CSCE 2202-02, Term Project Spring 2022

Text compression project

Huffman Coding

Department of Computer Science and Engineering, AUC

Mohamad Sherif, Hamza Algobba, Amr Abdelbaky, Roaa Bahaa

Abstract:

This is a paper regarding the technique used by Huffman's algorithm to compress text files, as well as a time and space analysis, a display of the experimentation results, and an overview of the data structures used in the program.

Keywords: Huffman coding, file compression, lossless

1. Introduction

Text compression is very important to save storage space on any device. It is also important to decrease the size of data being transferred between devices via the internet or USB, which will in turn decrease the time taken to transfer the files. The file compression algorithm tackled in this project is Huffman coding. It was developed by David Huffman in 1951 at MIT when he was working on a term paper Ken (1991). Huffman Coding is considered an example of lossless compression, which is a type of compression where all the data is preserved during the process.

2. Problem Definition

Dealing with large text files sometimes is a source of some complications and difficulties. For example, it can lead to a lack of storage in whatever device is used to store the files, and no

matter how large the memory of the device is, it is always better to make the best use of it to be able to store the largest number of files. More importantly, large-sized files make them much more complicated to be sent from one device to another, or in a network of devices used in workplaces, for instance. One of the reasons behind this problem is that each character in the ASCII code takes nearly 7 bits of the memory, nevertheless, most devices are programmed to reserve a complete byte (8 bits) for each character. Thus, this amount of bits multiplied by many characters for large text files results in wasting a large space of the memory. This is why the solution to such a problem mainly depends on trying to allocate fewer bits for each character, and this is what Huffman coding does by transforming these characters into a group of binary numbers (zeroes and ones) to be able to decompress the file to reduce its size. Huffman coding provides more compression if a text has a higher frequency of characters as well (Rahman, 2020). With the size reduced, all complex operations can be done, and then the file should be uncompressed again without losing any of the original stored data.

3. Methodology

Huffman coding substitutes the standard ASCII codes for symbols with custom assigned codes. Some variables affect how we assign those custom codes. Mainly, how many times each symbol is repeated in the file (frequency), and the probability of encountering this symbol(Klein Et al., 2020). Ideally, the symbols with high probability will be given a short custom code, while less frequent/probable symbols will be given relatively long custom codes. This way, when we calculate the number of bits in the code and sum the products of each element's frequency with its custom code length, we will get a small number of bits.

In order to compute the custom codes, the Huffman algorithm sorts the symbols ascendingly with respect to their probabilities. Each symbol is represented as a node. Initially, all nodes are parentless, but after sorting, we group the two smallest nodes by concatenating their symbols and summing their probability. Then, we set the grouped node as the parent of the two smallest nodes. We then sort the whole group again excluding the two smallest nodes and including the new parent node. This sequence of steps will be repeated until all symbols are grouped in one root node of probability 1.0. This is a bottom-up approach that gives Huffman coding an edge over other compression algorithms in terms of saving space. At the end of this process, we would have a binary tree such that its internal nodes will each have at least two symbols, and each of its leaf nodes will have exactly one symbol. Each leaf node will be given a unique custom code determined by its position relative to the root node. The convention is that each right turn taken in the sequence of steps from the root to the leaf is denoted as 1, and each left turn is denoted by 0. It is not necessary to follow the convention; however, consistency is important when it comes to encoding and decoding the files.

4. Specification of Algorithms to be implemented in :

- The heapsort algorithm is used to sort the leaf nodes with respect to their symbols' probabilities.
- A modified version of the inorder tree traversal has been used to record the custom codes of the symbols Data structures used:
- The nodes of the Huffman tree are stored in a vector, this allows flexible memory allocation, pushing, and popping.
- An ordered map is used to link every symbol with its corresponding custom code.

- The tree data structure is used to store the nodes containing the symbols, and to compute the custom codes.

5. Data Specifications

The type of characters that this program will work on is ASCII symbols, which include upper and lower case letters, numbers, special characters (i.e. ! @ # ?), and the line terminating command “\n” (Klein Et al., 2020).

7. Experimental Results

We decided to implement the code in two independent ways, the first is to include the table of custom codes with the encoded script in the same file, which will decrease the ratio between the file sizes before and after compression. The other way is to store the table in a separate file than the encoded text, which will create a very small ratio between the encoded script and the original.

The following results were obtained by analyzing the file before and after compression to one of the example files to provide a better analysis of the algorithm :

Section from Output on file: alphabet.single

average code length: 4.80767

efficiency: 97.7723

entropy: 4.70057

input file size: 100002

output file size: 60174

compression ratio: 0.601728

8. Analysis and critique

The Huffman's coding used in the project starts by inserting all (Si)s in binary tree nodes, so this takes O(n) in the worst-case scenario. Then it uses a minimum heap in order to store each symbol Si and its probability Pi, Then it executes the heapsort which takes O(n log n) time in all cases, where n is the number of symbols, then the algorithm proceeds to decrease the size of the vector to be sorted by one, and then sorts again. This process is repeated n - 1 times until the size of the vector is equal to one. In each iteration, the size of the vector to be sorted is decremented by one.

Analysis of the multiple sortings with the decreasing vector size:

N log n + (n - 1) log(n - 1) +.....+ (n - k)log(n - k) , where n - k = 1

$$T(n) = \sum_{i=1}^{n} ilogi$$

This results in a time complexity of $O(n^2 \log n)$ in the worst case

Which is an acceptable time complexity given that the priority in any compression algorithm is to save as much space as possible.

A critique of our algorithm would be that we do not need to heap sort on every level in the tree. We would just need to sort once at the beginning, then whenever we create a merged node, we place it in its correct position without sorting the whole vector again, which is already sorted except for the merged node.

9. Pseudo code

ALGORITHM HuffmanTree (S , P)

{

Store each symbol si , i = 1.. n in a parentless node of a binary tree.

Insert symbols and their probabilities pi in a minimum heap of probabilities.

Repeat {

Remove lowest two probabilities (pi , pj) from heap.

Merge symbols with (p i, pj) to form a new symbol (sisj) with

probability pij = pi + pj

Store symbol (sisj) in a parentless node with two children si and sj

Insert pij and its symbol (sisj) into the heap

} until all symbols are merged into one final symbol (root)

For each symbol (i), trace path from root to each leaf (symbol) to form the bit string Ci for that symbol. Concatenate “0” for a left branch, and “1” for a right branch.

Return the code words C

}

Adapted from professor Goneid’s assignment

10. Comparison

Another algorithm used for lossless file compression is Shannon fano coding, named after Claude Shannon and Robert Fano based on a methodology similar to that of Huffman coding. But unlike Huffman coding, which uses the idea of prefix codes, Shannon fano coding uses a cumulative distribution function, leading to it being less efficient and producing non-optimal codes on some occasions (Khan, 2012). Out of both compression coding algorithms, Huffman has the upper hand as it is considered the more optimal and consistent algorithm. This can be credited to Huffman constructing his tree in a bottom-up manner, unlike Fano who did it in a top-down manner.

11. Conclusion

Huffman coding as analyzed above uses a very optimal and efficient methodology that guarantees the success of the process of file compression and thus saves much-needed space. When compared to other algorithms in the field of file compression, it provided better consistency and optimization. Overall, Huffman coding is considered one of the most efficient and optimal algorithms that are being used in the field of file compression.

References

Khan, M. A. (2012). Evaluation of basic data compression algorithms in a distributed environment. *Journal of Basic and Applied Sciences, 8*(2) Retrieved from here.

Klein, S. T., Saadia, S., & Shapira, D. (2020). Forward looking huffm

an coding. *Theory of Computing Systems, 65*(3), 593-612. https://doi.org/10.1007/s00224-020-09992-7

Rahman, M. A. (2020). Burrows–Wheeler transform based lossless text compression using keys and huffman coding. *Symmetry (Basel), 12*(10), 1654. https://doi.org/10.3390/sym12101654

Huffman, Ken (1991).Profile: David A. Huffman: Encoding the "Neatness" of Ones and Zeroes. *Scientific American*: 54–58.

Retrieved from http://www.huffmancoding.com/my-uncle/scientific-american

Dr: Goned's slides .