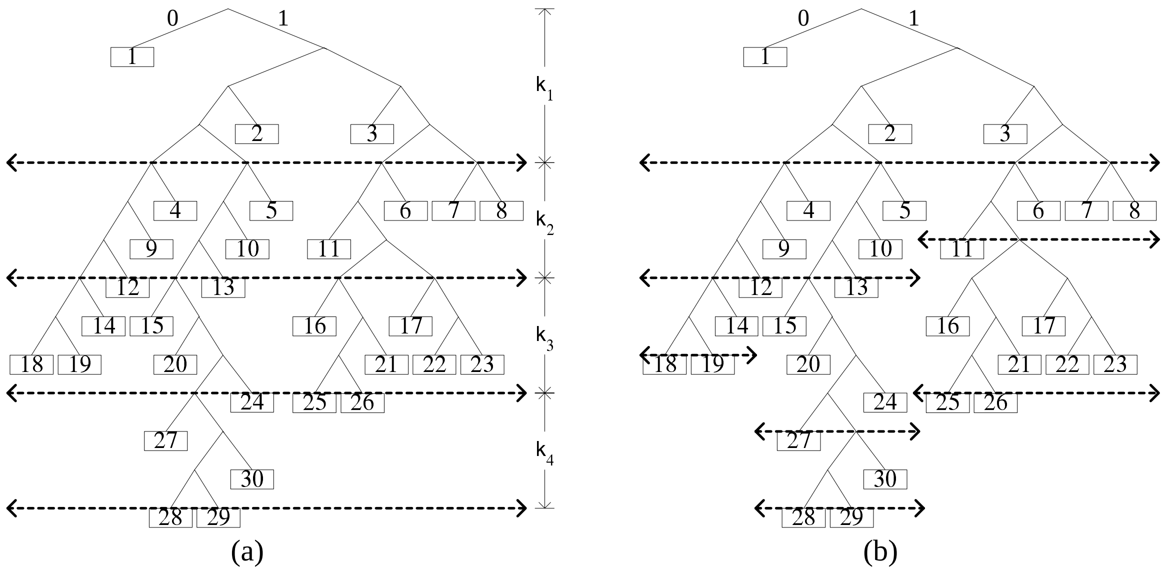

Fig. 6. Partition. (a) A partition obtained from the restricted HVP problem. (b) A partition obtained from the HVP problem.

C. Complexity of the HVP and RHVP Problems

Given a Huffman tree with height L, the maximum number of possible partitions $\pi(L)$ is (for the HVP problem and $L > 2$):

$$\pi(L) = \pi^{(2^1)}(L-1) + \pi^{(2^2)}(L-2) + \cdots + \pi^{(2^{(n-2)})}(2) + 2, \text{ and}$$

$$\pi(L) > \begin{cases} 2^{(2^L-1)/3} & , \quad L \geq 2,\ L \text{ even} \\ 2^{(2^L-2)/3} & , \quad L \geq 3,\ L \text{ odd.} \end{cases}$$

This means, even for L as small as 9, we can have more than 2^{170} partitions to test! Additionally, finding all the partitions is not a trivial task.

Even for the RHVP problem, the complexity increases exponentially with the increasing value of L, and in this case, the computational complexity is exactly 2^{L-1}. In the next subsection, we give a very simple greedy algorithm for finding a partition that can be used to customize our method to a given set of codewords.

D. A Greedy Algorithm for Finding a Good Partition

Let $\underline{\mathbf{B}}$ be an NxL matrix containing the bits of the given set of N codewords. The first row of $\underline{\mathbf{B}}$ contains the bits of the codeword with the largest integer value, the second row contains the bits of the codeword with the second largest integer value, and so on. Let $\underline{\mathbf{l}}$ be an Nx1 vector containing the lengths of the codewords. The first element of $\underline{\mathbf{l}}$ contains the length of the codeword in the first row of $\underline{\mathbf{B}}$, the second element of $\underline{\mathbf{l}}$ contains the length of the codeword in the second row of $\underline{\mathbf{B}}$, and so on. Then, a partition is formed by cutting the $\underline{\mathbf{B}}$ matrix at arbitrary columns. Generally, a partition made with less cuts yields better performance than the one with more cuts. So, we start with $m = 1$ and $k_1 = L$, and keep decreasing the value of k_1 until the memory requirement is satisfied. Then, we set $k_2 = L - k_1$ and keep decreasing the value of k_2 until the memory requirement is satisfied. We repeat this until $k_1 + k_2 + \cdots + k_m = L$.

There is a key benefit to using this greedy algorithm to obtain a partition; this algorithm avoids the need for the probability distribution of the codewords $\{pi\}$, which is required for calculating the average number of table lookups F. This is very useful when deriving a partition for VLCs specified in standards like JPEG and MPEG, where the the VLC tables are fixed and no distribution information is provided.

We have modified Flavor [26], [27], [28], [29] so that it incorporates both the new VLC decoding method (described in Section III) as well as the algorithm described in this section; hereinafter, we simply refer to the new decoding method (the Flavor5 method) as the Flavor method. In the following two sections, we show that the Flavor-generated VLC decoder yields better performance than the other ones described in this paper.

V. Performance Analysis

We analyze the performance of three VLC decoding methods (the Hashemian, MPEG and Flavor methods) discussed in this paper. Since both the Jiang and Aggarwal methods are not flexible to accommodate different memory constraints, we exclude them from our analysis. Additionally, their dependency on special functions to detect certain bit patterns makes them less favorable methods to be used in general-purpose computers. We also analyze some of the popular methods for decoding CCs [14], [17], and compare the Flavor method with the fastest CC decoding method – the TL method (the method that uses Algorithm TABLE-LOOKUP described in [14]). Even for such special type of VLCs, the Flavor method requires comparable decoding time as the methods described in [14], [17].

A. Analysis of General VLC Decoding Methods

To measure the decoding time, we use the number of table lookups and branching operations (or jumps). For example, using the tables defined in Figure 2, the Hashemian method requires 3 lookups and 2 jumps to decode the codeword `0b100100`. 3 lookups are needed for the tables LT1, ST, and LT3, in that order, and 2 jumps are needed for the branching operations after checking the CL of looked-up entry from LT1 and after obtaining the next-level LT (LT3) from the ST.

For the first analysis, each method is customized to the codewords defined in Table B15 of the MPEG-2 Video specification, with the memory upper bound U set to a fixed value of 340 entries (340 is obtained from the MPEG-2 reference software). We can safely assume that each entry of the tables used in all the methods occupies the same amount of memory space. Note that the Flavor method will require less memory space as described in Section III, but we will show that it still outperforms the other two methods.

Using the Hashemian method (for $U = 340$) we get 22 LTs and 1 ST, with $K = 7$; the code for using the MPEG method is shown in Figure 3; and we get $k_1 = 7$, $k_2 = 7$ and $k_3 = 2$ for the Flavor method. Table I(a) lists the decoding times needed for the given set of codewords. The first and second columns list the codeword length and the corresponding number of codewords, respectively. Looking at the 8^{th} entry of the table, we see that there are 3 codewords with length 9 bits, and it takes 3 lookups (3L) and 2 jumps (2J) to decode such codewords using the Hashemian method. Using the MPEG method, it takes 1L and 2J, and using the Flavor method, it takes 2L and 2J.

Clearly, the Flavor method is faster than the Hashemian method. Comparing with the MPEG method, the Flavor method yields the same performance when decoding codewords with length less than 8 bits. The MPEG method is definitely faster when decoding 8-bit long codewords. The MPEG method is also faster when decoding 9 or 10-bit codewords, but for the other codewords, the Flavor method is faster. Note that lookup operations are usually much faster than jump operations.

If we restrict U to 170 entries, then the average decoding time increases, and Table I(b) lists the new decoding times. For the Hashemian method, K is decreased to 3; for the MPEG method, smaller LTs are used; and for the Flavor method, we get $k_1 = 4$, $k_2 = 3$, $k_3 = 3$, $k_4 = 3$ and $k_5 = 3$. Again in this case, the Flavor method is always faster than the Hashemian method, and it is faster than the MPEG method for most of the codewords. Note that there are two entries for the codewords with length equal to 5 bits; there are 5 codewords with that length and two of them have integer values corresponding to one range and the other three have values corresponding to another range.

Analyzing the three methods with the codewords defined in Table B16 of the MPEG-4 Video specification also yields similar results. In Section VI, we experimentally show that the Flavor method indeed requires less average decoding time compared to the other methods.

B. Analysis of CC Decoding Methods

The basic CC decoding method (e.g., Algorithm CANONICAL-DECODE in [14]) requires bit-by-bit processing. This can be improved by using a special data structure called skeleton-tree (sk-tree) [17], where the bit-by-bit processing can be halted as soon as the processed bits represent the prefix of the codewords with the same length. The method using an sk-tree is similar to the Algorithm ONE-SHIFT described in [14], and for both cases, one or more lookups and zero or more jumps must be performed in order to obtain the length of the codeword to be decoded. Once the length is known, the codeword can be decoded with one lookup (e.g., the `diff` table used in [17]), and the decoded value can be used as an index into another table that maps the value into the actual symbol value. Therefore, a codeword is decoded with at least 3L; only a few codewords can be decoded this fast and other codewords require more lookups and jumps.

A faster CC decoding method is the TL method that uses an LT with 2^x entries so that codewords with length less than or equal to x bits can be decoded with only 3L. We compare this method with the Flavor method, and for this, we modified the codewords (by deleting the codewords with length 6 or 7 bits) in Table B5 of the MPEG-1 Video specification to make them canonical. Then using $x = 8$ for the LT, the TL method requires 256 entries for the LT, 16 entries for the `diff` table and 104 entries for the table listing the actual symbol values. Setting $U = 376$, we get $k_1 = 8$, $k_2 = 5$ and $k_3 = 3$ for the Flavor method, and *at most* only 3L and 3J are required.

VI. Experimental Results

To test the actual decoding time of the methods described in the previous section, three files are generated: `Uniform`, `Linear` and `Exponential`. The `Uniform` file contains 100,000, the `Linear` file contains 10,000 * $(16 - l_i + 1)$, and the `Exponential` file contains 100 * (2^{16-l_i}) of each codeword c_i. By using the `Uniform` file, we assume the worst case scenario where all codewords are equally-likely to occur. The `Linear` and `Exponential` files represent more practical scenarios where shorter codewords are more likely to occur than longer ones.

We first ran two experiments[2], testing the general VLC decoding methods. Table II lists the results, where the first three rows correspond to the decoding times of the codewords specified in Table B15 of the MPEG-2 Video specification with the `Uniform`, `Linear` and `Exponential` files containing 11,200,000, 6,190,000 and 6,129,600 codewords, respectively. The next three rows correspond to the decoding times of the codewords specified in Table B16 of the MPEG-4 Video specification, where the `Uniform`, `Linear` and `Exponential` files contain 10,300,000, 8,390,000 and 6,540,800 codewords, respectively.

From the first six rows of Table II, we see that the Flavor method outperforms the Hashemian method, and it also outperforms the MPEG method most of the time. For example, looking at the fifth row of the table, we see that the Flavor method takes 9.83% less time than the Hashemian method

[2] The testing was performed on a Intel Pentium M 1.6 GHz machine with 512 MB RAM. The `gcc` compiler with the `-O3` option is used to generate the decoders used to obtain the data in Table II

TABLE I
PERFORMANCE ANALYSIS: DECODING TIMES CORRESPONDING TO THE CODEWORDS DEFINED IN TABLE B15 OF THE MPEG-2 VIDEO SPECIFICATION, WITH (A) $U = 340$ AND (B) $U = 170$.

Len	#	Hashemian	MPEG	Flavor
2	1	1L 1J	1L 1J	1L 1J
3	2	1L 1J	1L 1J	1L 1J
4	1	1L 1J	1L 1J	1L 1J
5	5	1L 1J	1L 1J	1L 1J
6	5	1L 1J	1L 1J	1L 1J
7	9	1L 1J	1L 1J	1L 1J
8	14	3L 2J	1L 1J	2L 2J
9	3	3L 2J	1L 2J	2L 2J
10	2	3L 2J	1L 2J	2L 2J
12	10	3L 2J	1L 3J	2L 2J
13	12	3L 2J	1L 4J	2L 2J
14	16	3L 2J	1L 5J	2L 2J
15	16	5L 3J	1L 6J	3L 3J
16	16	5L 3J	1L 7J	3L 3J

(a)

Len	#	Hashemian	MPEG	Flavor
2	1	1L 1J	1L 2J	1L 1J
3	2	1L 1J	1L 2J	1L 1J
4	1	3L 2J	1L 2J	1L 1J
5	2	3L 2J	1L 1J	2L 2J
5	3	3L 2J	1L 2J	2L 2J
6	5	3L 2J	1L 4J	2L 2J
7	5	5L 3J	1L 1J	2L 2J
7	4	5L 3J	1L 4J	2L 2J
8	6	5L 3J	1L 1J	3L 3J
8	8	5L 3J	1L 3J	3L 3J
9	3	5L 3J	1L 5J	3L 3J
10	2	7L 4J	1L 5J	3L 3J
12	10	7L 4J	1L 6J	4L 4J
13	12	9L 5J	1L 7J	4L 4J
14	16	9L 5J	1L 8J	5L 5J
15	16	9L 5J	1L 9J	5L 5J
16	16	11L 6J	1L 10J	5L 5J

(b)

TABLE II
THE FIRST THREE ROWS CORRESPOND TO THE DECODING TIMES (IN MS) OF THE CODEWORDS DEFINED IN TABLE B15 OF THE MPEG-2 VIDEO SPECIFICATION; THE NEXT THREE CORRESPOND TO THOSE OF THE CODEWORDS DEFINED IN TABLE B16 OF THE MPEG-4 VIDEO SPECIFICATION; THE LAST THREE CORRESPOND TO THOSE OF THE CODEWORDS (EXCEPT THE ONES WITH LENGTH EQUAL TO 6 OR 7 BITS) DEFINED IN TABLE B5 OF THE MPEG-1 VIDEO SPECIFICATION. THE FIRST FOUR COLUMNS LIST THE DECODING TIMES OF THE RESPECTIVE METHODS, AND THE LAST THREE COLUMNS INDICATE THE PROCESSING GAIN OBTAINED USING THE FLAVOR METHOD COMPARED TO THE HASHEMIAN, MPEG, AND TL METHODS, RESPECTIVELY.

		Hashemian	MPEG	TL (x=8)	Flavor	Gain (H)	Gain (M)	Gain (T)
MPEG-2 Table B15	Uniform	961	861		871	10.33%	-1.15%	
	Linear	510	450		451	13.08%	-0.22%	
	Exponential	460	430		420	9.52%	2.38%	
MPEG-4 Table B16	Uniform	821	1021		771	6.49%	32.43%	
	Linear	670	861		610	9.83%	41.15%	
	Exponential	500	670		450	11.11%	48.89%	
MPEG-1 Table B5 (Modified)	Uniform	911	800	800	821	10.96%	-2.56%	-2.56%
	Linear	400	350	350	350	14.29%	0%	0%
	Exponential	420	400	390	400	5%	0%	-2.5%

when decoding the codewords in the `Linear` file. Also, as the frequency of shorter codewords gets bigger and that of the longer codewords smaller, which is the case for most of the practically coded sources, we see greater difference between the decoding times of the MPEG method and the Flavor method. This is due to the fact that the Flavor method uses hierarchical partition to ensure that the decoding time of shorter codewords is never greater than that of the longer ones; however, this is not the case with the MPEG method. Note that, from the first two rows of the table, we see that the Flavor method is slightly slower than the MPEG method, and the processing gain shown in the third row is not so significant. Nevertheless, the gain is obtained at zero cost, as the decoding code is automatically generated.

We also ran an experiment to compare the general VLC decoding methods with the TL method used for decoding CCs. For this, we used the codewords described in Section V-B, and the results are listed in the last three rows of Table II. Note that the Flavor method yields comparable decoding time as the TL method even though the the TL method has been designed specifically for CCs, whereas the Flavor method is not.

Note that, as described in the previous section, the methods tested in this section are customized so that they all utilize similar amount of memory.

VII. CONCLUSION

A novel and general method for software-based VLC decoding is proposed. We also introduced the VLC partitioning (VP) problem, whose solution can be used to optimize a given VLC decoding method. Our proposed method can be optimized to each set of codewords by solving the hierarchical VP (HVP) problem, which is shown to be NP-complete with the complexity of $\Omega(2^{(2^L/3)})$, where L is the length of the longest codeword. We presented a simple greedy algorithm for finding a hierarchical partition, and it can be used to customize the decoding method to any set of codewords.

We have enhanced Flavor to generate C++/Java VLC decoding code that utilizes the novel decoding method. It also uses the presented greedy algorithm to obtain a hierarchical partition, which is used to customize the decoding method to any given set of codewords. We have tested the new Flavor-generated decoder with other known decoders and have shown that it yields better or comparable performance, with about the same memory.

References

[1] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[2] *Standardization of Group 3 Facsimile Apparatus for Document Transmission*, CCITT (ITU Recommendation T.4), 1980, amended in 1984 and 1988.

[3] *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus*, CCITT (ITU Recommendation T.11), 1984, amended in 1988.

[4] *Information technology – Digital compression and coding of continuous-tone still images*, ISO/IEC 10918 International Standard (JPEG), 1994.

[5] *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbits/s*, ISO/IEC 11172 International Standard (MPEG-1), 1993.

[6] *Information technology – Generic coding of moving pictures and associated audio information*, ISO/IEC 13818 International Standard (MPEG-2), 1996.

[7] *Information technology – Coding of audio-visual objects*, ISO/IEC 14496 International Standard (MPEG-4), 1999.

[8] C. Fogg, "Survey of Software and Hardware VLC Architectures," in *SPIE Image and Video Compression*, 1994, pp. 29–37.

[9] M. K. Rudberg and L. Wanhammar, "High Speed Pipelined Parallel Huffman Decoding," in *IEEE Int. Symp. on Circuits and Systems*, 1997, pp. 2080–2083.

[10] R. A. Freking and K. K. Parhi, "An Unrestrictedly Parallel Scheme for Ultra-high-rate Reprogrammable Huffman Coding," in *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 1999, pp. 1937–1940.

[11] Y. Choueka, S. T. Klein, and Y. Perl, "Efficient Variants of Huffman Codes in High Level Languages," in *Int. ACM SIGIR Conf. on Research and Develp. in Info. Retrieval*, 1985, pp. 122–130.

[12] D. S. Hirschberg and D. A. Lelewer, "Efficient Decoding of Prefix Codes," *Communications of the ACM*, vol. 33, no. 4, pp. 449–459, 1990.

[13] R. Hashemian, "Memory Efficient and High-Speed Search Huffman Coding," *IEEE Trans. on Communications*, vol. 43, no. 10, pp. 2576–2581, 1995.

[14] A. Moffat and A. Turpin, "On the Implementation of Minimum Redundancy Prefix Codes," *IEEE Trans. on Communications*, vol. 45, no. 10, pp. 1200–1207, 1997.

[15] J. Jiang, C. Chang, and T. Chen, "An Efficient Huffman Decoding Method Based on Pattern Partition and Look-up Table," in *Asia-Pacific Conf. and Optoelectronics and Communications Conf. on Communications*, 1999, pp. 904–907.

[16] M. Aggarwal and A. Narayan, "Efficient Huffman Decoding," in *IEEE ICIP*, 2000, pp. 936–939.

[17] S. T. Klein, "Skeleton Trees for the Efficient Decoding of Huffman Encoded Texts," *Information Retrieval*, vol. 3, no. 1, pp. 7–23, 2000.

[18] E. S. Schwartz and B. Kallick, "Generating a Canonical Prefix Encoding," *Communications of the ACM*, vol. 7, no. 3, pp. 166–169, 1964.

[19] J. B. Connell, "A Huffman-Shannon-Fano Code," *Proceedings of the IEEE*, vol. 61, no. 7, pp. 1046–1047, 1973.

[20] Y. Takishima, M. Wada, and H. Murakami, "Reversible Variable Length Codes," *IEEE Trans. on Communications*, vol. 43, no. 2/3/4, pp. 158–162, 1995.

[21] H. Tanaka, "Data Structure of Huffman Codes and Its Applications to Efficient Encoding and Decoding," *IEEE Trans. on Communications*, vol. 27, no. 6, pp. 930–932, 1979.

[22] A. Sieminski, "Fast Decoding of the Huffman Codes," in *Information Processing Letters*, 1988, pp. 237–241.

[23] V. Iyengar and K. Chakrabarty, "An Efficient Finite-State Machine Implementation of Huffman Decoders," in *Information Processing Letters*, 1997, pp. 271–275.

[24] S. T. Klein and Y. Wiseman, "Parallel Huffman Decoding," in *IEEE DCC*, 2000, pp. 383–392.

[25] ——, "Parallel Huffman Decoding with Applications to JPEG Files," *The Computer Journal*, vol. 46, pp. 487–497, 2003.

[26] Y. Fang and A. Eleftheriadis, "A Syntactic Framework for Bitstream-Level Representation of Audio-Visual Objects," in *IEEE ICIP*, 1996, pp. II.426–II.432.

[27] A. Eleftheriadis, "Flavor: A Language for Media Representation," in *ACM Int. Conf. on Multimedia*, 1997.

[28] Y. Fang and A. Eleftheriadis, "Automatic Generation of Entropy Coding Programs Using Flavor," in *IEEE Workshop on MSP*, 1998, pp. 341–346.

[29] A. Eleftheriadis and D. Hong, "Flavor: A Formal Language for Audio-Visual Object Representation," in *ACM Int. Conf. on Multimedia*, ser. Proceedings, 2004, pp. 816–819.

[30] K. R. Rao and P. Yip, *Discrete Cosine Transform Algorithms, Advantages, Applications*. Academic Press, 1990.

[31] L. Van Sickle and K. M. Chandy, "Computational Complexity of Network Design Algorithms," in *IFIP Information Processing 77*, 1977, pp. 235–239.

[32] M. R. Garey and D. S. Johnson, *Computers and Intractability*. W. H. Freeman and Company, 1979.