Indian Institute of Technology

Indore

Data Compression

Project Report

Department of Computer Science and Engineering

Submitted by –

Ashutosh Bang (160001011)

Shashank Giri (160001054)

Under the Guidance of **Dr. Kapil Ahuja**

Index

Introduction

Data compression involves encoding the given information and representing them using fewer bits than the original representation. A simple characterization of data compression is that it involves transforming a string of characters in some representation (such as ASCII) into a new string (of bits, for example) which contains the same information but whose length is as small as possible. It is useful because it reduces the resources required to store and transmit data. As a trade-off for cheaper and faster transmission of data, computational resources are consumed in the compression and the decompression process.

Data compression paradigm involves trade-off among many factors, like Compression speed, Degree of compression, Decompression speed, and the computational resources required for compressing and decompressing the data.

Compression can be of 2 kinds, lossy or lossless. Lossless compression reduces bits by identifying and eliminating redundancy in the given file. No information is lost in lossless compression. Lossy compression, on the other hand, reduces bits by removing unnecessary or less important information.

In this project we have focussed on Text Compression ,which is a lossless data compression,as one would never prefer to compress a text on the expense of modifying the original text.

Objectives –

1. To study and analyze the HUFFMAN data compression algorithms.
2. Compression and decompression of a text file using HUFFMAN algorithm.
3. Attempt to improve the compression speed in the HUFFMAN algorithm, and discussing future aspects.

Motivation –

Data compression finds its applications everywhere. It is widely used in backup utilities, spreadsheet applications, and database management systems. It also eases data transfer and storage.

Lossless data compression finds its use mostly in the text compression . There are major three lossless data compression LZW , LZSS, and HUFFMAN CODING. HUFFMAN CODING is the latest one among these and mostly used.

Thus we tried to implement this algorithm in $O(n \log n)$ rather than naive $O(n^2)$.

Basic Idea Behind The Huffman Coding

Here we are originally scanning the original file and updating frequency array which stores frequency corresponding to each input character. After the whole input text file is scanned we are using this frequency array to build something called as HUFFMAN tree where each leaf node stores one of the characters present in the input file. The edges of the tree have a label namely '0' and '1' . Each input character is encoded with the labels of the paths used to reach form the root to the leaf containing these character. Here the most frequent letter is placed closest to the root. So the code given to this character is as small as possible.

Here one point should be also taken care of that no two characters should have common prefix of codes otherwise it would be impossible to guess while decompressing that which character to place in the decompressed file.

Algorithm Analysis

Algorithm name: HUFFMAN

Pseudocode:(Naive)

Compression:

```
Algorithm  Huffman(C)
1: n := |C|;
2: Q := C;
3: for i := 1 to n − 1 do
4: allocate a new node z
5: z.lef t := x := Extract-Min(Q);{Searching linearly}
6: z.right := y := Extract-Min(Q);{Searching linearly}
7: z.freq := x.freq + y.freq;
8: Insert(Q, z);
9: end for
10: return Extract-Min(Q); {return the root of the tree}
```

Decompression:

Tree Building using tries.

```
---------------------------------------------------------------------
void build_tree(char alpha,string bit_rep){
        struct node *temp=root;
        for(unsigned i=0;i<bit_rep.size();i++){
                if(bit_rep[i]=='0'){
                        if(temp->left==NULL)
                        temp->left=new node('#');
                        temp=temp->left;
                }
                else {
```

```
                    if(temp->right==NULL)
                    temp->right=new node('#');
                    temp=temp->right;
                }

        }
        temp->alpha=alpha;

}
```

After building the tree we are just reading the decoded.bin file bit by bit and while reading the bits we are also scanning the HUFFMAN Tree and as we reach the leaf we print the character present in the leaf to the output decoded file.

Complexity Analysis:

A simple implementation on the above algorithm takes $O(n^2)$ time for compression and $O(n \log n)$ time for decompression.

Justification –

1) In the compression algorithm, when building the huffman tree we are scanning the min heap vector twice to find the minimum two elements present and then removing these two and inserting a node having weight equals sum of these two minimum elements.
This will take $O(n^2)$ time where n is the number of elements present in the min heap initially.

2) In the decompression algorithm,we are using **trie data structure** to build the Huffman tree .We are scanning the frequency file and then using tries to make the Huffman coding tree for decoding . Then we are reading the decoded file and using the tree to obtain the original character for the decoded character. Trie takes O(nlogn) time for building the tree where n is the number of bits written in the frequency file.

Factors affecting speed and compression ratio

HUFFMAN algorithm

The speed of compression as well as the size of compressed file depends a lot on input file and the frequency table built up, stored and the way it is accessed. As the number of keys in the frequency table increases, a good idea would be to use tries to speed up access and storage. At the same time, when the number of keys increase decompression takes more time to access the values.

Note: One of the issues faced in compression and decompression when the file size is very large is that of the memory constraints (RAM) of the machine so many a times the file is loaded in chunks in the main memory and hence compressing and decompressing takes more time.

Optimisations

The naïve implementation of the HUFFMAN algorithm takes $O(n^2)$ time to compress a given text file, as discussed above. However, it can be optimized and can be done in O(n log n) time, using priority queue data structure.

Every node of the trie will be a structure containing 3 fields –

char symbol
node* left
node* right

Pseudocode –(Efficient)

```
Algorithm  Huffman(C)
1: n := |C|;
2: Q := C;
3: for i := 1 to n − 1 do
4: allocate a new node z
5: z.lef t := x := Extract-Min(Q);{Using MINHEAP}
6: z.right := y := Extract-Min(Q);{Using MINHEAP}
7: z.freq := x.freq + y.freq;
8: Insert(Q, z);
9: end for
10: return Extract-Min(Q); {return the root of the tree}
```

Real C++ code

```
void build_tree(){

priority_queue<struct node*,vector<struct node*>,compare> priority;  for(int
i=0;i<MAX_NUM;i++){

if(frequency[i]>0){

                priority.push(new node((char)i,frequency[i]));

        }}

    while(priority.size()!=1){

struct node *node1,*node2;

        node1=priority.top(); priority.pop();
```

```
            node2=priority.top(); priority.pop();
            struct node *new_node=new node('#',node1->weight+node2->weight);
            new_node->left=node1;new_node->right=node2;
            priority.push(new_node);
        }}
```

Complexity Analysis –

Compression-

We now first scan the uncompressed file and updates the frequency of each character present in the original file. Which takes O(number of bytes present in the input file).

The above implementation takes O(n logn) time to build the priority queue of the frequency of each character present in the original file . Once the priority queue is built we are just popping two least frequent characters and pushing a node with weight equal the sum of weights of these two nodes. After this we are building the Huffman tree depending on the most frequent character present at the nearest to the root and as the frequency of the characters decreases the depth of leaves corresponding to that character increases.

Decompression-

Decompression is same as the decompression described in the naive approach using **trie data structure** we build the Huffman tree and then read the decoded file bit by bit and traverse the Huffman tree correspondingly and as a leaf is encountered we flush the character present at that leaf to the decoded file. The

trie take O(n log n) time to build and after that O(number of bytes writtten in the decoded file) additional time is taken to read the decoded file and then writing the decompressed file.

-Tree building using the tries code is provided in the naive section.

- writing to the decompressed file

Code(C++)

For writing to the decompressed file after building the Huffman tree in decode process.

```
void read_decoded_file(string input,string output){
	FILE *fr=fopen(input.c_str(),"rb");
	FILE *fout=fopen(output.c_str(),"w");
	char c;struct node *temp=root;
	while((c=fgetc(fr))!=EOF){
		for(int i=6;i>=0;i--){
			int x=((c>>i)&1);

			if(x){
				temp=temp->right;
				if(temp->alpha!='#'){
					fprintf(fout,"%c",temp->alpha);
					temp=root;
				}
```

```
}else{temp=temp->left;
        if(temp->alpha!='#'){
                fprintf(fout,"%c",temp->alpha);
                temp=root;
        }}}}}
```

Implementation and Results

The naive HUFFMAN algorithm was implemented in C++, and the efficient HUFFMAN algorithm was also implemented in C++. Text files with varying sizes were compressed using the algorithm.

Data compression ratio = Uncompressed File / Compressed File

The results of the naive algorithm were as follows-

Name of file	Original Size	Time taken to compress	Compressed file size	Compression Ratio	Time taken to decompress
small.txt	15.3 KB	0.019489 seconds	9.6 KB	1.59	0.000922 seconds
medium.txt	95.1 KB	0.049676 seconds	58.0 KB	1.637	0.005016 seconds
large.txt	166.2 KB	0.062429 seconds	104.2 KB	1.593	0.010246 seconds

The results of the **efficient** algorithm were as follows-

Name of file	Original Size	Time taken to compress	Compressed file size	Compression Ratio	Time taken to decompress
small.txt	15.3 KB	0.003655 seconds	9.6KB	1.59	0.000922 seconds
medium.txt	95.1 KB	0.019118 seconds	58 KB	1.637	0.005016 seconds

large.txt	166.2 KB	0.034337 seconds	104.2 KB	1.593	0.010246 seconds

As a result, we can see a significant improvement in compression times, by using an efficient implementation of the HUFFMAN algorithm.

Future Work

HUFFMAN can be further improved by using various special techniques.

ADAPTIVE HUFFMAN is a modern data compression algorithm used in live video buffering. The benefit is that in this algorithm, the source can be encoded in real time.

DEFLATE is a modern data compression algorithm, used in the .zip file format. It improves Huffman by introducing LZSS codes. It is also based on the standard bit manipulation to denote the encoded and not coded streams.

Compression is achieved through two steps:

1) The matching and replacement of duplicate strings with pointers.
2) Replacing symbols with new, weighted symbols based on frequency of use.

Machine learning can also help in compressing data. A system that predicts the posterior probabilities of a sequence given its entire history can be used for optimal data compression (by using arithmetic coding on the output distribution) while an optimal compressor can be used for prediction (by finding the symbol that compresses best, given the previous history).

In arithmetic coding, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total.

Data compression is an open area of research.
After the deflate algorithm there have been many new discoveries - the most recent one is GOOGLE GUETZLI. The GOOGLE GUETZLI is a

JPEG encoder that aims for excellent compression density at high visual quality.

For Complete working code written by us please refer my github repository-
https://github.com/shashank98giri/ALgo_project

References –

1-https://github.com/gyaikhom/huffman

2- IEEE journal on a lossless data compression
http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=1715326