Ternary Tree and Memory-Efficient Huffman Decoding Algorithm

Pushpa R. Suri[1] and Madhu Goel[2]

**[1]Department of Computer Science and Applications,
Kurukshetra University, Haryana, India**

**[2]Department of Computer Science and Engineering,
Kurukshetra Institute of Technology and Management, KUK**

Abstract

In this study, the focus was on the use of ternary tree over binary tree. Here, a new one pass Algorithm for Decoding adaptive Huffman ternary tree codes was implemented. To reduce the memory size and fasten the process of searching for a symbol in a Huffman tree, we exploited the property of the encoded symbols and proposed a memory efficient data structure to represent the codeword length of Huffman ternary tree. In this algorithm we tried to find out the staring and ending address of the code to know the length of the code. And then in second algorithm we tried to decode the ternary tree code using binary search method. In this algorithm we tried to find out the staring and ending address of the code to know the length of the code. And then in second algorithm we tried to decode the ternary tree code using binary search method.

Key words: Ternary tree, Huffman's algorithm, adaptive Huffman coding, Huffman decoding, prefix codes, binary search

1. INTRODUCTION

Ternary tree or 3-ary tree is a tree in which each node has either 0 or 3 children (labeled as LEFT child, MID child, RIGHT child).

Huffman coding is divided in to two categories:-

1. Static Huffman coding
2. Adaptive Huffman coding

Static Huffman coding suffers from the fact that the uncompressed need have some knowledge of the probabilities of the symbol in the compressed files. This can need more bits to encode the file. If this information is unavailable, compressing the file requires two passes. FIRST PASS finds the frequency of each symbol and constructs the Huffman tree. SECOND PASS is used to compress the file. We already use the concept of static Huffman coding [12] using ternary tree And we conclude that representation of Static Huffman Tree [12] using Ternary Tree is more beneficial than representation of Huffman Tree using Binary Tree in terms of number of internal nodes, Path length [8], height of the tree, in memory representation, in fast searching and in error detection & error correction. Static Huffman coding methods have several disadvantages.

Therefore we go for adaptive Huffman coding.

Adaptive Huffman coding calculates the frequencies dynamically based on recent actual frequencies in the source string. Adaptive Huffman coding which is also called dynamic Huffman coding is an adaptive coding technique based on Huffman coding building the code as the symbols are being transmitted that allows one-pass encoding and adaptation to changing conditions in data. The benefits of one-pass procedure is that the source can be encoded in real time, through it becomes more sensitive to transmission errors, since just a single loss ruins the whole code.

Implementations of adaptive Huffman coding: -

There are number of implementations of this method, the most notable are

1. FGK (Faller Gallager Knuth) Algorithm
2. Vitter Algorithm

We already use the concept of FGK Huffman coding [13] using ternary tree And we conclude that representation of FGK Huffman Tree using Ternary Tree is more beneficial than representation of Huffman Tree using Binary Tree in terms of number of internal nodes, Path length [12], height of the tree, in memory representation, in fast searching and in error detection & error correction.

We also already use the concept of Vitter Huffman coding [14] using ternary tree And we conclude that representation of algorithm V Huffman Tree using Ternary Tree is more beneficial than representation of Huffman Tree using Binary Tree in terms of number of internal nodes, Path length [8], height of the tree, in memory representation, in fast searching and in error detection & error correction.

All of these methods are defined- word schemes that determine the mapping from source messages to code-words on the basis of a running estimate of the source message probabilities. The code is adaptive, changing so as to remain optimal for the current estimates. In this way, the adaptive Huffman codes responds to locality, in essence, the encoder is learning the characteristics of the source. The decoder must learn along with the encoder by continually updating the Huffman tree so as to stay in synchronization with the encoder. Here we are given the concept of error detection and error correction. And the main point is that, this thing is only beneficial in TERNARY TREE neither in binary tree nor in other possible trees.

Now here we try to use the concept of adaptive Huffman decoding algorithm using ternary tree.

In 1951, David Huffman [2] and his MIT information theory classmates gave the choice of a term paper or a final exam. Huffman hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient. In doing so, the student out did his professor, who had worked with information theory inventor Claude Shannon to develop a similar code. Huffman built the tree from the bottom up instead of from the top down.

Huffman codes are widely used in the area of data compression and telecommunications. Some applications include JPEG [3] picture compression and MPEG video and audio compression. Huffman codes are of variable word length, which means that the individual symbols used to compose a message are represented (encoded) each by a distinct bit sequence of distinct length. This characteristic of the codeword helps to decrease the amount of redundancy in message data, i.e., it makes data compression possible.

The use of Huffman codes [7] affords compression, because distinct symbols have distinct probabilities of incidence. This property is used to advantage by tailoring the code lengths corresponding to those symbols in accordance with their respective probabilities of occurrence. Symbols with higher probabilities of incidence are coded with shorter codeword, while symbols with lower probabilities are coded with longer codeword. However, longer codeword still show up, but tend to be less frequent and hence the overall code length of all codeword in a typical bit string tends to be smaller due to the Huffman coding.

A basic difficulty in decoding Huffman codes is that the decoder cannot know at first the length of an incoming codeword. As previously explained, Huffman codes are of variable length codes. Huffman codes can be detected extremely fast by dedicating enormous amounts of memory. For a set of Huffman code words with a maximum word length of N bits, 2^N memory locations are needed, because N incoming bits are used as an address into the lookup table to find the corresponding code words.

A technique requiring less memory is currently performed using bit-by-bit decoding, which proceeds as follows. One bit is taken and compared to all the possible codes with a word length of one. If a match is not found, another bit is shifted in to try to find the bit pair from among all the code words with word length of two. This is continued until a match is found. Although this approach is very memory-efficient, it is very slow, especially if the codeword being decoded is long.

Another technique is the binary tree search method. In this implementation technique, Huffman tables used should be converted in the form of binary trees. A binary tree is a finite set of elements that is either empty or partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are referred to as left and right sub trees of the original tree. Each element of a binary tree is called a node of the tree. A branch connects two nodes. Nodes without any branches are called leaves. Huffman decoding for a symbol search begins at the root of a binary tree and ends at any of the leaves; one bit for each node is extracted from bit-stream while traversing the binary tree [1]. This method is a compromise between memory requirement and the number of Huffman code searches as compared to the above two methods. In addition, the coding speed of this technique will be down by a factor related to maximum length of Huffman code.

Another technique currently used to decode Huffman codes is to use canonical Huffman codes. The canonical Huffman codes are of special interest since they make decoding easier. They are generally used in multimedia and telecommunications. They reduce memory and decoding complexity. However, most of these techniques use a special tree structure in the Huffman codeword tables for encoding and hence are suitable only for a special class of Huffman codes and are generally not suitable for decoding a generic class of Huffman codes.

As indicated in the above examples, a problem with using variable codeword lengths is the difficulty in achieving balance between speed and reasonable memory usage.

Huffman is a fairly standard compression algorithm, and it is still commonly used. In order to do this you need a very simple tree. The nodes need a char and a number of occurrences (I used an unsigned short in mine). The tree does not need any of the standard BST methods, but you will need to be able to create a tree by merging two existing trees. All data is stored in the leaf nodes, frequency information is stored in every node in the tree.

- *The message "go eagles" requires 144 bits in Unicode but only 38 using Huffman coding*
- *A Huffman tree is a binary tree [10] used to store a code that facilitates file compression*

There are basically two concepts in Huffman coding

- Huffman Encoding
- Huffman Decoding

1.1 HUFFMAN ENCODING:-

This is a two pass problem. The first pass is to collect the letter frequencies. You need to use that information to create the Huffman tree. Note that char values range from -128 to 127, so you will need to cast them. I stored the data as unsigned chars to solve for this problem, and then the range is 0 to 255.

Open the output file and write the frequency table to it. Open the input file, read characters from it, gets the codes, and writes the encoding into the output file.

Once a Huffman code has been generated, data may be encoded simply by replacing each symbol with its code.

1.2 HUFFMAN DECODING:-

This can be done in one pass. Open the encoded file and read the frequency data out of it. Create the Huffman tree [14] base on that information (The total number of encoded bytes is the frequency at the root of the Huffman tree.). Read data out of the file and search the tree to find the correct char to decode (a 0 bit means go left, 1 go right for binary tree and 00 bit means go left, 01 bit means go mid, 10 bit means go right in case of ternary tree) This gets tricky since you read in 8 bit blocks, but the codes can be shorter or longer than that and there are no separators.

If you know the Huffman code for some encoded data, decoding may be accomplished by reading the encoded data one bit at a time. Once the bits read match a code for symbol, write out the symbol and start collecting bits again.

1.3 Huffman codes to binary data

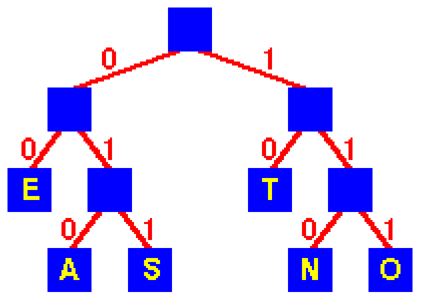
Since they are arbitrary in length, Huffman codes can be difficult to represent. The string data type has major advantages, the length [15] can be changed, and characters can be appended to them, or removed from them at either end. While you will probably use strings to represent the codes, you are not going to write a string of ones and zeros to the file. That would defeat the point of the program, which is file compression. You will need to convert from a string of length 8 to a char value which can written to the file, and do there reverse process as well. This problem is that of finding the minimum length bit string which can be used to encode a string of symbols.

One application is text compression:

What's the smallest number of bits (hence the minimum size of file) we can use to store an arbitrary piece of text?

Huffman's scheme uses a table of frequency of occurrence for each symbol (or character) in the input. This table may be derived from the input itself or from data which is representative of the input. For instance, the frequency of occurrence of letters in normal English might be derived from processing a large number of text documents and then used for encoding all text documents. We then need to assign a variable-length bit string to each character that unambiguously represents that character. This means that the encoding for each character must have a unique prefix. If the characters to be encoded are arranged in a binary tree:

Encoding tree for ETASNO



An encoding for each character is found by following the tree from the route to the character in the leaf: the encoding is the string of symbols on each branch followed.

For example:

String	Encoding
TEA	10 00 010
SEA	011 00 010
TEN	10 00 110

We already used the concept of Huffman encoding using ternary tree. Further I tried to use he concept of Huffman decoding [13] using ternary tree .we now implemented an algorithm which is used for Huffman decoding.

Huffman codes are widely used and very effective techniques for compressing data. Huffman's algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string (i.e. a codeword). The running time of Huffman algorithm on a set of n characters is o (log n).

Hasemian presented an algorithm [6] to speed up the search process for a symbol in a Huffman tree and to reduce the memory size. He used a tree clustering algorithm[13] to avoid high sparsity of the Huffman tree. However, finding the optimal solution of the clustering problem is still open. Moreover, the codeword of a single side growing [16] Huffman tree is different from the codeword of the original Huffman tree. Later, Chung gave a memory efficient data structure, which needs the memory size 2n-3, to represent the Huffman tree. In this paper, we shall purpose a more efficient algorithm to save memory space.

Now I try to introduce new concept of memory efficient Huffman decoding using binary search method.

Huffman code has been widely used in text, image and video compression. For example, it is used to compress the result of quantization stage in JPEG (Hashemain, 1995). The simplest data structure used in the Huffman decoding is the Huffman tree. Array data structure (Chen *et al.*, 2005) has been used to implement the corresponding complete ternary tree for the Huffman tree. However, the sparsity in the Huffman tree causes a huge waste of memory space for array implementation (Chen *et al.*, 2005) which needs $O(2^d) = O(n\log n)$ memory, where n is the number of source symbols and d is the depth of the Huffman tree. The number of nodes in the Huffman tree is 2n-1 and the search time is O(d) (Chen *et al.*, 2005).

Huffman decoding can be done in one pass. Create the Huffman tree (Pushpa and Goel, 2008) base on that information (The total number of encoded bytes is the frequency at the root of the Huffman tree.).

Here we are first presents a new array data structure to represent the Huffman tree using Ternary Tree. The memory required in the proposed data structure is nd = O(n), which is less than the previous ones. We then address an efficient Huffman decoding algorithm based on the proposed data structure; given a Huffman code, the search time for finding the source symbol is O(logn).

2 MATERIALS AND METHODS

The proposed data structure: Consider a set of source symbols $S = \{S_o, S_1, ..., S_{n-1}\}$ with frequencies $W = \{w_o, w_1, ..., w_{n-1}\}$ for $w_o \geq w_1 \geq ... \geq w_{n-1}$, where the symbol

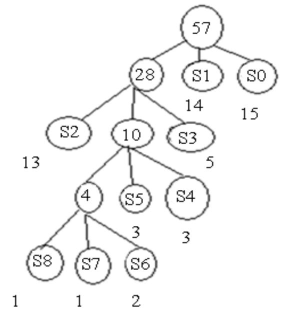


Fig. 1: An example of the Huffman tree

Table 1: An example of Huffman encoding

Symbol	Weight	Code words
S0	15	10
S1	14	01
S2	13	0000
S3	5	0010
S4	3	000110
S5	3	000101
S6	2	00010010
S7	1	00010001
S8	1	00010000

Table 2: Interval representation of symbols in Table 1

Interval address	Symbol	Starting
Int2	S2	00000000
Int8	S8	00010000
Int7	S7	00010001
Int6	S6	00010010
Int5	S5	00010100
Int4	S4	00011000
Int3	S3	00100000
Int1	S1	01000000
Int0	S0	10000000

S_i has frequency w_i. Using the Huffman algorithm to construct the Huffman tree T, the codeword c_i, $0 \leq i \leq n-1$, for symbol S_i can then can be determined by traversing the path from the root to the leaf node associated with the symbol S_i, where the left branch is corresponding to "00", mid branch is corresponding to "01" and the right branch is corresponding to "10". Let the level of the root be zero and the level of the other node is equal to summing up its parent's level and one. Codeword length l_i for s_i can be known as the level of S_i. Then the weighted external path length $\sum^{n=4}_{i=0} w_i\, l_i$ is minimum.

For example, the Huffman tree corresponding to the source symbols {s_o, s_1, ..., s_8} with the frequencies {15,14, 13, 5, 3, 3, 2, 1, 1} is shown in Fig. 1. The codeword set C {c_o, c_1, ..., c_8} is derived as {10, 01, 0000, 0010, 000110, 000101, 00010010, 00010001, 00010000}, where the length set L{ l_o, l_1, ..., l_8} is {2, 2, 4, 4, 6, 6, 8, 8, 8} is given in Table 1. here d = 4 is the depth of the Huffman tree.

The codeword generated from the Huffman tree could be treated as prefixes, which obey the prefix property, i.e., no codeword is the prefix, or start of the code for another codeword. Each prefix could be expressed as an interval with d-bit starting/ending addresses by appending 0's/1's to the prefix. For example, c_0 could be treated as an interval from address 10000000-1111111and c_5 starts from 00010100- 00010111. Since there is no empty branch in the Huffman tree, each address is occupied by exactly one interval of the prefix.

3 RESULTS & DISCUSSION

Lemma 1: Each address is occupied by exactly one interval of the codeword prefix.

Proof: If some address is not occupied by any interval, this means that there is undefined code in the compressed data and will result in error decoding. Otherwise, if some address is occupied by at least two intervals, this also means that the code corresponding to this address can be decoded as two code words and violates the prefix property.

Consequently, we could represent the Huffman tree by the set of intervals. As shown in Table 2. The range of each interval int_i (i.e., ending address-starting address+1) can derive the prefix length of the corresponding symbol s_i. For example, the range of int_1 is 64 = 01111111-01000000+1 (or 2^6), hence the height of S_i equals (2d-6 =) 2. Since each symbol corresponds to a leaf in the Huffman tree, each Huffman tree could be expressed as a unique set of intervals.

The data structure [13] of the intervals could be further improved by exploiting two properties: Ascending order and contiguity. Observing the leaves of the Huffman tree, the starting addresses corresponding to the leaves' intervals is in an ascending order from left to mid and mid to right. Thus the intervals with ascending order could be derived by reading the symbols from left to mid and mid to right (or by depth first search). In addition, these intervals are contiguous, i.e., ¥i, 0≤i<n-1, int_j whose starting address equals to the ending address of int_j plus one. Therefore, the successive intervals could be expressed by merely their starting addresses, as shown in Table 3.

Table 3: Contiguous interval representation of symbols in Table 1

Interval address	Symbol	Start add	End add
Int0	S0	10000000	10111111
Int1	S1	01000000	01111111
Int2	S2	00000000	11111111
Int3	S3	00100000	00101111
Int4	S4	00011000	00110000
Int5	S5	00010100	00010111
Int6	S6	00010010	00010010
Int7	S7	00010001	00010001
Int8	S8	00010000	00010000

```
1. Interval generation algorithm:
2. Input: The constructed d-level Huffman tree with n symbols
3. Output: N sorted mutually exclusive intervals
4. Generate (root, level, count, address) BEGIN
5. IF (root→leaf! = true) BEGIN
6. Count1 = Generate (root→left, level+1, count, address);
7. count1 = Generate (root→mid, level+1, count,
   address+$2^{d-level-1}$);
8. count = Generate (root→right, level+1, count1,
   address+$2^{d-level-1}$);
9. return count;
10. END
11. ELSE BEGIN
12. Int [count]. address = address;
13. Int [count]. symbol = root→symbol;
14. Return count+1;
15. END
16. END
```

In the following, the detailed algorithms to generate the intervals are presented. The algorithm is based on the depth first search. By traversing the Huffman tree, the begin address for each symbol is derived. The generated interval is then appended to the entry int[i], 0≤ i≤n-1, of the interval array. Since each node is traversed exactly once, the time complexity for the interval generation is O(n).

For each Huffman tree, the required storage for the interval representation is n entries. Each entry contains two fields: Address and symbol. The length of address is d bits and the storage complexity is O(n).

Decoding algorithm: With the array representation of Huffman tree, we can accomplish the decoding procedure by simple binary search. Given an input code '00100000', the fifth ((1+9/2) = 5) entry '00010100' is tested. Since '00010100' is smaller than '00100000', the third ((6+9)/2 = 8) entry '01000000' is evaluated. Since the input code is smaller than the 8th entry, the next entry compared is the fourth ((6+7)/2 = 7) one. Since the input code is equal to the 7th entry, the seventh entry corresponding to S_3 is the result. In the following, the length of the codeword must be derived to decide the starting bit of the next input code. The length can be calculated by subtracting the next interval to the matching one. If the input is totally equal to matching one, then no need to calculate next input code. Otherwise we have to repeat our algorithm and again calculate the starting address of the input bit. This extra calculation can eliminate the storage for the length of the codeword.

The detailed algorithm is listed below.

```
1. Decoding algorithm:
2. Input: The constructed n-entry interval array and the
   input code
3. Output: The symbol of the matching interval
4. Decode (code, start, end) BEGIN
5. Mid = [start+end/2];
6. IF (int [mid]>code) BEGIN
7. IF (mid = start+1)
8. Return start
9. Decode (code, start, mid-1)
10. END
11. ELSE BEGIN
12. IF (end = mid+1)
13. Return mid
14. Decode (code, mid, end)
15. END
16. END
```

4 CONCLUSIONS

The main contribution of this study is exploiting the property implied in the Huffman tree to simplify the representation of the Huffman tree and the decoding procedure. Moreover our algorithm can also be parallelized easily. We already showed that representation of Huffman Tree using Ternary tree is more beneficial than representation of Huffman Tree using Binary tree. The proposed data structure does not need any branch or pointer, thus is very compact as compared with the existing schemes.

ACKNOWLEDGEMENTS

The author Madhu Goel would like to thank Kurukshetra University Kurukshetra for providing me University Research Scholarship & support of Kurukshetra Institute of Technology & Management (KITM).

REFERENCES

1. BENTLEY, J. L., SLEATOR, D. D., TARJAN, R. E., AND WEI, V. K. A locally adaptive data compression scheme. Commun. ACM 29,4 (Apr. 1986), 320-330.
2. Chen, H.C., Y.L. Wang and Y.F. Lan, 2005.,"A memory efficient and fast Huffman decoding algorithm." Proceedings of the 19th International Conference 2005 IEEE. 69: 119-122.scialert.net/fulltext/?doi=itj.2007.776.779
3. DAVID A. HUFFMAN, Sept. 1991, profile Background story: Scientific American, pp. 54-58
4. ELIAS, P. Interval and regency-rank source coding: Two online adaptive variable-length schemes. IEEE Trans. InJ Theory. To be published.
5. FALLER, N. An adaptive system for data compression. In Record of the 7th Asilomar Conference on Circuits, Systems, and Computers. 1913, pp. 593-591.
6. GALLAGER, R. G. Variations on a theme by Huffman. IEEE Trans. Inj Theory IT-24, 6 (Nov.1978), 668-674.
7. Hashemain, "memory efficient and high-speed search Huffman Coding" IEEE Trans. Communication 43(1995) pp. 2576-2581.
8. *Hu, Y.C. and Chang, C.C., "A new losseless compression scheme based on Huffman coding scheme for image compression",*
9. KNUTH, D. E, 1997. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd edition. Reading, MA: Addison-Wesley, pp. 402-406
10. KNUTH, D. E. Dynamic Huffman coding. J. Algorithms 6 (1985), 163-180.
11. *MacKay, D.J.C., Information Theory, Inference, and Learning Algorithms, Cambridge University Press, 2003.*
12. MCMASTER, C. L. Documentation of the compact command. In UNIX User's Manual, 4.2Berkeley Software Distribution, Virtual VAX- I Version, Univ. of California, Berkeley, Berkeley,
13. Calif., Mar. 1984. ,
14. PUSHPA R. SURI & MADHU GOEL, Ternary Tree & A Coding Technique, IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.9, September 2008
15. PUSHPA R. SURI & MADHU GOEL, Ternary Tree & FGK Huffman Coding Technique, IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.1, January 2009
16. PUSHPA R. SURI & MADHU GOEL, A NEW APPROACH TO HUFFMAN CODING, Journal of Computer Science. VOL.4 ISSUE 4 Feb. 2010 .
17. ROLF KLEIN, DERICK WOOD, 1987, on the path length of Binary Trees, Albert-Lapwings University at Freeburg.
18. ROLF KLEIN, DERICK WOOD, 1988, On the Maximum Path Length of AVL Trees, Proceedings of the 13th Colloquium on the Trees in Algebra and Programming, p. 16-27, March 21-24.
19. SCHWARTZ, E. S. An Optimum Encoding with Minimum Longest Code and Total Number of Digits. If: Control 7, 1 (Mar. 1964), and 37-44.
20. TATA MCGRAW HILL, 2002 theory and problems of data structures, Seymour lipshutz, tata McGraw hill edition, pp 249-255

21. THOMAS H. CORMEN, 2001 Charles e. leiserson, Ronald l. rivest, and clifford stein.
22. Thomas H.Cormen Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Section 16.3, pp. 385–392.

Dr. Pushpa Suri is a reader in the department of computer science and applications at Kurukshetra University Haryana India. She has supervised a number of Ph.D. students. She has published a number of research papers in national and international journals and conference proceedings.

Mrs. Madhu Goel has Master’s degree (University Topper) in Computer Science. At present, she is pursuing her Ph.D. and working as Lecturer in Kurukshetra Institute of Technology & Management (KITM), Kurukshetra University Kurukshetra. Her area of research is Algorithms and Data Structure where she is working on Ternary tree structures. . She has published a number of research papers in national and international journals.