DELHI TECHNOLOGICAL UNIVERSITY

MIDTERM EVALUATION COMPONENT PROJECT REPORT

HUFFMAN CODING: IMPLEMENTATION AND APPLICATION

DATA STRUCTURES IT:201

Submitted To: Ms. Nidhi

Harshal Chowdhary

2k19/EC/071

harshalchowdhary_2k19ec071@dtu.ac.in

Priti Gangwar

2k19/EC/134

pritigangwar_2k19ec134@dtu.ac.in

TABLE OF CONTENTS

SERIAL NO.	TITLE
1.	ACKNOWLEDGEMENT
2.	OBJECTIVE
3.	INTRODUCTION
4.	THEORY
5.	METHODOLOGY
6.	IMPLEMENTATION RESULTS
7.	CONCLUSION AND FUTURE WORK
8.	REFERENCES

1. ACKNOWLEDGEMENT

We would like to express our sincere gratitude to several individuals and organizations for supporting us. First, we wish to express our sincere gratitude to our teacher, Ms. Nidhi, for her enthusiasm, patience, insightful comments, helpful information, practical advice and unceasing ideas that have helped us tremendously at all times in the project. Her immense knowledge, profound experience and professional expertise has enabled us to complete this work successfully.

We also wish to express our sincere thanks to the Delhi Technological University for accepting us into the undergraduate program.

Sincerely,

Harshal Chowdhary (2k19/EC/071)

Priti Gangwar (2k19/EC/134).

2. OBJECTIVE

The objective of this project is to understand the working of Huffman Coding theoretically and then implement it to create real world projects. Here, our aim will be to discuss data compression algorithms and their real-life applications, create a File Zipper website using Huffman coding to compress and decompress a text file using suitable programming languages such as JavaScript, HTML/CSS and Bootstrap.

3. INTRODUCTION

Some compression-related analysis within the areas of data theory and data compression algorithmic rule will be discussed in this section. A brief overview of data compression and lossless compression will be provided.

BASIC CONCEPTS OF DATA COMPRESSION

Data compression, also known as source coding in computer science and communication theory, is the art of encoding original data with a specific mechanism to fewer bits (or other information related units).

The ZIP file format, which is widely used in PCs, is a popular living example. It not only has compression functions, but it also has archive tools (Archiver) that can store multiple files in a single root file. Data compression communication works only when both the sender and receiver are aware of the encoding mechanism. This feature (data encryption during compression) is used by some compression algorithms to ensure that only authorized parties can obtain accurate data.

Because most real-world data contains a high level of statistical redundancy, data compression is possible. Redundancy can take many forms. Data redundancy exists in multimedia information as well. The following types of data redundancy exist:

1. **Spatial redundancy.** (Many of the pixels in an image's static architectural background, blue sky, and lawn are the same. If such an image is saved pixel by pixel, it will take up a lot of space.)

2. **Temporal redundancy.** (In television, animation images between adjacent frames most likely have the same background, with the only difference being the position of moving objects.)

3. **Structure redundancy**

4. **Knowledge redundancy**

5. **Information entropy redundancy.** (It is also known as encoding redundancy, which refers to the amount of entropy that data carries.)

As a result, one aspect of compression is the elimination of redundancy. According to economic theory, data compression can reduce the consumption of expensive resources such as hard disk space and bandwidth connections. However, compression consumes information processing resources, which can be costly.

Therefore, an effective compression mechanism must always make a tradeoff between compression capability, distortion, computational resources required, and other factors.

4. THEORY

CLASSIFICATION OF COMPRESSION TECHNIQUES

There are many methods to compress data. Different sets of data need different compression methods.

It can be classified in the following aspect:

1. Instant compression and non-instant compression

2. Data compression and file compression

3. Lossless compression and lossy compression

Since our project mainly deals with the third aspect, let us look into lossless compression in detail.

Lossless and Lossy Compression Techniques

- There are two types of data compression techniques: lossless and lossy.
- Lossless techniques allow for the exact reconstruction of the original document from compressed data, whereas lossy techniques do not.
- Lossless techniques include run-length, Huffman, and Lempel-Ziv, whereas JPEG and MPEG are lossy.
- Lossy compression techniques typically achieve higher compression rates than lossless compression techniques, but the latter are more accurate.
- Lempel-Ziv coding reads variable-sized input and outputs fixed-length bits, whereas Huffman coding does the inverse.
- Lossless techniques are divided into two types: static and adaptive.
- Before compression begins in a static scheme, such as Huffman coding, the data is scanned to obtain statistical information.
- Adaptive models, such as Lempel-Ziv, start with a statistical distribution of text symbols and modify it as each character or word is encoded.
- Adaptive schemes are more accurate, but static schemes require fewer computations and are faster.

- **In conclusion, some compression mechanisms are reversible, allowing the original data to be restored; this is referred to as lossless data compression. Other mechanisms, known as lossy data compression, allow for some data loss in order to achieve a higher compression rate.**

UNDERSTANDING HUFFMAN CODING

Huffman coding is a popular lossless algorithm used in entropy encoding. This algorithm's process is to assign VLC (variable-length code) to characters (using different arrangements of 0 or 1 to represent characters). The length of the code is determined by the frequency of the corresponding character; the most frequently occurring character has the shortest code, while the least frequently occurring character has the longest code.

The essence of the Huffman algorithm is the re-coding of an original information source based on statistical results of character occurrence frequency, rather than dealing with character repetition or substring repetition. The frequency of The procedure necessitates two steps (statistical counting and coding), which are both time-consuming and inefficient. In contrast, an adaptive (or dynamic) Huffman algorithm does not require statistical counting. It can dynamically adjust the Huffman tree during compression, significantly increasing the processing rate.

As a result, Huffman coding has a high productivity, quick operation speed, and a flexible implementation procedure.

Huffman coding is essentially an algorithm for representing data in a different representation in order to reduce information redundancy. The data is binary encoded.

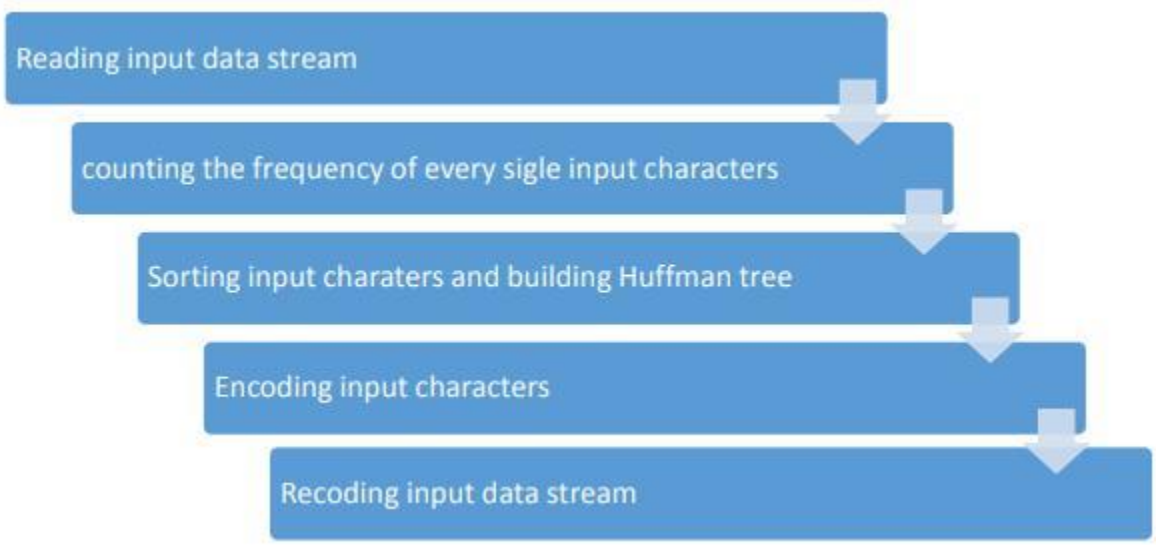
The idea is to assign variable-length codes to input characters, with the lengths of the assigned codes determined by the frequency of the corresponding characters. The most frequently occurring character is assigned the smallest code, while the least frequently occurring character is assigned the largest code.

The variable-length codes assigned to input characters are Prefix Codes, which means that the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of any other character's code. Huffman Coding ensures that there is no ambiguity when decoding the generated bitstream in this manner.

5. METHODOLOGY

ALGORITHM IMPLEMENTATION DESIGN

Huffman coding procedure can be illustrated as the following figure:



A. Reading Input Data Stream

```
window.onload = function () {
    decodeBtn = document.getElementById("decode");
    encodeBtn = document.getElementById("encode");
    uploadFile = document.getElementById("uploadfile")
    submitBtn = document.getElementById("submitbtn");
    step1 = document.getElementById("step1");
    step2 = document.getElementById("step2");
    step3 = document.getElementById("step3");
    codecObj = new Codec();

    submitBtn.onclick = function () {
        var uploadedFile = uploadFile.files[0];
        if (uploadedFile === undefined) {
            alert("No file uploaded.\nPlease upload a file and try aga
in");
            return;
```

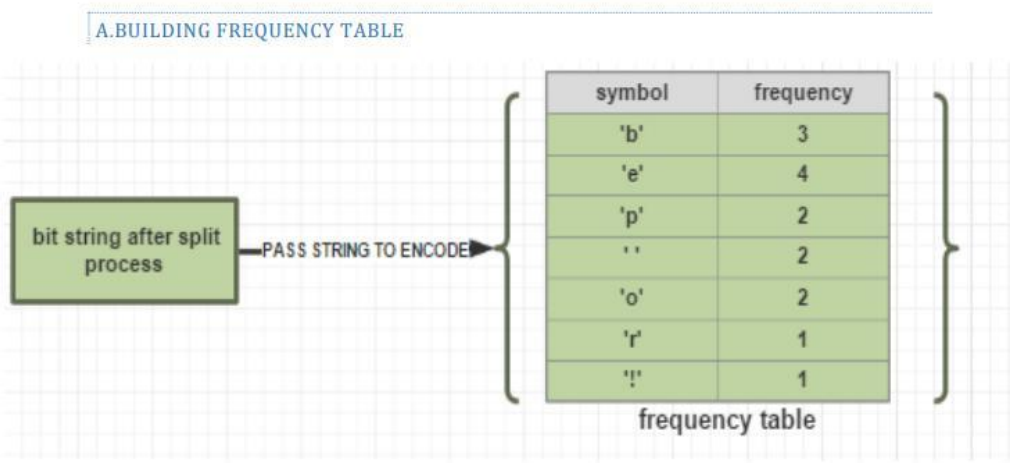
```
        }
        let nameSplit = uploadedFile.name.split('.');
        var extension = nameSplit[nameSplit.length -
        1].toLowerCase(); if (extension != "txt") {
            alert("Invalid file type (." + extension + ") \nPlease upl
oad a valid .txt file and try again");
            return;
        }
        document.getElementById("step1").style.display = "none";
        document.getElementById("step2").style.display = "inline-flex"
;
        document.getElementById("startagain").style.visibility = "visi
ble";
    }

    encodeBtn.onclick = function () {
        var uploadedFile = uploadFile.files[0];
        if (uploadedFile === undefined) {
            alert("No file uploaded.\nPlease upload a file and try aga
in");
            return;
        }
        console.log(uploadedFile.size);
        if(uploadedFile.size === 0){
            alert("You have uploaded an empty file!\nThe compressed fi
le might be larger in size than the uncompressed file (compression rat
io might be smaller than one).\nBetter compression ratios are achieved
for larger file sizes!");
        }
        else if(uploadedFile.size <= 350){
            alert("The uploaded file is very small in size (" + upload
edFile.size +" bytes) !\nThe compressed file might be larger in size t han
the uncompressed file (compression ratio might be smaller than one
).\nBetter compression ratios are achieved for larger file sizes!");
        }
        else if(uploadedFile.size < 1000){
```

```
            alert("The uploaded file is small in size (" + uploadedFil
e.size +" bytes) !\nThe compressed file's size might be larger than ex
pected (compression ratio might be small).\nBetter compression ratios
are achieved for larger file sizes!");
        }
        onclickChanges2("Compressing your file ...\n",
        "Compressed"); var fileReader = new FileReader();
        fileReader.onload = function (fileLoadedEvent)
            { let text = fileLoadedEvent.target.result;
            let [encodedString, outputMsg] = codecObj.encode(text);
            myDownloadFile(uploadedFile.name.split('.')[0] + "_compres
sed.txt", encodedString);
            ondownloadChanges(outputMsg);
        }
        fileReader.readAsText(uploadedFile, "UTF-8");
        document.getElementById("step2").style.display = "none";
        document.getElementById("step3").style.display = "inline-flex"
;
    }

    decodeBtn.onclick = function () {
        console.log("decode onclick");
        var uploadedFile = uploadFile.files[0];
        if (uploadedFile === undefined) {
            alert("No file uploaded.\nPlease upload a file and try aga
in!");
            return;
        }
        onclickChanges2("De-compressing your file ...\n", "De-Compress
ed");
        var fileReader = new FileReader();
        fileReader.onload = function (fileLoadedEvent)
            { let text = fileLoadedEvent.target.result;
            let [decodedString, outputMsg] = codecObj.decode(text);
            myDownloadFile(uploadedFile.name.split('.')[0] + "_decompr
essed.txt", decodedString);
            ondownloadChanges(outputMsg);
```

```
        }
        fileReader.readAsText(uploadedFile, "UTF-8");
        document.getElementById("step2").style.display = "none";
        document.getElementById("step3").style.display = "inline-flex"
;
    }
```

B. Counting the frequency of every single input characters

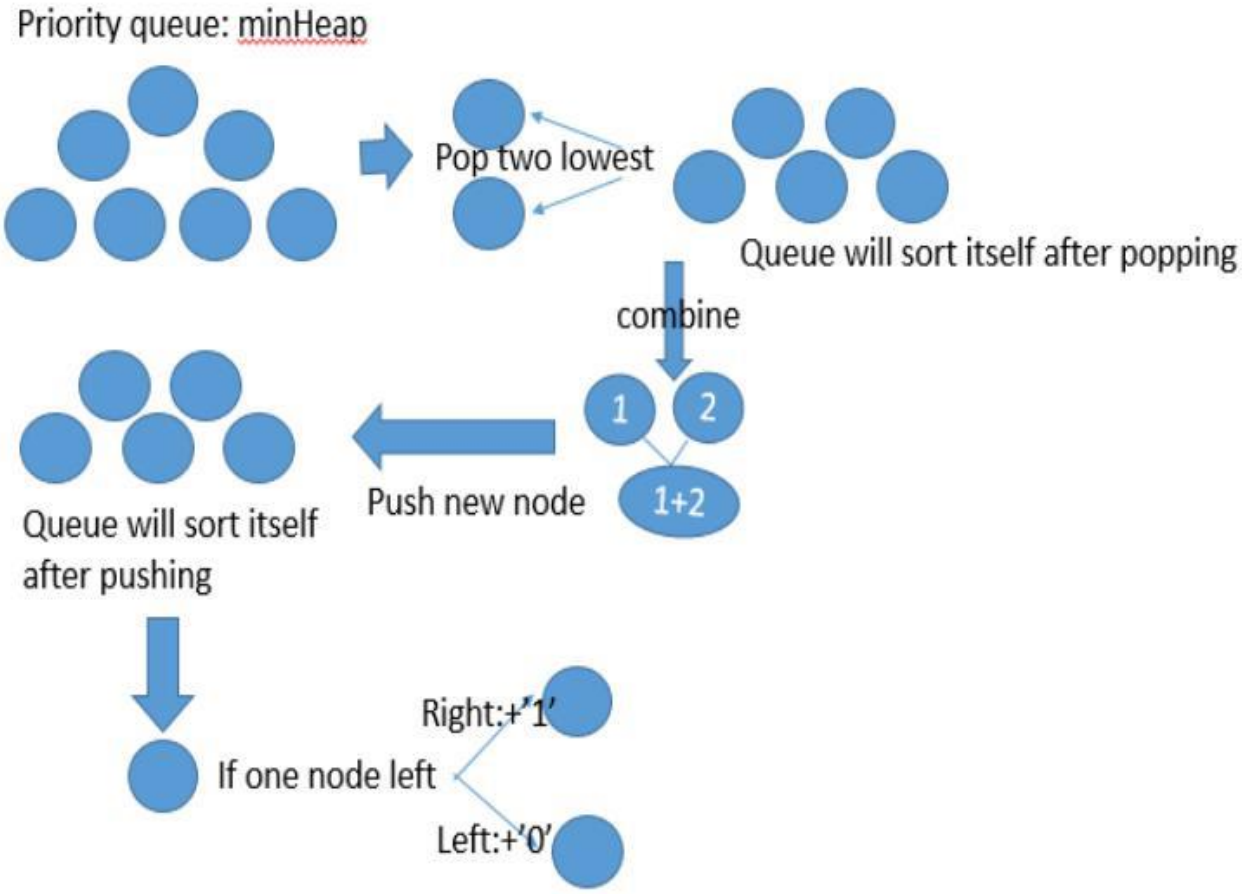
- Encoding is the most important step of the whole process. The details of implementation are divided into several parts and can be illustrated in following steps:



- This step is passing the split bit string to the first encoding procedure (building frequency table). The STL ordered map is used to store the frequency of symbols. Each symbol has its own appearance frequency. In the program, ‘for’ loop is used to get the frequency count for every single symbol.

C. Sorting Input characters and building a Huffman Tree

- After building the frequency table, the next step is building the Huffman tree which is the crucial part for encoding and decoding. A simple visual of this process is given by following figure:

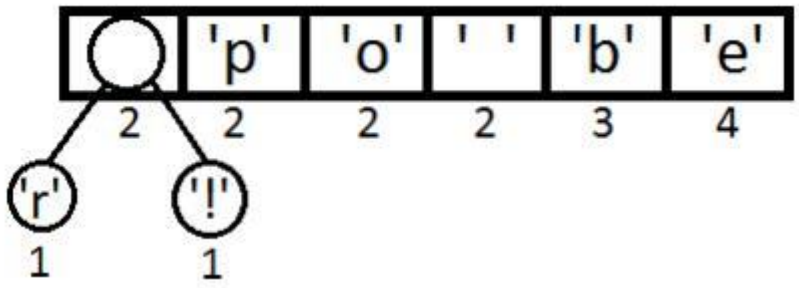


  - Firstly, the information of frequency table is passed and stored in minHeap(priority queue). The minHeap is sorted according to priority (for Huffman coding is less one at front) and stored in an array. So following array is the sorted queue according to frequency table in the A part and priority:

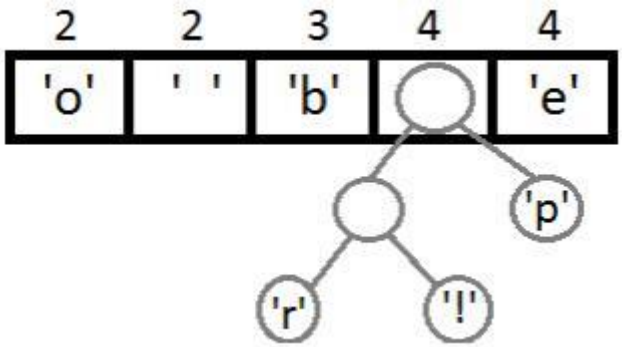
'r'	'!'	'p'	'o'	' '	'b'	'e'

  - Secondly, the minHeap is needed to be transformed to binary tree. Every two elements of the beginning (or two with lowest frequencies) are pop off to create one binary tree (first one is left child and second one is right child). Then, adding the frequencies of these two

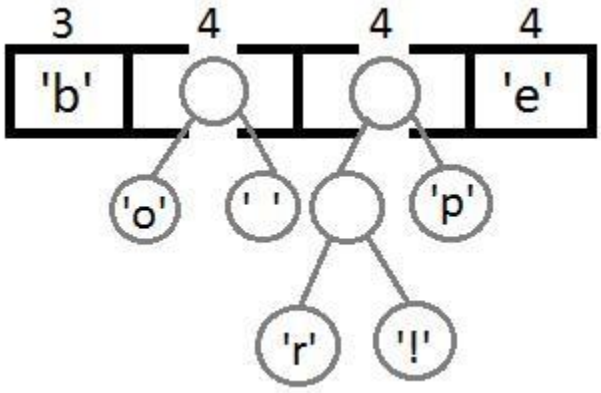
elements and pushing back to minHeap. So the data graph can be showed as following:

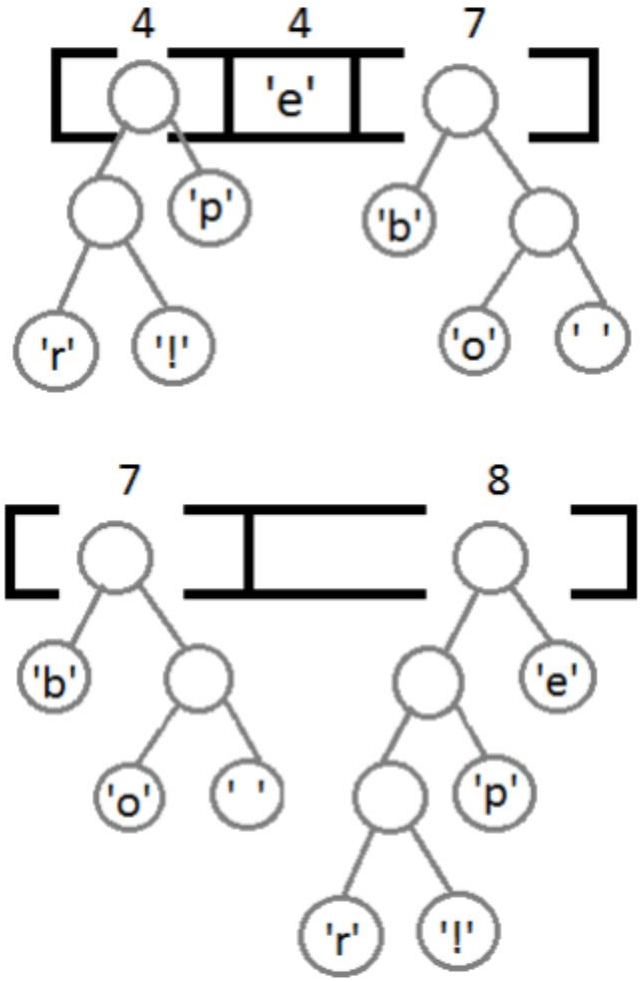


- Similarly, popping off another first two elements to form a node with frequency equals four. Then it is pushed back to minHeap again:

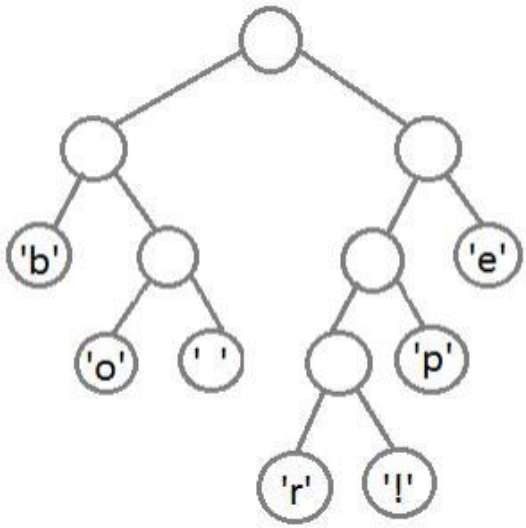


- Continuing (it can be seen that this is a process of build binary tree from bottom to top):



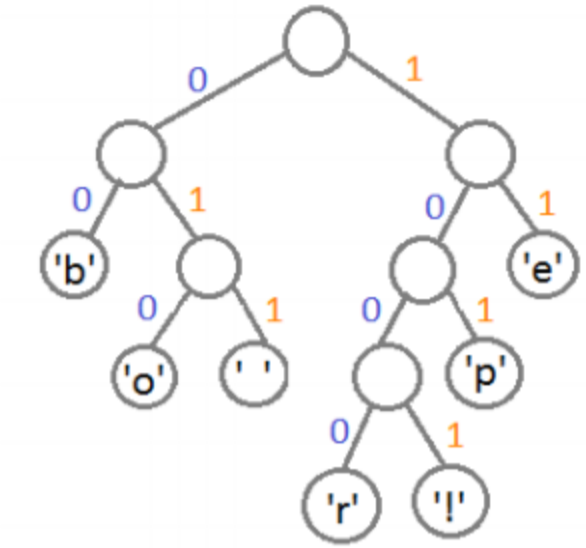


- The final binary tree formed:



- The **left children** can be coded as **'0'** and the **right children** can be coded as **'1'.** Then we can traverse the Huffman tree to get the encoding string of each symbol. For instance, encoding string of 'b' is 00 and

encoding string of ‘p’ is 101. Also, we can see that the more frequently a symbol appears, the higher level it will be.



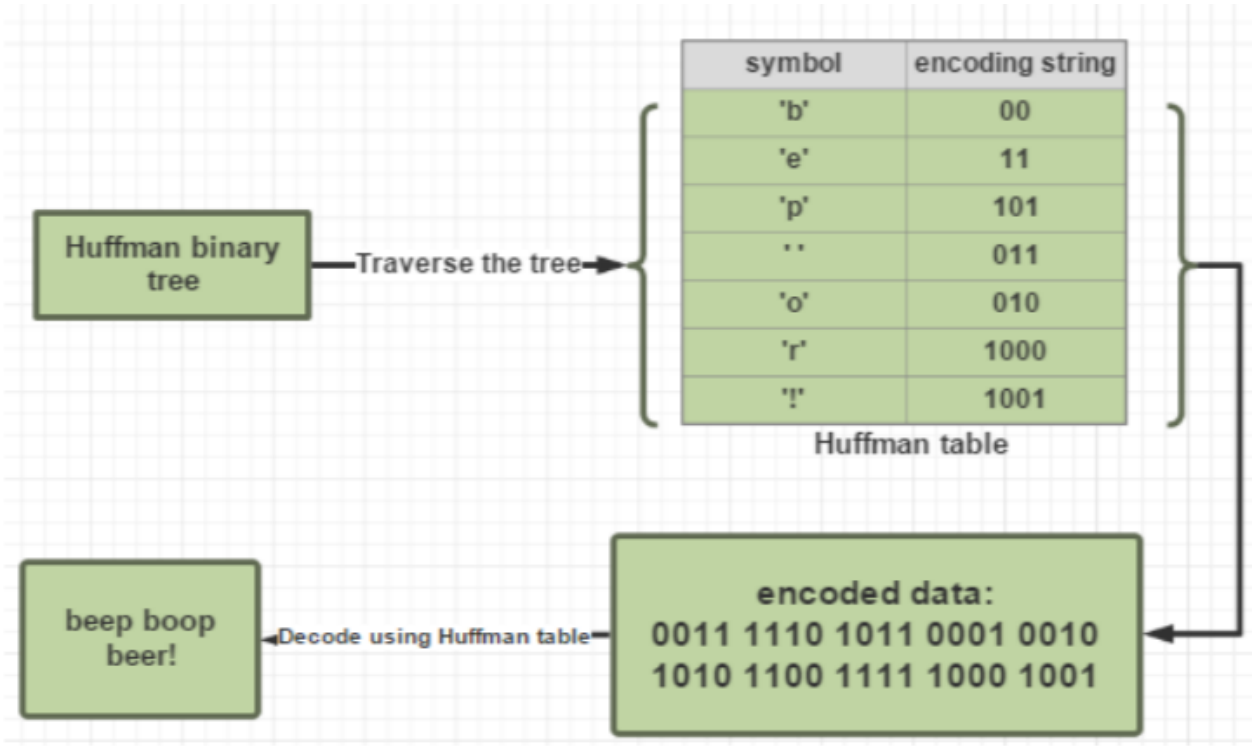

```
class MinHeap {
    constructor() {
        this.heap_array = [];
    }
    size() {
        return this.heap_array.length;
    }
    empty() {
        return (this.size() === 0);
    }
    push(value) {
        this.heap_array.push(value);
        this.up_heapify();
    }
    up_heapify() {
        var current_index = this.size() -
        1; while (current_index > 0) {
            var current_element = this.heap_array[current_index]; var
            parent_index = Math.trunc((current_index - 1) / 2); var
            parent_element = this.heap_array[parent_index];
```

```
            if (parent_element[0] < current_element[0])
                { break;
            }
            else {
                this.heap_array[parent_index] = current_element;
                this.heap_array[current_index] = parent_element;
                current_index = parent_index;
            }
        }
    }
    top() {
        return this.heap_array[0];
    }
    pop() {
        if (this.empty() == false) {
            var last_index = this.size() - 1;
            this.heap_array[0] = this.heap_array[last_index];
            this.heap_array.pop();
            this.down_heapify();
        }
    }
    down_heapify() {
        var current_index = 0;
        var current_element = this.heap_array[0];
        while (current_index < this.size()) {
            var child_index1 = (current_index * 2) + 1;
            var child_index2 = (current_index * 2) + 2;
            if (child_index1 >= this.size() && child_index2 >= this.si
ze()) {
                break;
            }
            else if (child_index2 >= this.size()) {
                let child_element1 = this.heap_array[child_index1];
                if (current_element[0] < child_element1[0]) {
                    break;
                }
```

```
                else {
                    this.heap_array[child_index1] = current_element;
                    this.heap_array[current_index] = child_element1;
                    current_index = child_index1;
                }
            }
            else {
                var child_element1 = this.heap_array[child_index1];
                var child_element2 = this.heap_array[child_index2];
                if (current_element[0] < child_element1[0] && current_
element[0] < child_element2[0]) {
                    break;
                }
                else {
                    if (child_element1[0] < child_element2[0]) {
                        this.heap_array[child_index1] = current_elemen
t;
                        this.heap_array[current_index] = child_element
1;
                        current_index = child_index1;
                    }
                    else {
                        this.heap_array[child_index2] = current_elemen
t;
                        this.heap_array[current_index] = child_element
2;
                        current_index = child_index2;
                    }
                }
            }
        }
    }
}
```

D. Encoding the input characters

- Next step to building a Huffman table is encoding and decoding. The Huffman tree was created from the last past and will be used to generate the Huffman table. All the codes will be pushed into an ordered map to record each symbol's encoding string. Also, based on this table, an encoding string of the whole data can be generated and encoded data can be decoded to original data. The whole process can be illustrated in following figure:




```
encode(data) {
        this.heap = new MinHeap();

        var mp = new Map();
        for (let i = 0; i < data.length; i++) {
            if (mp.has(data[i])) {
                let foo = mp.get(data[i]);
                mp.set(data[i], foo + 1);
            }
            else {
                mp.set(data[i], 1);
            }
        }
```

```
        if (mp.size === 0) {
            let final_string = "zer#";

            let output_message = "Compression complete and file will b
e downloaded automatically." + '\n' + "Compression Ratio : " + (data.l
ength / final_string.length).toPrecision(6);
            return [final_string, output_message];
        }

        if (mp.size === 1) {
            let key, value;
            for (let [k, v] of mp) {
                key = k;
                value = v;
            }
            let final_string = "one" + '#' + key + '#' + value.toStrin
g();
            let output_message = "Compression complete and file will b
e downloaded automatically." + '\n' + "Compression Ratio : " + (data.l
ength / final_string.length).toPrecision(6);
            return [final_string, output_message];
        }
        for (let [key, value] of mp) {
            this.heap.push([value, key]);
        }
        while (this.heap.size() >= 2) {
            let min_node1 = this.heap.top();
            this.heap.pop();
            let min_node2 = this.heap.top();
            this.heap.pop();
            this.heap.push([min_node1[0] + min_node2[0], [min_node1, m
in_node2]]);
        }
        var huffman_tree = this.heap.top();
        this.heap.pop();
        this.codes = {};
        this.getCodes(huffman_tree, "");
```

```
        /// convert data into coded
        data let binary_string = "";
        for (let i = 0; i < data.length; i++) {
            binary_string += this.codes[data[i]];
        }
        let padding_length = (8 - (binary_string.length % 8)) %
        8; for (let i = 0; i < padding_length; i++) {
            binary_string += '0';
        }
        let encoded_data = "";
        for (let i = 0; i < binary_string.length;) {
            let curr_num = 0;
            for (let j = 0; j < 8; j++, i++)
                { curr_num *= 2;
                curr_num += binary_string[i] - '0';
            }
            encoded_data += String.fromCharCode(curr_num);
        }
        let tree_string = this.make_string(huffman_tree);
        let ts_length = tree_string.length;
        let final_string = ts_length.toString() + '#' + padding_length
.toString() + '#' + tree_string + encoded_data;
        let output_message = "Compression complete and file will be do
wnloaded automatically." + '\n' + "Compression Ratio : " + (data.lengt
h / final_string.length).toPrecision(6);
        return [final_string, output_message];
    }
```

E.Decoding the (encoded) input stream

- In the process of building Huffman table, depth-first search algorithm and recursive function are used to note down the encoding string for each symbol. Decoding is using encoded data and traversing Huffman tree to get original input string.

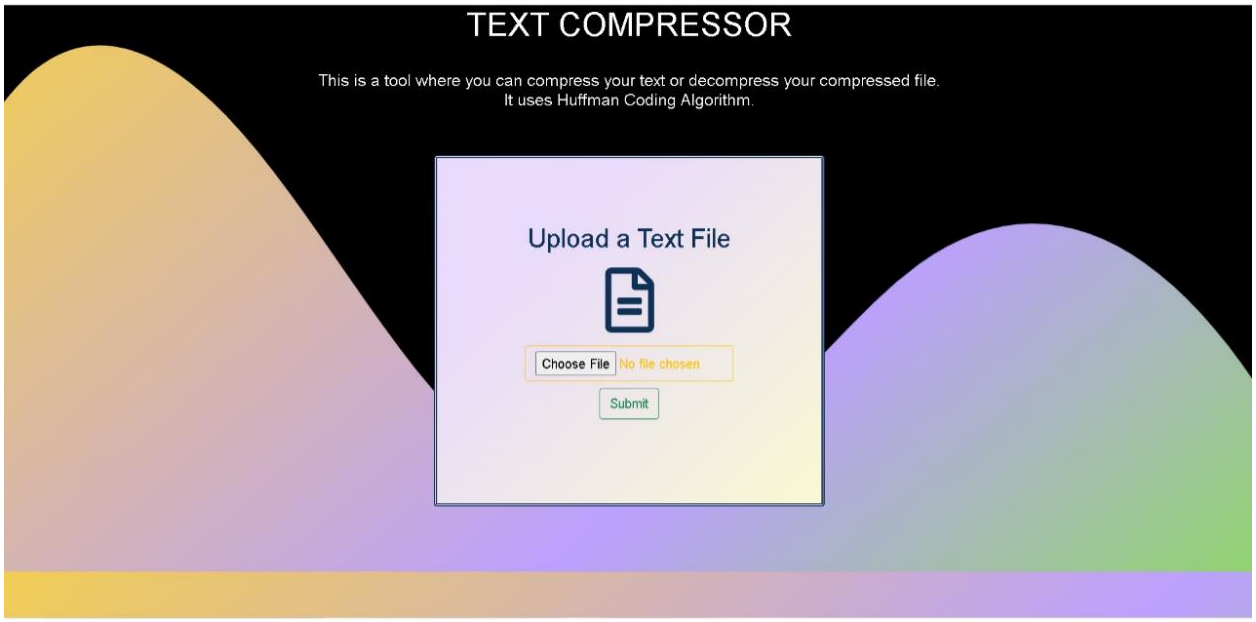
```
   decode(data) {
        let k = 0;
        let temp = "";
        while (k < data.length && data[k] != '#')
            { temp += data[k];
            k++;
        }
        if (k == data.length){
            alert("Invalid File! Please submit a valid De-Compressed f
ile");
            location.reload();
            return;
        }
        if (temp === "zer") {
            let decoded_data = "";
            let output_message = "De-Compression complete and file wil
l be downloaded automatically.";
            return [decoded_data, output_message];
        }
        if (temp === "one") {
            data = data.slice(k + 1);
            k = 0;
            temp = "";
            while (data[k] != '#') {
                temp += data[k];
                k++;
            }
            let one_char = temp;
            data = data.slice(k + 1);
            let str_len = parseInt(data);
            let decoded_data = "";
```

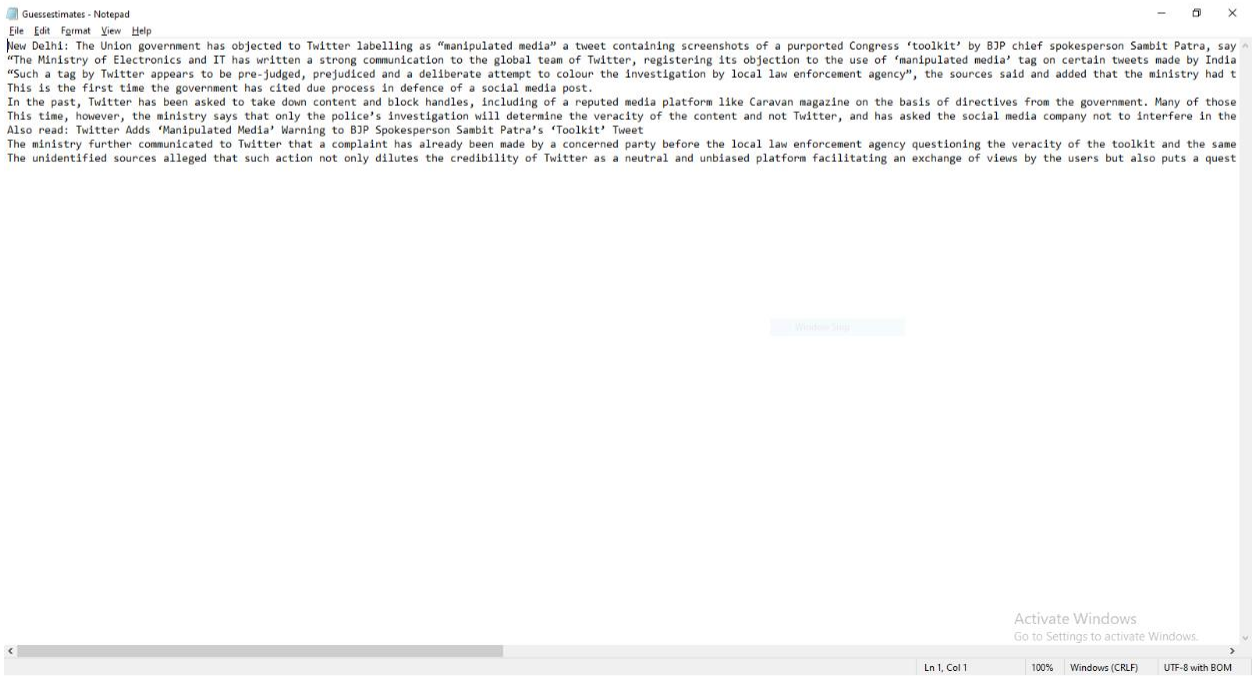
```
            for (let i = 0; i < str_len; i++) {
                decoded_data += one_char;
            }
            let output_message = "De-Compression complete and file wil
l be downloaded automatically.";
            return [decoded_data, output_message];

        }
        data = data.slice(k + 1);
        let ts_length = parseInt(temp);
        k = 0;
        temp = "";
        while (data[k] != '#') {
            temp += data[k];
            k++;
        }
        data = data.slice(k + 1);
        let padding_length =
        parseInt(temp); temp = "";
        for (k = 0; k < ts_length; k++) {
            temp += data[k];
        }
        data = data.slice(k);
        let tree_string = temp;
        temp = "";
        for (k = 0; k < data.length; k++) {
            temp += data[k];
        }
        let encoded_data = temp;
        this.index = 0;
        var huffman_tree = this.make_tree(tree_string);

        let binary_string = "";
        for (let i = 0; i < encoded_data.length; i++) {
            let  curr_num  =  encoded_data.charCodeAt(i);
            let curr_binary = "";
            for (let j = 7; j >= 0; j--) {
```

```
                let foo = curr_num >> j;
                curr_binary = curr_binary + (foo & 1);
            }
            binary_string += curr_binary;
        }
        binary_string = binary_string.slice(0, -padding_length);
        let decoded_data = "";
        let node = huffman_tree;
        for (let i = 0; i < binary_string.length; i++)
            { if (binary_string[i] === '1') {
                node = node[1];
            }
            else {
                node = node[0];
            }

            if (typeof (node[0]) === "string")
                { decoded_data += node[0];
                node = huffman_tree;
            }
        }
        let output_message = "De-Compression complete and file will
be downloaded automatically.";
        return [decoded_data, output_message];
    }
```

6. IMPLEMENTATION RESULTS

I. Uploading the Original Text File (Input in .txt format)

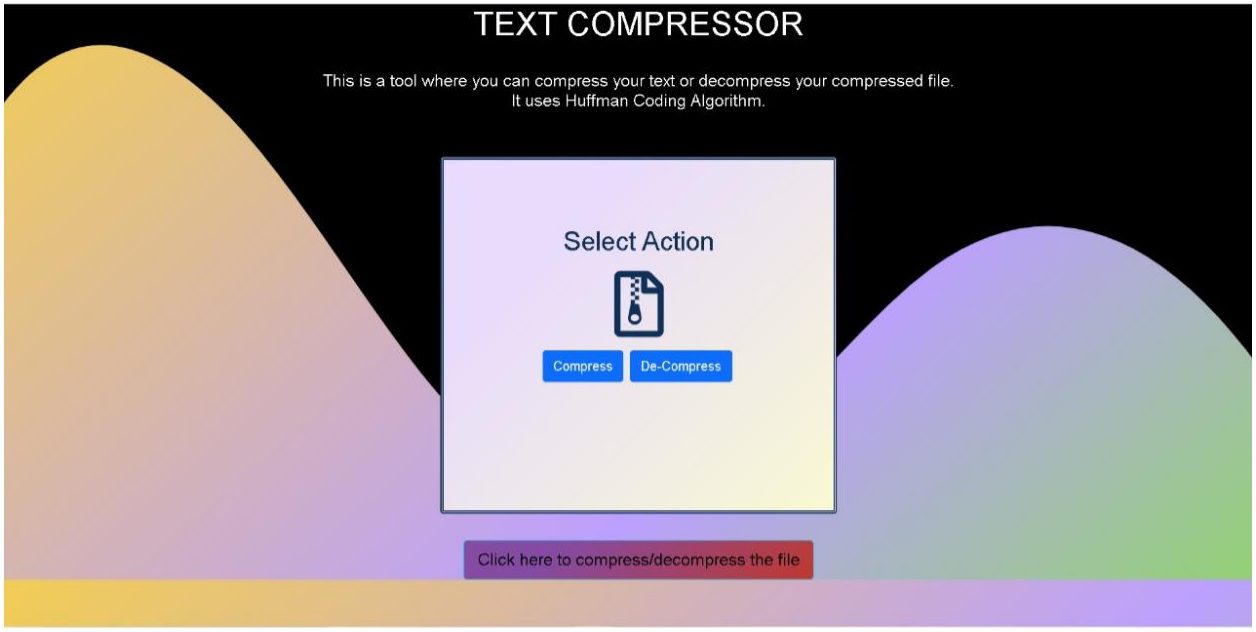


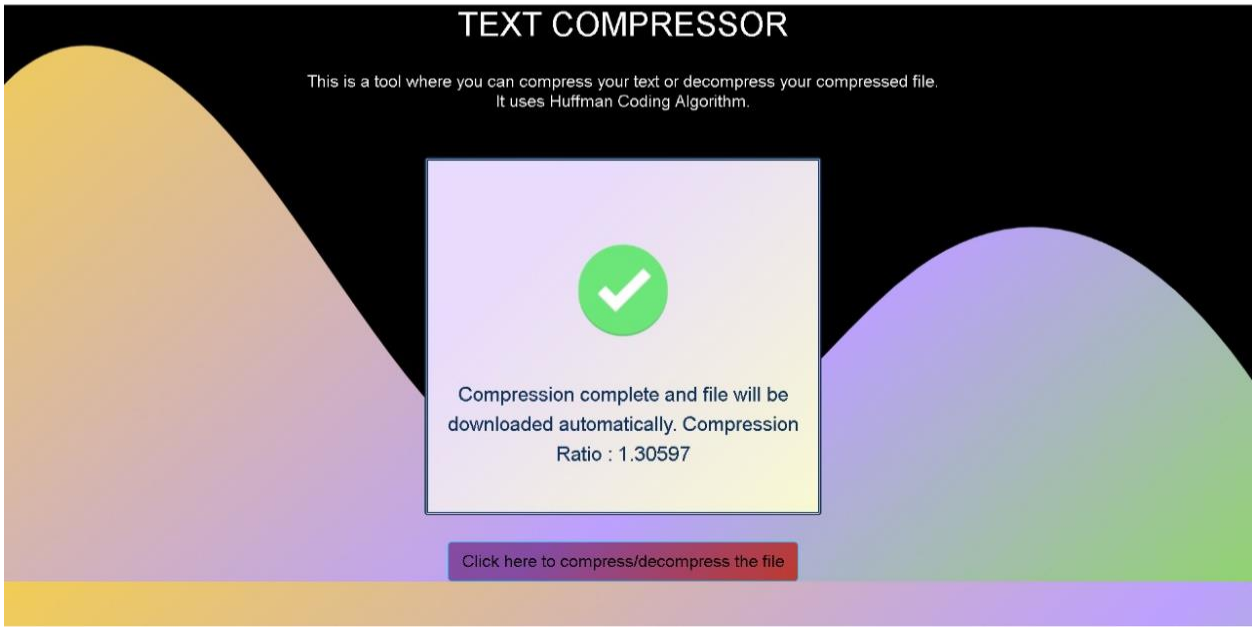


Link to the text fle:

https://drive.google.com/fle/d/1uaAPVilLEJySMK9eURUCgg1QlOZVIZC9/view?usp=sharing

II. Compressing the Text File (Huffman Encoding)

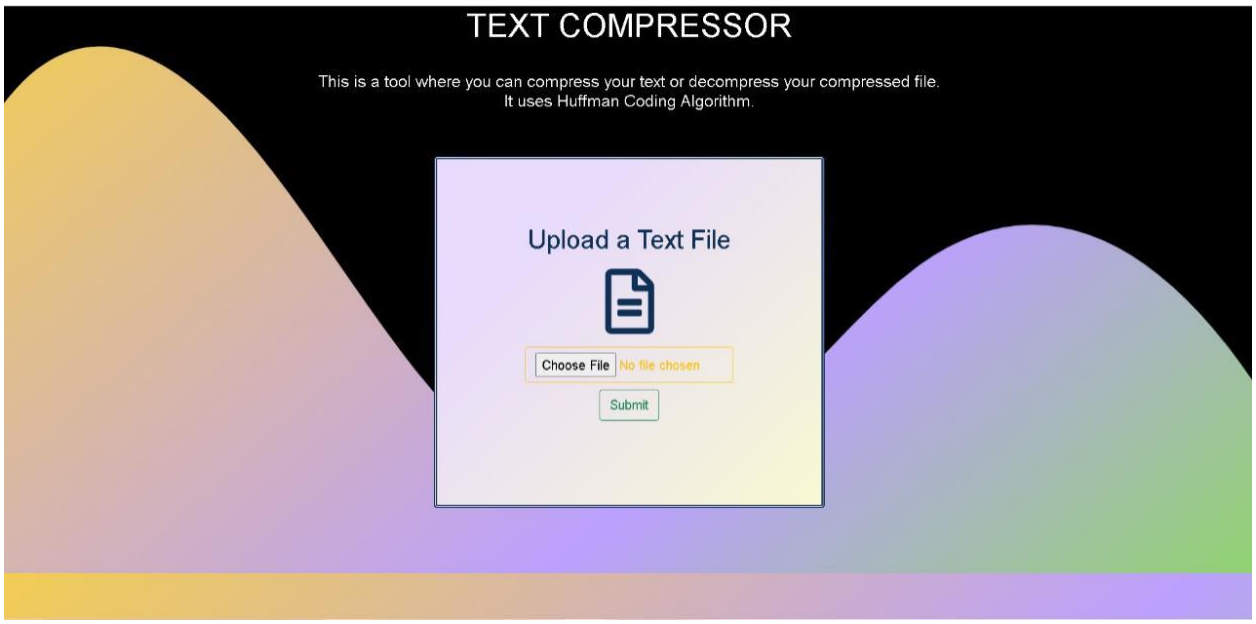


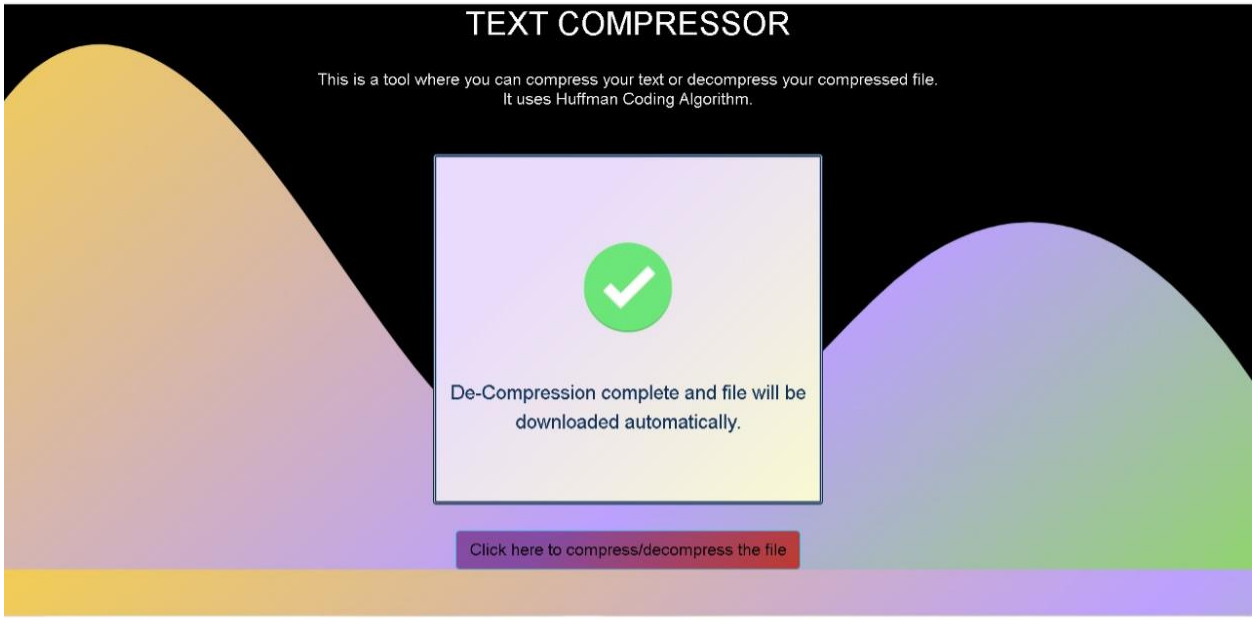


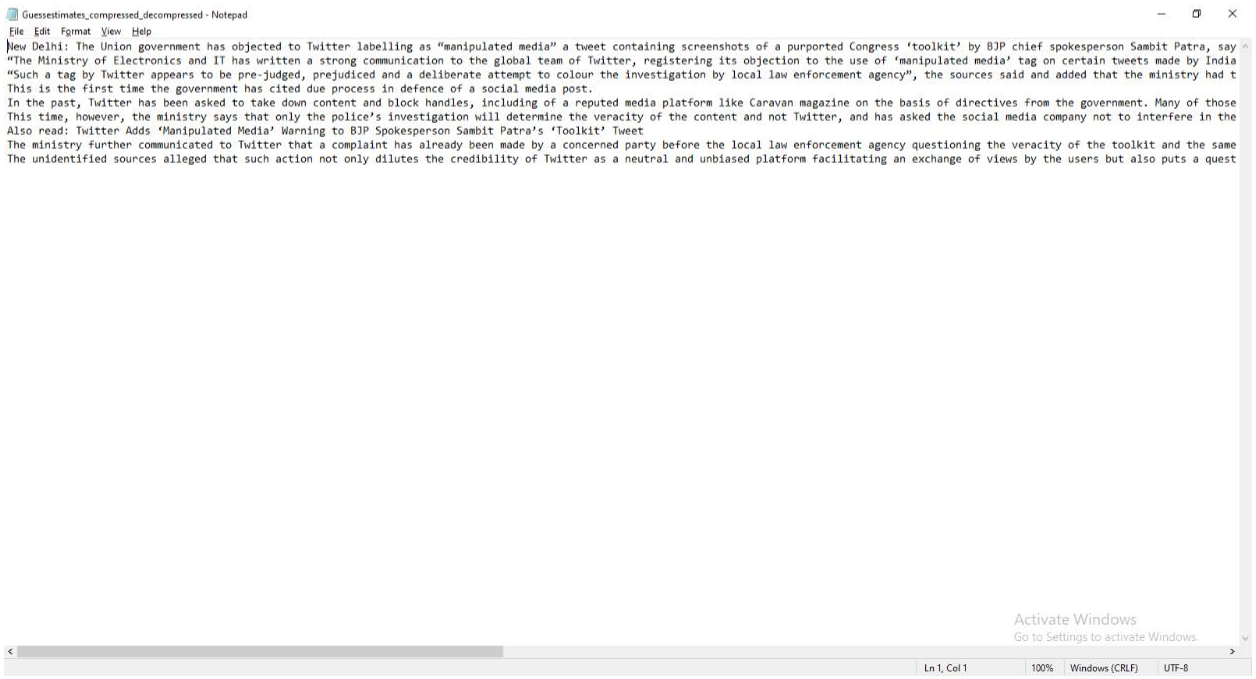
Link to the text fle:

https://drive.google.com/fle/d/1EsSyGqXCZFPVhKR4nwtewPXtGAeoasC_/view?usp=sharing

III. Decompressing the Compressed Text File (Input in .txt format)







Link to the text fle:

https://drive.google.com/fle/d/1gvH5DxagfsCL2e8DzFZrkoiShbTIyXBM/view?usp=sharing

7. CONCLUSION AND FUTURE WORK

In Conclusion, in this project we have seen the topic of data compression in terms of classification, theory and application. The lossless data compression is mainly discussed in this report for further implementations. Huffman coding was implemented for investigating how algorithms can be used to improve compression ratio. The implementation is based on several methods with respect to information search, concepts generation and research analysis. The objectives of this project are mostly completed. A user-defined bit length Huffman coding program was designed to get a better understanding of data compression and the mathematical working of algorithms.

8. REFERENCES

- https://www.massey.ac.nz/~albarcza/ResearchFiles/HaominLiu2015.pdf
- http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf
- https://arxiv.org/ftp/arxiv/papers/1109/1109.0216.pdf
- http://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/compression.pdf
- http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1126/handouts/220%20Huffman%20Encoding.pdf