Logamithran M.R., 22BRS1373

Anirudh Sridhar, 22BCE5252

Kaviya Elakkiya M.

BCSE102L

02/11/2023

Huffman Coding for Frequency Table Compression: A Novel Approach

by Logamithran M.R. and Anirudh Sridhar

Abstract:

This research paper explores a novel program that employs the Huffman coding algorithm to compress frequency tables into binary trees, providing efficient storage and transmission of frequency data. We discuss the algorithm's theoretical foundation, the implementation details of our program, and the evaluation of its performance against alternative methods. Our findings demonstrate the effectiveness of this approach in reducing the size of frequency tables, making it a valuable tool in various applications.

1. Introduction:

The Huffman Coding algorithm is a widely used and efficient compression algorithm in the field of data encoding, designed to achieve lossless data compression, where data is encoded in a manner that minimizes its storage or transmission size without any loss of information. The Huffman encoding algorithm works by constructing a binary tree called a Huffman tree, where characters with higher frequencies are represented by shorter binary codes, while characters with lower frequencies are represented by longer codes. This results in variable-length codes, with more frequent characters having shorter codes, allowing for efficient compression and decompression.

1.1 Approach:

A frequency table, also known as a frequency distribution, is a statistical representation of data that displays the frequency (count or number of occurrences) of various values or categories within a dataset. It organizes data into distinct bins or categories and records how many times each value falls into each category. Frequency tables provide a concise summary of a dataset, making it easier to understand the distribution of data. This is particularly useful when dealing with large datasets. When dealing with massive datasets and a large amount of frequency tables, the time to access data may scale to an unaffordable level. Huffman coding can be extended to compress frequency tables by encoding the frequency of occurrence of each symbol in a way that minimizes the overall storage or transmission size. In this context, the frequency table represents the frequencies of various symbols or events. The algorithm constructs a Huffman tree based on the frequencies of these events, where higher frequency events correspond to shorter codes and lower frequency events are assigned longer codes.

While the Huffman Coding algorithm has been used before in the fields of compression to shorten file size, it has not been used previously to compress the data present in a frequency table. In this paper, we aim to apply the Huffman Coding algorithm to the data present in a frequency table, creating a code for each element present between frequency tables as we aim to analyze and compress the file size of frequency tables.

2. Research Survey:

The major application of the Huffman Coding algorithm is in the compression of image and audio files, and is applied in many major compression formats like .ZIP (Sharma and Gupta). It has also been used in steganography, the practice of hiding messages within files and documents, where the secret message is Huffman-encoded then embedded into the original message, such that the receiver is able to decode the secret message (Das and Tuithung).

3. Huffman Algorithm:

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and n-1 internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
- Remove the two nodes of highest priority (lowest probability) from the queue
- Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
- Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log n) time per insertion, and a tree with n leaves has 2n−1 nodes, this algorithm operates in O(n log n) time, where n is the number of symbols.

If the symbols are sorted by probability, there is a linear-time (O(n)) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

- Start with as many leaves as there are symbols.
- Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).

- While there is more than one node in the queues:
- Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
- Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
- Enqueue the new node into the rear of the second queue.
- The remaining node is the root node; the tree has now been generated.

Once the Huffman tree has been generated, it is traversed to generate a dictionary which maps the symbols to binary codes as follows:

- Start with current node set to the root.
- If node is not a leaf node, label the edge to the left child as 0 and the edge to the right child as 1. Repeat the process at both the left child and the right child.
- The final encoding of any symbol is then read by a concatenation of the labels on the edges along the path from the root node to the symbol.

It is generally beneficial to minimize the variance of codeword length by breaking ties between queues through choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

4. Implementation of the Huffman Algorithm

As discussed previously, we see that the Huffman encoding algorithm is effective in compression of data that contains repeating symbols. One such kind of data that contains repeating data in this regard are frequency tables. As discussed in the approach, we can compress the data of a large number of frequency tables using the Huffman Encoding algorithm to represent the table in the form of a binary tree. As the encoding algorithm represents the data with the most occurrences at the top of the table (Huffman), we also obtain a representation for viewing the data present in the table in the order of most occurring, to least occurring.

In the implementation of this idea, we:

- obtain the size of the frequency tables.
- Read the occurrences of each element present in the table.
- Create a priority queue to sort the elements based on their frequency.
- Create a Huffman coding tree based on the frequency of the elements present in the table.
- Return the elements in the binary tree representation made by the Huffman Coding algorithm

The code for this implementation is presented in Appendix A-1.

This solution allows us to reduce access times to the frequency tables by instead, accessing the binary tree representation of the frequency table, and also sort the given frequency table, as the highest-occuring element has a specific code while the lowest-occuring element has a specific code.

5. **Conclusion:**

The Huffman coding algorithm is an efficient, smart algorithm capable of analyzing data and compressing information in a form that is easy to transmit and receive, allowing us to transfer data that would normally take much longer at faster rates. We use the Huffman Encoding algorithm to represent data present in frequency tables in a manner that is quicker to access as well as more readable, as it sorts the elements based on their priority in frequency and displays them in that order.

6. **References:**

1. Huffman, David A. “A Method for the Construction of Minimum-Redundancy Codes.” *Proceedings of the IRE*, vol. 40, no. 9, Institute of Electrical and Electronics Engineers, Sept. 1952, pp. 1098–101. https://doi.org/10.1109/jrproc.1952.273898
2. Sharma, Komal, and Kunal Gupta. “Lossless Data Compression Techniques and Their Performance.” *IEEE*, May 2017, https://doi.org/10.1109/ccaa.2017.8229810 .
3. Srikanth, Sure, and Sukadev Meher. “Compression Efficiency for Combining Different Embedded Image Compression Techniques With Huffman Encoding.” *IEEE*, Apr. 2013, https://doi.org/10.1109/iccsp.2013.6577170.
4. Das, Rig, and Themrichon Tuithung. “A Novel Steganography Method for Image Based on Huffman Encoding.” *2012 3rd National Conference on Emerging Trends and*

Applications in Computer Science, Institute of Electrical and Electronics Engineers, Mar. 2012, https://doi.org/10.1109/ncetacs.2012.6203290 .

7. Appendix:

7.A) Code of the Implementation

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <queue>
#include <map>
using namespace std;

//#define pqueue(T, cmp) priority_queue<T, vector<T>, decltype(cmp)>

//Definition of the node of the huffman tree.
class MHnode {
    public:
        char data;
        unsigned freq;

        MHnode *left, *right;

        MHnode(char data, unsigned freq) {
            left = right = nullptr;
            this -> data = data;
            this -> freq = freq;
        }
};

//compare function for the priority minheap queue
bool cmp(MHnode* l, MHnode* r) {
    return (l->freq > r->freq);
}


priority_queue<MHnode*, vector<MHnode*>, decltype(cmp)*> MinHeap(cmp); //
creation of priority queue for sorting
                                                                        //
elements based on frequency

class Huffmancoding {
    public:
        static map<char, string> codes;

        //function creates a binary tree from the frequency table
        static void Codes(map<char, int> m1 ,int size) {
            MHnode *left, *right, *top;
            for (auto it = m1.begin(); it != m1.end(); it++) {
                MinHeap.push(new MHnode(it->first, it->second));
            }

            while (MinHeap.size() != 1) {
                left = MinHeap.top();
                MinHeap.pop();

                right = MinHeap.top();
                MinHeap.pop();

                auto temp = left->freq + right->freq;
                top = new MHnode('$', temp);

                top->left = left;
                top->right = right;
                MinHeap.push(top);
            }

            storecodes(MinHeap.top(), "");
        }

        //function prints codes of the elements
        static void printcodes(MHnode* root, string str) {

            if (!root) {
                return;
            }

            if (root->data != '$') {
                cout << root->data << ": " << str << "\n";
            }

            printcodes(root->left, str+"0");
            printcodes(root->right, str+"1");
        }

        //function stores the binary tree encode of the elements
        static void storecodes(MHnode* root, string str) {

            if (root == NULL) {
                return;
            }

            if (root->data != '$') {
                Huffmancoding::codes[root->data] = str;
            }

            storecodes(root->left, str+"0");
            storecodes(root->right, str+"1");
        }

};

map<char, string> Huffmancoding::codes = {}; //Initialise the map for the
codes of the characters
```

```
int main() {

    map<char, int> dataset; //Frequency table that is represented as a map.


    int size; //Size of frequency table (no. of rows)
    cout << "Enter size of frequency table: " << "\n";
    cin >> size;

    for (int i= 0; i < size; i++) {
        char temp;
        int temp2;
        cout << "Enter name and frequency of element" << i+1 << "\n";
        cin >> temp >> temp2;

        dataset.insert({temp, temp2});

    }

    /*
    char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    */

    // binary tree formation of the given data
    Huffmancoding::Codes(dataset, size);

    cout << "Original Frequency Table: " << "\n";

    //printing the original frequency table. auto is a variable type
    //that can take any datatype

    for (auto it3 = dataset.begin(); it3 != dataset.end(); it3++) {
        cout << it3->first << ": " << it3->second << "\n";
    }

    cout << "Encoded Frequency Table: " << "\n";

    for (auto it2 = Huffmancoding::codes.begin(); it2 !=
Huffmancoding::codes.end(); it2++) {
        cout << it2->first << ": " << it2->second << "\n";
    }

    return 0;
}
```