IMPLEMENTATION OF THE HUFFMAN CODING ALGORITHM

SIDDHANT KHANNA – 2K20/CO/441
VARIN THAKUR – 2K20/CO/475

Abstract

This project aims to implement the Huffman Coding Algorithm, a lossless data compression algorithm utilising some of the data structures taught in the course, in the programming language of C++.

1 Introduction

Data compression is the process of encoding, restructuring or otherwise modifying data in order to reduce its size. Fundamentally, it involves re-encoding information using fewer bits than the original representation. There are two Types of Data Comprssion, one which involves some loss of data (Lossy Data Compression) and the other which reduces the size without any loss in data (Lossless Data Compression). The process of compressing data has a vide variety of advantages ranging from a decreased use in storage to an increase in efficiency of file transferring operations and bandwidth consumption.
The Huffman Coding Algorithm is an Algorithm devloped by David A. Huffman for Loseless Data Compression. It works on the principle of assigning specific codes to various characters, based upon their frequencies in the input.

2 Idea behind Huffman Coding

The idea behind Hufman coding is that we can use a specific number of distinct bits to represent a limited amount of characters used in the text instead of using the 8-bit approach, to save our space. The codes given to the characters are of variable length. This is decided by their frequencies in the given string of text. If the character is more frequent, i.e. if its frequency is high, assign a code of small length, so that more space can be saved. It should also be ensured that no code given to a character is a prefix code for another code, This means the binary code string for a character shouldn't be a substring of any other character code. This is done to remove the ambiguity while decoding.

3 Approach

The complete set of Binary Codes for the various characters can be represented in the form of a Binary tree known as a Huffman Tree. In this tree the Leaf Nodes are the ones storing the characters along with their frequencies. The code for that character is given when we travel from root to that leaf node containing that character. By convention, travelling the left child appends a 0 while travelling through the right child appends a 1 in the encoded string.

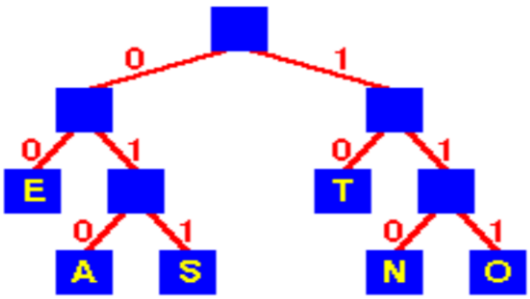

Figure 1: A Huffman Tree

The characters which occur more are assigned with smaller binary codes and are near to the root whereas the characters which are less frequent are far away from the root and have longer binary codes.

4 The Algorithm

The algorithm for implementing the Huffman Coding is as follows-

1) Make characters with their corresponding frequencies as leaf nodes.
2) From this character set, select the 2 characters with the minimum frequencies. Connect them with a parent storing the sum of frequencies of its children. Now the comparisons have to be made considering this resulting frequency and the left frequencies.
3) Repeat the steps of finding the two minimum frequencies and connecting them to the parent until all the frequencies are under a single parent which is the root in the Huffman Tree.
4) Now travel from the root to every leaf node to get the codes for the character appending a 0 whenever going to the left child and 1 whenever going to the right child.
5) Using the codes, encode and decode.

5 Our Implementation

We have declared 3 classes -

1) node
   Private data members
   a) int frequency – to store the frequency of entered character.
   b) node * left – a pointer to the left child of the node.
   c) node* right – a pointer to the right child of the node.
   d) string data – to store the value of character entered.
   e) int code – to store the value based on whether the node is a left child (0) or right child (1).
   f) string char_code – to store the prefix code for the character entered.

   Public members
   a) node(string, int) – a constructor to initialize the data and frequency.
   b) void node_create(string, int) – creates a node with data and frequency as given by the user.
   c) int get_frequency() – returns the frequncy for the given node.
   d) void assign_left(node *lchild) – Assigns the left child for a given node.(Used to pass a border case.)

   Declared classes Minheap and HuffmanTree as friends of node class so they can access the private data members.

2) Minheap
   Private data members
   a) node* Heap – to store the base address of the Min Heap array.
   b) int size – to store the size of the Min Heap array.

   Public members

   a) void Insert(node*) – It is used to insert a node in the Min Heap. When a node is inserted, it inserts it at the end index to preserve the property of heap as a complete binary tree. It then compares the nodes frequency to its parent and if it is less than the parent's frequency the swaps them. The operation continues till the node reaches index 1 i.e., the top or if the parent's frequency is less than the nodes frequency.
   b) node* Delete() – It is used to delete a node from the Min Heap and returns a pointer to the deleted node. It replaces the root node with the last element and stores the address of the root node in a pointer. To keep the properties of the min heap we heapify the new root. As long as the node has a direct child we compare the frequencies of its children and take the one with the least. We then compare the child's frequency with that of the node and if the child's frequency is less we swap them. If the node has only one child we can skip the step of comparing frequencies of the children and only the parents and its child's frequencies are compared. It then returns the pointer to the deleted node.

3) HuffmanTree

Public menbers

a) node* root – stores the root of the huffman tree
b) Hufffman() – constructor to initialize the root to NULL
c) void Assign_root(node*) – updates the root variable once the huffman tree is ready
d) node* InsertHuffman(node*, node*) – creates a new node with frequency equal to the sum of the frequencies of the two deleted nodes from the min heap and returns a pointer to that node.
e) void AssignCode(node*) – assigns codes to the nodes based on whether the node is a left child or right child using recursion. If that node is a left child then its code will be 0 else 1.
f) void CharCodes(node*, string, Table*) – It is used to assign prefix codes to the leaf nodes. It traverses the entire Huffman Tree using recursion and stores the code for each node in a string. When we reach the leaf node it assigns the prefix codes in the char_code variable. It stores the character with its respective prefix code in an array of data type Table. The recursion continues till we reach a null pointer.
g) void Encode(string, Table* ,int) – It is used to encode the given string. It loops through the string and uses the table consisting of the characters and their respective prefix codes. For each character it searches its prefix code in the table. If the match is found it copies the prefix codes in a string. Once we exit the loop it prints the encoded string.
h) void decode(char*, Table*, int) – It is used to decode the given encoded string. It loops through the string and through the table and compares each prefix code we have in the table to a sub string. If it finds a match the character corresponding to that prefix code is stored in a string. Once we exit the loop it prints the decoded string.

Declared a structure table which stores characters and their corresponding prefix codes.

In the main function we take the desired inputs of characters and their frequencies. Then we create a min heap using insert function of class Minheap. Using the delete function we delete the 2 nodes from the min heap and using InsertHuffman create a new node having frequency equal to the sum of the frequencies of the two nodes. This operation continues till heap size gets reduced to1. Then the node at the top of the heap is our root so we make the necessary assignment. We now call function CharCodes to update the prefix codes of each leaf node. With the Huffman tree ready we call functions Encode and Decode to print the encoded and decoded strings.

6 Time Complexity

The time complexity of our program and the algorithm in general is $O(nlogn)$. This due to the insertion and deletion in a minimum heap. The best case takes the time of $O(n)$. This happens when the frequencies of the characters are sorted by default. The worst case is when the frequencies are in the recursive relation of that of a Fibonacci sequence.

$$f(n) = f(n-1) + f(n-2)$$

The encoding and decoding of a given string takes the time of $O(n * length(string))$.

7 Pros and Cons

The advatages of our implementation are as follows –

1) Since variables code lengths are assigned, this saves a lot of space.
2) Binary codes generated are prefix-free, hence ambiguity is avoided.

The disadvantages of our implementation are as follows -

1) Extra space is used.
2) It happen in 2 phases, so the process is slower than other Lossless Algorithms.
3) Variable length codes make it difficult to check if the file is corrupt.

8 Applications

Huffman Coding is used in a plethora of real life applications. Some of them are –

1) Used in compression formats like GZIP, PKZIP, BZIP2 by applications like WinZip and WinRAR.
2) Used by multimedia codecs like JPEG and MP3.

References

https://stackoverflow.com/

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

https://en.wikipedia.org/wiki/Huffman_coding

https://nishikatyagi.medium.com/real-world-applications-of-huffman-coding-b6cf46fd0663

https://www.youtube.com/