Text Compression

Mona Ahmad Kamel, Maya Hussein, Nada Badawi

CSCE 2202: Analysis and Design of Algorithms, Section 02

Department of Computer Science and Engineering, AUC

Introduction

Text compression is a technique where each symbol in a text file is given a specific binary code. The purpose of the coding is to produce a compressed file with a smaller file size in order to save time and space. In this project, Huffman's coding is implemented as it is the most efficient text compression technique.

Problem Definition:

For a given file, it is consuming to transfer and store the original file. Hence, the process is optimized using Huffman Coding. Given a set of symbols, it is feasible to calculate their frequencies and probabilities, and then find a set of binary code words that minimize the average length of the coded symbols. This solution was first introduced by David Huffman in 1952 based on Binary Merge Trees.

Methodology:

Huffman coding is an optimal code that uses the greedy algorithm technique. Since the purpose of text compression is to reduce the file size without the loss of the original data (lossless compression), Huffman's coding provides the most frequent symbols with codes of small length.

Specification of Algorithms to be used:

To compress the input file administered by the user, Huffman coding is required to take the data from the input text file and traverse through it character by character to read and store them in two vectors. The first vector called input, stores all characters in their given order. Meanwhile, the second vector, named alphabet, stores the set of characters. The difference between both is the number of times a character is stored in the vector. To illustrate, the first vector stores the set of characters in the following way: "abaabccb", while the alphabet stores it

as follows: “abc”, i.e without repetition. Then, the algorithm calculates the frequency in which each symbol occurs, and the total number of characters in the file. This way, we can calculate the probability of occurrence of each character. All of these pieces of information are stored in ‘alphabet’, since it is a vector of a type-defined data type (struct).

The following is the pseudocode for the discussed section:

```
ALGORITHM CalculateProbabilty(fileName, alphabet,input)
{
    occurred = false;
    ifstream readFile(fileName.c_str());
    for  i = 0 to   alphabet.size() - 1 do
        alphabet[i].freq = 1;
    char byte;
    if readFile.fail()
    {
        cout << "Error!";
        exit(1);
    }
    k = 0, j = 0;
    while (readFile.get(byte))
    {
        input[k] = byte;
        k++;
        for i = 0 to alphabet.size() - 1 do
        {
            if (byte == alphabet[i].symbol)
            {
                alphabet[i].freq++;
                occurred = true;
            }
        }
        if (!occurred)
        {
            alphabet[j].symbol = byte;
            alphabet[j].virtualNode = byte;
            j++;
        }
        occurred = false;
    }
    input.resize(k);
    alphabet.resize(j);
    int total = 0;
    for i = 0 to  alphabet.size() - 1do
             total += alphabet[i].freq;
```

```
    for i = 0 to  alphabet.size() - 1do
            alphabet[i].prob = (double)alphabet[i].freq / total;
}
```

The compiler then takes that vector of alphabets and sorts it using a heap in O(nlogn) time. While sorting it uses a function named *"merge"* that takes the nodes/data with the lowest two probabilities and combines the two nodes together, or in other words; it creates a virtual node with the added probabilities, and the collection of the new characters with a frequency indicated to be -1. This step is repeated until all nodes have been merged into one single virtual node and is stored for reverse traversal when setting "0"s and "1"s.

Below is the pseudocode for the desired section:

```
heapify(vector, n, i)
{
    smallest = i;
    l = 2 * i + 1;
    r = 2 * i + 2;

    if (l < n && vector[l].prob > vector[smallest].prob)
        smallest = l;

    if (r < n && vector[r].prob > vector[smallest].prob)
        smallest = r;

    if (smallest != i)
    {
        swap(vector[i], vector[smallest]);
        heapify(vector, n, smallest);
    }
}


build_heap(vector, n)
{
    for (i = n / 2 - 1 → 0)
        heapify(vector, n, i);
}

heap_sort(vector, n)
{
    build_heap(vector, n);
    for (i = n - 1 → 0) {
```

```
        swap(vector[0], vector[i]);
        heapify(vector, i, 0);
    }
}

merge(Symbol a1, Symbol a2)
{
    Symbol merged;
    merged.virtualNode = a1.virtualNode + a2.virtualNode;
    merged.prob = a1.prob + a2.prob;
    merged.freq = -1;

    return merged;
}
```

Afterwards to initalize the nodes with their unique binary expressions, the virtual nodes are stored into a vector named *storedSymbol* and are reversely traversed over, where it concatenates the left side of the virtual nodes with a "0," and the right with a "1". This process is repeated up until the compiler reaches the leaf nodes, where the data (leaf nodes) are then stored in a code table with their desired codes. The following is the pseudocode for the explained section:

```
mergeTree(vector<Symbol>& alphabet)
{
    Create a copy of alphabet called copy_alphabet
    j=0;

    while (copy_alphabet.size() > 1)
    {
        heap_sort(copy_alphabet, copy_alphabet.size());

        dummy = merge(copy_alphabet[0], copy_alphabet[1]);

        storedSymbol[j] = copy_alphabet[0];
        storedSymbol[++j] = copy_alphabet[1];

        storedSymbol[++j] = dummy;
        copy_alphabet.push_back(dummy);
        copy_alphabet.erase(copy_alphabet.begin());
        copy_alphabet.erase(copy_alphabet.begin());
        j++;
    }
```

```
        Resize stored symbol to j
    storedSymbol[storedSymbol.size() - 1].code = "";
    for (i = storedSymbol.size() - 2 → 1)
    {
        storedSymbol[i].code = storedSymbol[i + 1].code + "1";
        storedSymbol[--i].code = storedSymbol[i + 2].code + "0";
        i--;
        for (j = i + 1 → storedSymbol.size())
        {
        if (storedSymbol[i].virtualNode == storedSymbol[j].virtualNode)
                storedSymbol[i].code = storedSymbol[j].code;
        }
    }
    storedSymbol[0].code = storedSymbol[2].code + "0";
    storedSymbol[1].code = storedSymbol[2].code + "1";


    int k(0);
    for (i = 0 → storedSymbol.size())
    {
        if (storedSymbol[i].virtualNode.length() == 1)
        {
            codeTable[k] = storedSymbol[i];
            k++;
        }
    }

    codeTable.resize(k);

    return codeTable;

}
```

The algorithm WriteCodedFile simply matches the characters that were taken from the input file and stored in the input vector with the characters in the code table. When a match is found, then the corresponding code for the symbol is written on the output file to be given to the user. The following is a description of the algorithm:

```
ofstream WriteCodedFile(vector<Symbol> codeTable, vector<char> input,
string outFile)
{
    vector<int> Char;
    ofstream writeComp;
    writeComp.open(outFile);

    if (writeComp.fail())
```

```
        cout << "Failed to open.\n";
    for i = 0 to  input.size() - 1 do
    {
        for j = 0 to codeTable.size() - 1 do
        {
            if (codeTable[j].symbol == input[i])
                writeComp << codeTable[j].code;
        }
    }
    for i = 0 to codeTable.size() - 1 do
                writeComp  <<  endl<<codeTable[i].symbol  <<  "    "  <<
codeTable[i].code;
    return writeComp;
}
```

Regarding the decompression section of the program, the algorithm requires only the input file that contains the binary string and the code table. The algorithm stores the binary string in a string and traverses it. The method of traversing involves choosing the first bit and comparing it with the codes of the symbols given in the table. If there exists a symbol with the code corresponding to the bit chosen, then the symbol is written in the output file. If not then the chosen bit and the one adjacent to it are chosen together to be compared with the code table. This continues to happen until a matching code is found. Once this happens the next one bit is chosen. A description of the algorithm is given below:

```
decompression()
{
      Ask the user to insert compressed and decompressed file paths

    while (!file.eof())
    {
        file >> temp;
        if (temp[0] != '"' && temp != "\n")
        {
            sym.virtualNode = temp;
            file >> temp;
            sym.code = temp;
            Push back symbol in vector
        }
        else
        {
            if (temp.substr(1, 2) == "\\n")
            {
```

```
                sym.virtualNode = "\n";
                file >> temp;
                sym.code = temp;
                  Push back symbol in vector

            }
            else if (temp[0] == '"')
            {
                file >> temp;
                string s;
                s.push_back('"');
                s.push_back(' ');
                s.push_back('"');
                sym.virtualNode = s;
                file >> temp;
                sym.code = temp;
                vec.push_back(sym);
            }
        }
    }
    for (i = 0 → compressed.length())
    {
      Push back compressed[i] to str
        index = find(str, vec);
        if (index != -1)
        {
            temp = vec[index].virtualNode;
            if (temp[0] != '"' && temp != "\\n")
                newfile << temp;
            else if (temp == "\\n")
                newfile << '\n';
            else if (temp[0] == '"')
                newfile << " ";
            str = "";
        }
    }
  Close both files: newfile and file
}
```

Data Specifications:

There were two methods of data used: the data formatted by the user and the data structures used in the algorithm itself.

Firstly, the user is asked to indicate their choice of source coding style, and then to input both the input and output paths/file they would like to use for compression/decompression.

In order to compress and decompress files using huffman coding, a unique set of data structures were utilized to maintain space and time efficiency. The following are the major structures used for text compression:

1. Vectors

   They were used to store the input data from the file. Vectors were preferred over an array, for example, because its flexibility to resize is advantageous. Since the number of characters to be stored changes depending on the input file, then the data structure that stores the input should be resizable. They were also used to store the merged symbols for it presents easier access to every symbol, and gives potential to resizing the table based on every merge.

2. Structs

   Structs were used to define many attributes related to one entity, i.e the characters retrieved from the file. Each character has its own symbol, code assigned by the huffman tree, frequency, probability of occurrence. This allows it to treat each node in the tree as one entity containing all of its related information. Moreover, structs were used to store virtual nodes as an entire unit, where every virtual node has a unique string of merged symbol, a standard frequency of -1, and a temporary code that is utilized in encoding the leaf nodes.

3. Strings

   Strings were used to store the code assigned by the huffman tree to each symbol. They were also used to receive and store the input and output file paths entered by the user. This process provided a lot more flexibility for the paths as string were easily implemented into fstream data types.

4. Binary Tree

The binary tree is the main data structure used to run the programs. The huffman code is based on the binary merge tree. The tree is binary to be able to assign 0s and 1s to the children, where the right child takes a 1 and the left child takes a 0. The binary tree was constructed by sorting the nodes and with every frequency, it merges lowest two probabilities, and assigning a virtual node them, where the indicated node is also assign a code bit for future reverse transversal.

5. Heap

Heaps were primarily used in heap sort algorithm, which is was the fundamental sorting algorithm, for it provides O(nlogn) complexity. They initially take the vector of symbols, where every character is a node, and sort them according to their probabilities in a minimum heap. This process was also conducted to merge firstly the leaf nodes, and consequently the virtual nodes with the lowest probabilities. This reduces the time and space complexity, for no data was necessarily overstored in a new array or vector. The heap simply overwrote in its original vector.

Experimental Results:

Several files were tested with compression and decompression, and below is a sample of the collected data:

Source Coding Style	Input	Output

Compression Ex1	`abdebdeaece2920c`	`01110000111100001110111100`
		`01110101001010101000`
		`9 0100`
		`0 0101`
		`c 000`
		`d 001`
		`b 100`

		2 101 a 011 e 11
Compression Ex2	Hello! This project is for CSCE 2202 Course. The project is about text compression using Huffman coding, and this file is for testing.	01001001001000010000111011 10010111101110001101101010 11111000000011110101011010 01011001000111101010111111 10001101001111100100010111 10010001110111110100001000 01011100100011100100110101 01000111011100101111011101 11000110110011110000000111 10101011010010110010001111 01010111111100110100101110 10101010001111000100111001 00100011101100110101111100 00000111001101110111010110 10001111010101011101000010 01011110110101010110001100 00111111100110001111011001 10101001110100001001010111 01011111001100010100111111 00001101101010111111100010 10000011001111101010111111 10001101001111111000100110 11000101000010010101111 0 0 0101110 S 0101111 , 0111010 E 0111011 H 0100100 b 0100101 x 1100100 ! 1100101 . 011110 m 011111 j 010110 d 010011 T 011100 a 110011 p 00000 l 00001 C 00100 g 00101 2 01000 c 01100 h 01101 u 01010 f 11000 n 0001 r 0011

		t 1000 e 1001 i 1010 s 1011 o 1101 111
Decompression	011100001111000011101111 000111010100101010100 9 0100 0 0101 c 000 d 001 b 100 2 101 a 011 e 11	abdebdeaece2920c

Below is a sample of the console screenshot of the procedure:

Microsoft Visual Studio Debug Console

```
                            Welcome to My Codec :)
                            Kindly choose what would you like to do:
                            Do you like to:
                            1. Compress
                            2. Decompress
                            Your Choice: 1

Please insert your input file path:
D:\Fall 2021\CSCE220202 - ANL&DSGN OF ALGORITHMS\Assignments\CodingProjects\CoderDecoder\Codec_005\TextFileName.txt

Path you entered: D:\Fall 2021\CSCE220202 - ANL&DSGN OF ALGORITHMS\Assignments\CodingProjects\CoderDecoder\Codec_005\TextFileName.txt
Please insert your output file path:
D:\Fall 2021\CSCE220202 - ANL&DSGN OF ALGORITHMS\Assignments\CodingProjects\CoderDecoder\Codec_005\CompressedFile.txt
            a          0.125
            b          0.125
            d          0.125
            e           0.25
            c          0.125
            2          0.125
            9         0.0625
            0         0.0625


Code: 0100
Probability: 0.0625  L: 4
Code: 0101
Probability: 0.0625  L: 4
Code: 000
Probability: 0.125  L: 3
Code: 001
Probability: 0.125  L: 3
Code: 100
Probability: 0.125  L: 3
Code: 101
Probability: 0.125  L: 3
Code: 011
Probability: 0.125  L: 3
Code: 11
Probability: 0.25  L: 2
AvgL: 2.875

Here is H: 2.875
Eta: 1


Compression Ratio: 0.359375
Encoding Efficiency: 1
```

```
H      0.00746269
e       0.0671642
l       0.0223881
o       0.0746269
!      0.00746269
         0.164179
T       0.0149254
h       0.0298507
i       0.0746269
s       0.0746269
p       0.0223881
r       0.0447761
j       0.0149254
c       0.0298507
t       0.0597015
f       0.0373134
C       0.0223881
S      0.00746269
E      0.00746269
2       0.0223881
0      0.00746269
u       0.0298507
.       0.0149254
a       0.0223881
b      0.00746269
x      0.00746269
m       0.0149254
n       0.0447761
g       0.0223881
d       0.0149254
,      0.00746269


Code: 0101110
Probability: 0.00746269  L: 7
Code: 0101111
Probability: 0.00746269  L: 7
Code: 0111010
Probability: 0.00746269  L: 7
Code: 0111011
```

Microsoft Visual Studio Debug Console

```
Code: 00101
Probability: 0.0223881  L: 5
Code: 01000
Probability: 0.0223881  L: 5
Code: 01100
Probability: 0.0298507  L: 5
Code: 01101
Probability: 0.0298507  L: 5
Code: 01010
Probability: 0.0298507  L: 5
Code: 11000
Probability: 0.0373134  L: 5
Code: 0001
Probability: 0.0447761  L: 4
Code: 0011
Probability: 0.0447761  L: 4
Code: 1000
Probability: 0.0597015  L: 4
Code: 1001
Probability: 0.0671642  L: 4
Code: 1010
Probability: 0.0746269  L: 4
Code: 1011
Probability: 0.0746269  L: 4
Code: 1101
Probability: 0.0746269  L: 4
Code: 111
Probability: 0.164179  L: 3
AvgL: 4.44776

Here is H: 4.41345
Eta: 0.992287


Compression Ratio: 0.55597
Encoding Efficiency: 0.992287
```

Conclusion:

In conclusion, the huffman code was able to solve the problem defined above. It gave to every character a flexible set binary digits based on its frequency and probability; unlike, the standard ASCII definition, which has a fixed set of binary codes per character enlarging the size of file. Overall, The program works perfectly on compressing and decompressing the files. It can output the compressed file size, the encoding efficiency, the compression ratio. Moreover, the input files are analyzed correctly to obtain the symbols written, their frequency, probability, and their unique Huffman code. On the other hand, the program decompresses, when desired by the user, a compressed file and gives the user the output file that is translated through the code table provided to the decoder.

Appendix: Listing of all Implementation Codes

```
struct Symbol
{
    char symbol;
    int freq;
    double prob;
    string virtualNode;
    string code;
};
vector<Symbol> storedSymbol;
vector<Symbol> codeTable;
void Menu();
void CalculateProbabilty(string& fileName, vector<Symbol>& alphabet,
vector<char>& input);
ofstream WriteCodedFile(vector<Symbol> codeTable, vector<char> input,
string);
void CalculateCRH(vector<Symbol>& alphabet, double& compressionratio,
double& H);
void CompChoice();

void heapify(vector<Symbol>& arr, int n, int i);
void build_heap(vector<Symbol>& arr, int n);
void heap_sort(vector<Symbol>& arr, int n);
Symbol merge(Symbol a1, Symbol a2);
vector<Symbol>  mergeTree(vector<Symbol>& alphabet);


int find(string required, vector<Symbol>vec);
void decompression();


int main()
{
    Menu();
}

void Menu()
{
    cout << "\t\t\t\tWelcome to My Codec :) \n";
    cout << "\t\t\t\tKindly choose what would you like to do:\n";
      cout << "\t\t\t\tDo you like to: \n\t\t\t\t1. Compress\n\t\t\t\t2.
Decompress\n";
    cout << "\t\t\t\tYour Choice: ";
    int choice;
    cin >> choice;
    cout << "\n";
    if (choice == 1)
        CompChoice();
    else if (choice == 2)
        decompression();
    else
```

```
    {
        cout << "\t\t\t\tPlease Try Again.\n";
        Menu();
    }
}

void CompChoice()
{
    string inpath, outpath, dummy;
    cout << "Please insert your input file path: \n";
    getline(cin >> ws, inpath);
    cout << "\nPath you entered: " << inpath << endl;
    cout << "Please insert your output file path: \n";

    getline(cin >> ws, outpath);
    vector<char> input(5000);
    vector<Symbol> alphabet(5000);
    double compressionR = 0, H = 0;
    CalculateProbabilty(inpath, alphabet, input);
    codeTable = mergeTree(alphabet);
    WriteCodedFile(codeTable, input, outpath);
    CalculateCRH(codeTable, compressionR, H);
      cout << "\n\nCompression Ratio: " << compressionR << "\nEncoding
Efficiency: " << H << "\n";
}

void CalculateProbabilty(string& fileName, vector<Symbol>& alphabet,
vector<char>& input)
{
    bool occurred = false;
    ifstream readFile(fileName.c_str());
    for (int i = 0; i < alphabet.size(); i++)
    {
        alphabet[i].freq = 1;
    }
    char byte;
    if (readFile.fail())
    {
        cout << "Error!";
        exit(1);
    }
    int k = 0, j = 0;
    while (readFile.get(byte))
    {
        input[k] = byte;
        k++;
        for (int i = 0; i < alphabet.size(); i++)
        {
            if (byte == alphabet[i].symbol)
            {
                alphabet[i].freq++;
                occurred = true;
```

```
            }
        }
        if (!occurred)
        {
            alphabet[j].symbol = byte;
            alphabet[j].virtualNode = byte;
            j++;
        }
        occurred = false;
    }
    input.resize(k);
    alphabet.resize(j);
    int total = 0;
    for (int i = 0; i < alphabet.size(); i++)
    {
        total += alphabet[i].freq;
    }
    for (int i = 0; i < alphabet.size(); i++)
    {
        alphabet[i].prob = (double)alphabet[i].freq / total;
    }
    for (int i = 0; i < alphabet.size(); i++)
    {
                cout << setw(15) << alphabet[i].symbol << setw(15) <<
alphabet[i].prob << endl;
    }
    cout << "Original File size: " << total << " bytes\n";
}
ofstream WriteCodedFile(vector<Symbol> codeTable, vector<char> input,
string outFile)
{
    vector<int> Char;
    ofstream writeComp;
    writeComp.open(outFile);
    string all;
    if (writeComp.fail())
        cout << "ERROR";
    for (int i = 0; i < input.size(); i++)
    {
        for (int j = 0; j < codeTable.size(); j++)
        {
            if (codeTable[j].symbol == input[i])
            {
                all += codeTable[j].code;
                writeComp << codeTable[j].code;
            }
        }
    }
    for (int i = 0; i < codeTable.size(); i++)
    {
                writeComp << endl << codeTable[i].symbol << "    " <<
codeTable[i].code;
```

```
    }
     cout << "Size of the compressed file: " << all.length() / 8.0 << "
bytes\n";
    return writeComp;
}
void  CalculateCRH(vector<Symbol>&  alphabet,  double&  compressionratio,
double& H)
{
    double avgL = 0, sum = 0;
    double temp1, temp2;
    for (int i = 0; i < alphabet.size(); i++)
    {
        temp1 = alphabet[i].prob;
        temp2 = alphabet[i].code.length();
        cout << "Code: " << alphabet[i].code << endl;
        cout << "Probability: " << temp1 << "  L: " << temp2 << endl;
        avgL = avgL + temp1 * temp2;
        sum = temp1 * (log(1 / temp1) / log(2));
        H += sum;
    }
    compressionratio = avgL / 8.0;
    H = H / avgL;
}


void heapify(vector<Symbol>& arr, int n, int i)
{
    int smallest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l].prob > arr[smallest].prob)
        smallest = l;

    if (r < n && arr[r].prob > arr[smallest].prob)
        smallest = r;

    if (smallest != i)
    {
        swap(arr[i], arr[smallest]);
        heapify(arr, n, smallest);
    }
}
void build_heap(vector<Symbol>& arr, int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
}
void heap_sort(vector<Symbol>& arr, int n)
{
    build_heap(arr, n);
    for (int i = n - 1; i >= 0; i--) {
```

```
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
Symbol merge(Symbol a1, Symbol a2)
{
    Symbol merged;
    merged.virtualNode = a1.virtualNode + a2.virtualNode;
    merged.prob = a1.prob + a2.prob;
    merged.freq = -1;

    return merged;
}
vector<Symbol>  mergeTree(vector<Symbol>& alphabet)
{
    vector<Symbol> copy_alphabet;
    copy_alphabet = alphabet;
    Symbol dummy;
    int j = 0;
    storedSymbol.resize(5000);
    while (copy_alphabet.size() > 1)
    {
        heap_sort(copy_alphabet, copy_alphabet.size());

        dummy = merge(copy_alphabet[0], copy_alphabet[1]);

        storedSymbol[j] = copy_alphabet[0];
        storedSymbol[++j] = copy_alphabet[1];

        storedSymbol[++j] = dummy;
        copy_alphabet.push_back(dummy);
        copy_alphabet.erase(copy_alphabet.begin());
        copy_alphabet.erase(copy_alphabet.begin());
        j++;
    }
    storedSymbol.resize(j);
    storedSymbol[storedSymbol.size() - 1].code = "";
    for (int i = storedSymbol.size() - 2; i > 1; i--)
    {
        storedSymbol[i].code = storedSymbol[i + 1].code + "1";
        storedSymbol[--i].code = storedSymbol[i + 2].code + "0";
        i--;
        for (int j = i + 1; j < storedSymbol.size(); j++)
        {
if (storedSymbol[i].virtualNode == storedSymbol[j].virtualNode)
                storedSymbol[i].code = storedSymbol[j].code;
        }
    }
    storedSymbol[0].code = storedSymbol[2].code + "0";
    storedSymbol[1].code = storedSymbol[2].code + "1";

    cout << endl << endl;
```

```
    codeTable.resize(5000);
    int k(0);
    for (int i = 0; i < storedSymbol.size(); i++)
    {
        if (storedSymbol[i].virtualNode.length() == 1)
        {
            codeTable[k] = storedSymbol[i];
            k++;
        }
    }

    codeTable.resize(k);
    return codeTable;

}



int find(string required, vector<Symbol>vec)
{
    for (int i = 0; i < vec.size(); i++)
    {
        if (required == vec[i].code)
            return i;
    }
    return -1; //acts as a flag
}
void decompression()
{
    int index;
    string str;
    ifstream file;
    ofstream newfile;
    string filename;
    string compressed;
    string temp;
    vector<Symbol>vec;
    Symbol sym;
    cout << "Enter the path of the compressed file: ";
    getline(cin >> ws, filename);
    file.open(filename);
    cout << "Enter the desired path of the non-compressed file: ";
    getline(cin >> ws, filename);
    newfile.open(filename);
    file >> compressed;
    while (!file.eof())
    {
        file >> temp;
        if (temp[0] != '"' && temp != "\n")
        {
            sym.virtualNode = temp;
            file >> temp;
```

```
            sym.code = temp;
            vec.push_back(sym);
        }
        else
        {
            if (temp.substr(1, 2) == "\\n")
            {
                sym.virtualNode = "\n";
                file >> temp;
                sym.code = temp;
                vec.push_back(sym);
            }
            else if (temp[0] == '"')
            {
                file >> temp;
                string s;
                s.push_back('"');
                s.push_back(' ');
                s.push_back('"');
                sym.virtualNode = s;
                file >> temp;
                sym.code = temp;
                vec.push_back(sym);
            }
        }
    }
    for (int i = 0; i < compressed.length(); i++)
    {
        str.push_back(compressed[i]);
        index = find(str, vec);
        if (index != -1)
        {
            temp = vec[index].virtualNode;
            if (temp[0] != '"' && temp != "\\n")
                newfile << temp;
            else if (temp == "\\n")
                newfile << '\n';
            else if (temp[0] == '"')
                newfile << " ";
            str = "";
        }
    }
    newfile.close();
    file.close();
}
```