University of Science and Technology in Zewail City

CIE 425

Huffman Project report

2022

Noureldin Mohamed

s-noureldin.hamedo@zewailcity.edu.eg

Contents

1 Task 1:Design a Huffman code for an English Language character set.

1.1 Huffman Class

This is a class that encapsulates the data for each character.
The attributes were kept public for simplicity to avoid the continous use of setters and getters.

1.1.1 Attributes

- **string c**: The character
- **string h**: Its huffman code for the character
- **double p**: The frequency of the character

1.1.2 Constructors

A default constructor was created where the huffman code was initiated as an empty string, this step was not necessary; however it was done for debugging issues.

The default constructor should be created or an error would pop up when an object is created without setting the attributes could be kept empty and nothing would be affected.

A non default constructor is responsible for creating objects with the string and frequency passed:

```
Huffman(string c, double f)
{
        this->p = f; //filling the frequency attribute by the passed value
        this->c = c; // filling the character attribute by the passed value
};
```

1.2 Functions

1.2.1 Main Function

Step 1: The first function is the main function which is the function where the frequency array and the character arrays where declared. It is the maistro that collects all the tasks together.

```
Huffman(string c, double f)
---------------------Step 1.Create the arrays.---------------

string c_array[26] = {"A","B","C","D","E","F","G","H","I","J" ,
"K","L","M" ,"N","O","P" ,"Q","R","S" ,"T","U","V" ,"W","X","Y","Z" };

double f_array[26] = {8.12,1.49,2.71,4.32,12.02,2.30,2.03,5.92,7.31,0.10,0.69,
3.98,2.61,6.95,7.68,1.82,0.11,6.02,6.28,9.10,2.88,1.11,2.09,0.17,2.11,0.07 };
```

The array of characters is declared as a string array due to casting issues that would be discussed later in this documentation.

Step 2: Create array of Huffman objects to encapsulate the data.

```
Huffman h_array[26]; //array of huffman objects
for (int i = 0; i < 26; i++)
{
	h_array[i] = Huffman(c_array[i], f_array[i]);
}
```

1.2.2 Huffman main Function

This function takes the huffman array created in the previous step to apply the huffman logic and return a huffman array with the objects' huffman codes filled.

```
Huffman* Huffman_main(Huffman* h_array)
```

Step 3: Sort the array once before working on it.
We need to sort the array for the first time before applying the algorithm. The sorting is done based on the probability of the objects.

To sort the array for the first time, the following function is used. The implementation would be discussed in its section.

```
sort_func(h_array);
```

To check that the sort function worked successfully a function that outputs to the command prompt is used.

```
h_array->Huffman_check_sort(h_array);
```

Step 4: Create vector of huffman objects to work on. The importance of this vector is its dynamic size, this vector will be destroyed once we exit the function, so it is just a temporary data holder for our operaations.

```
vector<Huffman> operations;
for (int i = 0; i < 26; i++)
{
	operations.push_back(h_array[i]);
}
```

The operation vector is now ready to be passed to the create Huffman function which is responsible for the creation and storing the generated Huffman string.

```
h_array->Create_Huffman(operations, h_array);
```

Finally, we return the huffman objects array that has the huffman code encapsulated within them.

1.2.3 Sort Function

This function is responsible for sorting the Huffman array for the first time in descending order based on the probabilities f the Huffman objects.
The sorting idea depends on comparing the first element of the array with rest of the array and swapping objects in case that the current object at index "i" has a smaller probability than the object at index j.

Not to lose the value during swapping a temp is used to hold the first value while equating the second value to the first one

```
Huffman* sort_func(Huffman* h_array)
{
        // To sort the array for the first time before beginning the process
        Huffman temp
        for (int i = 0; i < 26; i++)
        {
                for (int j = i + 1; j < 26; j++)
                {
                        if ((h_array[i].p) < (h_array[j].p))
                        {
                                temp = h_array[i];
                                h_array[i] = h_array[j];
                                h_array[j] = temp;
                        }
                }
        }
        return h_array;
}
```

1.2.4 Create Huffman

counter is a static variable which is used to check that 25 summations where done, until then we will keep calling the function recursively.
The last 2 elements of the operation vector are stored in variables and popped.A is a Huffman object that has its Huffman code inside it, the searcher modifier function is the one responsible for incriminating the Huffman code, it looks for the character in the Huffman array to an compare it with each character in the sticking together characters in the operations vectors popped attribute to be able to alter the Huffman code for the specified Huffman object. Once the Huffman code is incremented we will stick both letters together and push them back in the vector, note that the vector size is reduced by 1 every addition, and the counter is incremented by 1. Sticking both characters together is one for a reason, this is because the rest of the sequence for the Huffman code will be the same for the 2 elements stick-ed together. We will keeping adding sticking and re-sorting the operation vector until we finally generate all the Huffman codes. Sort Huff function is responsible for sorting the operations vector based on probabilities.

```
Huffman* Create_Huffman(vector <Huffman> operations, Huffman* h_array)
{

        if (counter != 25)
        {
                counter++;
                Huffman A, B, C;
                string new_char;
                A = operations.back(); // last element
                operations.pop_back();
                B = operations.back();
                operations.pop_back();
                h_array = searcher_modifier(h_array, A, B); // increments the huffman code
                new_char = A.c + B.c; // forms a new character and pushes it
                C = Huffman(new_char, A.p + B.p);
                operations.push_back(C);
                operations = sort_Huff(operations);
                return Create_Huffman(operations, h_array);
        }
```

```
        else
        {
                for (int i = 0; i < 26; i++)
                {
                        reverse(h_array[i].h.begin(), h_array[i].h.end()); // to reverse the huffman string
                }
                return h_array;
        }
}
```

1.2.5 Searcher Modifier Function

The searcher modifier function is the one responsible for incriminating the Huffman code, it looks for the character in the Huffman array to an compare it with each character in the sticking together characters in the operations vectors popped attribute to be able to alter the Huffman code for the specified Huffman object.

```
Huffman* searcher_modifier(Huffman* h_array, Huffman A, Huffman B)
        {

                for (int i = 0; i < A.c.size(); i++)
                {
                        for (int j = 0; j < 26; j++)
                        {
                                if (h_array[j].c[0] == A.c[i])
                                {
                                        h_array[j].h = h_array[j].h + '1';
                                        break;
                                }
                        }
                }
                for (int i = 0; i < B.c.size(); i++)
                {
                        for (int j = 0; j < 26; j++)
                        {
                                if (h_array[j].c[0] == B.c[i])
                                {
                                        h_array[j].h = h_array[j].h + '0';
                                        break;
                                }
                        }
                }
                return h_array;
        }
```

1.2.6 Sort Huff function

This function is responsible for sorting the operations vector through the use of a stack.

```
vector <Huffman> sort_Huff(vector <Huffman> operations)
        {
                stack<Huffman>s;
                int counter = 0;
                //takes the last element in the vector(The one added) and places it in position
                Huffman temp = operations.back();
                operations.pop_back();
```

```
        //cout << "operation size: " << operations.size() << endl;        //debugging
        for (int i = operations.size() - 1; i >= 0; i--)
        {
            //cout << i << endl;              //debugging
            if (operations.at(i).p < temp.p)
            {
                counter++;
                s.push(operations.back());
                operations.pop_back();
            }
            else {
                break;
            }

        }
        operations.push_back(temp); // no changes happen push back the temp to its place
        if (counter != 0)
        {
            while (s.empty() == false)
            {
                operations.push_back(s.top());
                s.pop();
            }

        }
        return operations;
    }
```

1.2.7 Get file content and save file content

Two more function that were used are the get and save file contents, those functions are responsible to read the file into a single string array and save the file into a single string array as well.

```
string Get_File_Content(string name)
{
    fstream my_file;
    string file_content="";
    my_file.open(name+".txt", ios::in);
    if (!my_file)
    {
        cout << "File␣not␣created!"<<endl;
    }
    else
    {
        cout << "File␣created␣successfully!"<<endl;
        my_file >> file_content;
        my_file.close();
    }
    return file_content;
}
void Save_File_Content(string name, string huffman)
{
    fstream my_file;
    my_file.open(name+".txt", ios::out);
    if (!my_file)
    {
```

```
            cout << "File␣not␣created!"<<endl;
        }
        else
        {
            cout << "File␣created␣successfully!"<<endl;
            my_file << huffman;
            my_file.close();
        }
}
```

1.3 Results

1.3.1 Huffman Code

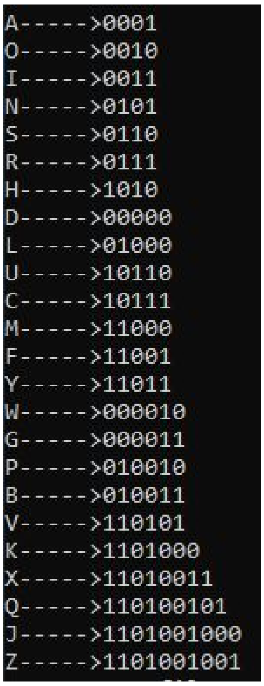

Figure 1: Alphabet Huffman code

1.3.2 Requesting a file name to read and a file name to save to

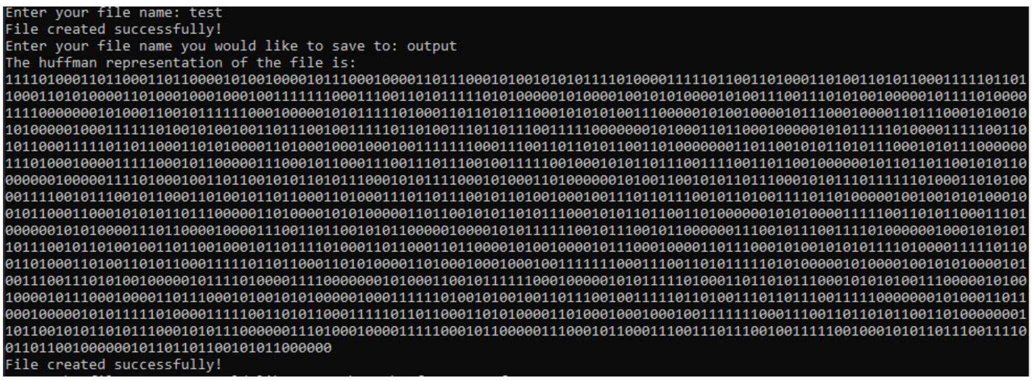

Figure 2: Simulating Task 1

1.3.3 Data Analysis

```
-------------------------------------Using the english character freqency set----------------------------------------
Sum of frequencies: 99.99
The average length is: 4.21192
The bit sequence length is:1991
Entropy is: 4.18139
```

Figure 3: Data analysis on results

2 Task 2:Design a Huffman code based on a frequency of character set in a file

2.1 Reading the data

The user is prompted to input the name of the file he would like to use.

```
Enter the file name you would like to gather the frequency from: test
File created successfully!
```

Figure 4: Requesting to input a file name

2.1.1 Frequency file former

Once we read the data into a single string we will use the following function to form the frequency array.

```
double* Frequency_file_former(string name,double* f_array_2)
{
        string content;
        content = Get_File_Content(name);
        string s;
        int counter_freq = 65; //ASCII for letter A
        int temp_counter=0;
        //double freq[26];
        while (counter_freq <= 90)
        {
                for (int i = 0; i < content.length(); i++)
                {
                        //s.push_back(content[i]);
                        if (content[i] == counter_freq)
                        {
                                temp_counter++;
                        }
                }
                f_array_2[counter_freq - 65] = temp_counter;
                temp_counter = 0;
                counter_freq++;
        }
        return f_array_2;
}
```

A similar approach to task 1 is used to generate the following results.

2.2 Results

2.2.1 Huffman Code

```
A----->011
E----->101
T----->111
S----->0010
N----->0011
I----->0101
O----->1000
R----->1100
H----->1101
U----->00001
L----->00011
Y----->01001
G----->10011
P----->000000
C----->000100
D----->000101
M----->010001
F----->100100
B----->100101
W----->0000010
K----->0000011
V----->0100000
Q----->010000110
X----->010000111
J----->010000100
Z----->010000101
```

Figure 5: Alphabet Huffman code

2.2.2 Data Analysis

```
------------------------------------------Using the File character freqency set------------------
Sum of frequencies: 471
The bit sequence length is:1966
The average length is: 4.1741
Entropy is: 4.1545
```

Figure 6: Data analysis on results

3 Comparing the results

Since the same test file is used, we expect the second approach used in **Task 2** ,where the frequency of the characters in that specific file is obtained, to have a better result than the approach used in **Task 1**.

```
---------------------------------------Using the english character freqency set---------
Sum of frequencies: 99.99
The average length is: 4.21192
The bit sequence length is:1991
Entropy is: 4.18139
-------------------------------------------Using the File character freqency set--------
Sum of frequencies: 471
The bit sequence length is:1966
The average length is: 4.1741
Entropy is: 4.1545
```

Figure 7: Comparing Results