2015

Hoffman Coding – Project Report

JAFER KHAN SHAMSHAD
MALIK FAIZAN ZAFAR
MUHAMMAD FAIQ FARHAN
BS-CS(3A)

Abstract:

Huffman Encoding is an algorithm used to achieve loss-less data compression of data files. More frequently occurring symbols are mapped to a shorter bit string as compared to the less frequently occurring symbols. The algorithm is fairly efficient and takes O(NlogN) to construct the code. Huffman Encoding is the most efficient encoding possible so long as only character by character encoding is considered.

In the report to follow, a detailed description of the algorithm, its complexity and proof for working of the algorithm will be discussed.

Introduction:

Fixed length encoding is a method that uses a fixed number of bits to represent an alphabetic letter (as in all computers). For example, the string "ABRACADABRA" coded by using in the standard ASCII table yields the following binary string of 88 bits:

01000001010000100101001001000001010000110100000101000100010000010100001001010010010000001

To decode this bit-string “message”, it is simple to merely read off eight bits at a time and convert back to the letters according to the ASCII table. In such a fixed-length code any character requires the same number of (8) bits, regardless of whether it appears only once or many times in the text.

Clearly, if a smaller number of bits can be used to code letters occurring more frequently (e.g., ‘A’ in "ABRACADABRA"), while those less frequently occurring letters (e.g., ‘C’ or ‘D’) could be coded by a larger number of bits, and there should be an overall net savings of bits to code the entire given text. Of course, such a Variable length encoding of messages is adaptive to the text given, in that only letters appearing in the give message need to be coded, while any letter not appearing in the text needs no code at all. Consequently, this is a more effective file/data-compression technique where, instead of using fixed eight bits, only a few bits are used for frequently appearing letters and more bits for rare letters, thus, minimizing the total number of bits needed to encode the message and optimizing the overall space efficiency/economy. For example, if a text uses only 15 of the 26 letters in the alphabet, then only 4 or 5 bits might be enough, not the fixed 8 bits required by the ASCII table. For "ABRACADABRA", the strategy might simply use at most 2 bits for the 5 letters, e.g., A:0, B:1, R:01, C:10, D:11. Then the binary code uses only 15-bits: 0-1-01-0-10-0-11-0-1-01-1.

Since the number of bits to code a letter is no longer fixed but variable, there must be some kind of delimiter (separator) to mark the end of each code word (bit string), such as the dash “-“. The needed

number of delimiters (between every two code word) is only one less than the total number of letters, essentially doubling the number of letters to encode, which would clearly wipe out any advantage of using variable-length coding and any savings from using fewer bits to code more frequently occurring letters! The fundamental difficulty is actually that of distinguishing the bit string for the delimiter itself from all the other code words is basically impossible, as there are only two bits, 0 or 1, in the binary alphabet.

Fortunately, a tree-like data structure, called “min-heap”, can eliminate the need of delimiters. In a min-heap, no (bit-string) code word can be the prefix of any other code word (i.e., these two code words can not be mixed up), and no delimiter will be needed. For example, if A is coded by 0, R by 11, B by 100, D by 1010, and C by 1011 as seen from the min-heap, then "ABRACADABRA" is encoded as 01001101011010100100110 (only 23 bits). This bit string can be uniquely decoded with the min-heap without using any delimiters. Such an encoding and decoding scheme (using a min-heap, where keys can only reside at the leaf nodes) is known as the (famous) Huffman code. The unique code word for any key (letter) at the leaf is a binary string following the path from the root to the leaf in the Huffman min-heap, where "left" is always "0" and "right" always “1”.

To encode a message:

The Huffman tree is built according to the frequencies of occurrence for each letter in the text message, by a first pass through the text to count the frequency of each letter and build a table. Next, a simple (singleton) tree node is created for each letter, together with its (nonzero) frequency (as an attribute value of the tree node). These nodes are all put into a priority queue (heap) and ordered by the frequency values, smallest on top. Then, two nodes with the smallest frequencies will be taken from (the root of) the heap (and the hole healed), to construct a new node with these two nodes as the children, with the (parent) frequency equal to the sum of the children’s frequencies. This combined node is put back into the heap using the new sum frequency as the priority value. Continuing thus, where each time two nodes of the smallest frequencies are taken from the heap (they can be combined sub-trees, not just singletons), they are joined to form a new node with the sum as its own frequency and put back into the heap. Ultimately (when the heap becomes empty), all the (originally singleton) nodes are combined into one single (Huffman) tree, where any low frequency node (letter) ends up far down in the tree, with in a longer code word (path length). Then, applying one of three traversal methods (e.g., preorder) on this binary tree (min-heap) to get the code book (all the code words) for each letter in the text. As the text is read (on the 2nd pass), each letter can be looked up for its code word from the code book table and attached to the binary string of the already coded text, until the end of the text (EOF) is reached.

To decode a coded message, given the Huffman tree:

Read the coded binary string of the text, starting always from the root and proceed down the min-heap according to the bits read one by one from the bit string (i.e., 0 go left and 1 go right) until a leaf (letter) is reached (decoded). Return to the root and process (decode) the remaining bits similarly, until the end of the bit stream to get back (decode) the original text (no delimiter needed, just the tree).

Implementation:

The main data structures used in this program are priority queues (min-heaps), binary trees and maps. Priority queue is being employed for the forest of the disjoint trees built in the start of the algorithm where each of them has only the symbol stored in it. The two maps are being used for frequency counting in the text file and the other map maps characters to its equivalent bit-strings. As we pop two disjoint trees from the priority queue, we combine the symbols with two lowest frequencies and make another node which contains the sum of the frequencies of the two trees and make that the parent node and then re-insert it into the priority queue. We keep on doing this until we have only one tree in the min-heap. This is known as the Huffman tree.

We chose to write our encoded files in two parts. The first part contains information used to reconstruct the Huffman code (a header) and the second part contains the encoded data.

Header

In order to decode files, the decoding algorithm must know what code was used to encode the data. Being unable to come up with a clean way to store the tree itself, we chose to store information about the encoded symbols.

To reconstruct a traditional Huffman code, we chose to store a list of all the symbols and their counts. By using the symbol counts and the same tree generation algorithm that the encoding algorithm uses, a tree that matching the encoding tree may be constructed.

To save some space, we only stored the non-zero symbol counts, and the end of count data is indicated by an entry for a character zero with a count of zero. The EOF count is not stored in our implementation that encode the EOF, both the encoder and decoder assume that there is only one EOF.

Canonical Huffman codes usually take less information to reconstruct than traditional Huffman codes. To reconstruct a canonical Huffman code, you only need to know the length of the code for each symbol and the rules used to generate the code. The header generated by our canonical Huffman algorithm consists of the code length for each symbol. If the EOF is not encoded the total number of encoded symbols is also included in the header.

Encoded Data

The encoding of the original data immediately follows the header. One natural by-product of canonical Huffman code is a table containing symbols and their codes. This table allows for fast lookup of codes. If symbol codes are stored in tree form, the tree must be searched for each symbol to be encoded. Instead of searching the leaves of the Huffman tree each time a symbol is to be encoded, our traditional Huffman implementation builds a table of codes for each symbol. The table is built by performing a depth first traversal of the Huffman tree and storing the codes for the leaves as they are reached.

With a table of codes, writing encoded data is simple. Read a symbol to be encoded, and write the code for that symbol. Since symbols may not be integral bytes in length, care needs to be taken when writing each symbol. Bits need to be aggregated into bytes. Our implementation use two classes that we have created ourselves to mimic bit-writing and bit-reading (BitReader and BitWriter Class) to handle writing any number of bits to a file.

Decoding Encode Files

Like encoding a file, decoding a file is a two-step process. First the header data is read in, and the Huffman code for each symbol is reconstructed. Then the encoded data is read and decoded.

We have read that the fastest method for decoding symbols is to read the encoded file one bit at time and traverse the Huffman tree until a leaf containing a symbol is reached. We do know that the tree method is faster for the worst case encoding where all symbols are 8 bits long. In this case the 8-bit code will lead to a symbol 8 levels down the tree, but a binary search on 256 symbols is O(log2(256)) or an average of 16 steps.

Since conventional Huffman encoding naturally leads to the construction of a tree for decoding, we chose the tree method here. The encoded file is read one bit at a time, and the tree is traversed according to each of the bits. When a bit causes a leaf of the tree to be reached, the symbol contained in that leaf is written to the decoded file, and traversal starts again from the root of the tree.

Results:

Nearly around 50% compression rate is achieved with larger text files and the time taken by the algorithm truly is in the order of O(NlogN) as we have timed the compression and decompression functions.

Enter your file name (with extension): koran.txt

Encoded data bits: 3576069

Tree bits: 689

Padded bits: 2

The number of bits taken by the original file is: 803301

The number of bits taken by the compressed file is: 3576782

Enter your file name (with extension): ee710.txt

Encoded data bits: 3766546

Tree bits: 859

Padded bits: 3

The number of bits taken by the original file is: 1037461

The number of bits taken by the compressed file is: 3767429

Discussion & Conclusion:

Binary trees are useful data structures for encoding and decoding information. The Huffman algorithm generates optimal binary trees for this purpose and can do so efficiently. Binary search trees are useful data structures for storing information that is to be searched; they also produce good codes.

Greedy binary search trees, which are nearly optimal, can be generated very efficiently. The greedy method of algorithm design does not yield optimal solutions but may still yield very good solutions. Again it was demonstrated that careful selection of data structures can significantly change the time requirements of an algorithm.

This implementation of the algorithm could be made faster if we have to use two queues for it. One queue could contain all the leaf nodes (symbols) and the other for resulting parent nodes. This could clearly become O(N) and not O(NlogN).