Shravan Honaganahalli

Computer Science CS – 430

25 November 2021

Report of the Analysis of Huffman Algorithm from the developed Huffman Binary tree encoder program

Introduction

In this report we will be looking at the various observations and data points I have obtained while working on this project. This is not limited to only algorithm complexity but also to the broader range of Huffman algorithm implementation in many applications. Recently, a renewed interest in data compression methods has been observed. As a result, new software packages for data compression in microcomputer systems are becoming commercially available. Compression enables us to reduce the input length of textual data even by half; in the case of graphical data, the reduction could be several times greater, thus increasing the global effectiveness of a computer system. The Huffman algorithm for the construction of minimum redundancy codes one of the most important data compression algorithms. However, it is more important from the theoretical than from the practical point of view. If the frequencies of character occurrences in a data base are known and are statistically independent, then the Huffman algorithm produces a compact code, i.e., a code with the shortest average length of a co word and which is uniquely decodable. The practical significance of the algorithm is not particularly large. The reason is simple: it adjusts the code word lengths to the frequencies of character occurrences and, therefore, all the Huffman code processing algorithms proposed so far

manipulate single bits or, at best, short variable-length strings of bits. This in turn brings about several disadvantages: -the processing algorithms are far more complicated than in the case of fixed-length codes. Coding and decoding is very slow. It is difficult to code the algorithms by using high-level languages. In order to effectively implement a compression algorithm, one should at first collect data on statistical properties of the processed files. ln comparison with other algorithms, collecting the data for the Huffman algorithm is fairly simple.

Moreover, the Huffman codes could achieve substantial compression of files which have very diverse statistical properties. Other, more popular algorithms do not have such advantages. For that reason, the Huffman algorithm is recommended for reducing the size of different types of files in microcomputer systems and the size of digital images

Algorithm design and complexity analysis:

Our interest in the algorithm comes mainly from its importance for generating optimal evaluation trees in the compilation of expressions. Not only does the algorithm build optimal trees with respect to execution time, space usage, and roundoff error for many classes of limited expressions, it does so very efficiently. If N is the number of leaves in the tree to be constructed, Huffman's algorithm can be implemented in time O(N log N) when a fast priority queue is used. Moreover, we have seen that this time bound can be reduced to O(N) if the leaf weights are given in sorted order. (This suggests that the complexity of the algorithm is (N log N), since sorting is at least that difficult.) The two variations of the Huffman algorithm mentioned above are based on the same construction process but use different tree cost functions and node merging methods. In the construction process each node has some associated weight, and these

nodes are combined along with their weights to form a new weighted internal (parent) node in the tree. Construction terminates when all nodes have been combined into a final root node. The weighted path-length variation produces parent nodes having weights equal to the sum of the weights of its sons, while the tree-height variation uses the maximum of the son weights plus some nonnegative constant. This will all be discussed in greater detail below It is not immediately apparent why these two apparently unrelated incarnations of the Huffman algorithm both produce optimal trees. From the point of view of compiling it would be nice if we could use instances of the same construction to obtain trees optimal with respect to yet other cost measures besides tree height and path length. For example, suppose we wish to construct parse trees for parallel evaluation of sums of positive numbers, optimal with respect to some measure of both roundoff and time used. Since error bounds in this case correspond to path-length, and execution time to tree height, an optimal parse tree cannot be constructed using the Huffman algorithm unless a node-merging method more complicated than the two above is used. This problem raises the following question addressed and investigated in this paper: for which problems will the Huffman algorithm produce optimal trees under exactly which cost? We will identify a wide class of weight combination functions (encompassing the two standard methods above) which all produce optimal tree with the Huffman algorithm under corresponding classes of tree cost functions. We also tie together some results from information theory and the theory of convex functions in studying the optimality in these instances. Construction of a (full) binary tree on these leaves is then affected by n merges of pairs of "available" nodes. Each node in the pair is marked unavailable after a merge and their father (the result of the merge) is marked available, having as his weight some function F of the weights of its sons. Leaves are initially all marked available, of course. Note that n merges are necessary and sufficient since binary trees on n + 1

leaves have n internal nodes. Each internal node defines the root of a binary subtree of the constructed tree, which implies that tree construction can be defined inductively in terms of forests (collection of trees) in the obvious way. The construction begins with a forest of n + 1 one-node trees and repeatedly reduces the number of trees by 1 via root merge operations until only one tree is left.

Conclusion

Throughout the creation of this project, the thorough understanding of the practical implementation of the Huffman algorithm does not suffice, we must go back into the primitive understanding of binary trees and understand how they have contributed to this big picture. One of the most important structures in programming is binary trees. The concepts of path length and weighted pathlength of a tree are introduced in the investigation of binary trees.

. Using the input from the user, we build a Huffman Tree. This is a binary tree where each leaf represents a unique character from the input. Once this is done, we use the structure of the tree to map each unique character to a specific encoding. These encodings are then used in conjunction with the user input to create the series of bits that represents the input. Metadata is constructed using the character mappings, and then added to an array in preparation for writing to a file. The series of bits is divided into one byte chunks, and appended to the array, completing the download file. When an encoded file is uploaded, the process is essentially reversed. The mapping metadata is used to reconstruct the user input from the bit series, and then that input is used once again to reconstruct the tree. The compression percentage is calculated by subtracting the size of the encoded file divided by what would have been the size of the plain text file from one and displayed in a nifty bar below the output. These characteristics play a substantial role in the analysis of the efficiency of algorithms since they are associated directly with the time of

execution of the algorithm and the representation of the trees in the electronic computer. The weighted path length is used, in particular, in cases when the tree is utilized as a search procedure. A Huffman algorithm described in this report permits the connection of a binary tree with minimal weighted path length for given probabilities (weights). Depending on the domain of application the Huffman trees can allow of different interpretations: optimal search procedure in search theory prefix code with minimal mean word length in information theory etc. We have developed a very feasible linear time algorithm to construct optimal alphabetic binary Huffman trees for records. For arbitrary arrangements very good approximations have been found in linear time. It leads to a new algorithm for constructing ordinary Huffman trees and the argument that its complexity is essentially optimal has been found. The introductions of weakly stable alphabetic trees which are easier to maintain than optimal trees when accessed using frequencies of characters are modified or changed at run time as per the project. Using the appropriate data structures a perfectly stable tree on n records has been constructed in n logn steps