BBM203 2020 Fall Assignment 4 Report

1 OCAK

Yazan: Selçuk Yılmaz 21828035

1- Problem

Making less byte txts with Huffman Tree Encoding way

2- What is Huffman Tree

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper ”A Method for the Construction of Minimum-Redundancy Codes”.

3- Frequency Calculate Algorithm

I calculate frequency with map. I take letter by letter from input.txt and make them lowercase then put them in encoder_map. If letter repeats itself then I add letters value 1. If it is not repeats itself then I put letter in encode_map and make its value 1. If input.txt has more than 1 line then I put “\” and “n” into map manually. At last I push all map items to a vector<pair> named as encoder_list. Function returns encoder_list.

```
std::vector<std::pair<int,string>> Frequency::calculate_frequency() {
    map<string, int> encoder;
    int line=0;
    vector<pair<int,string>> encoder_list;
    string original_text,temp_text;
    ifstream MyReadFile("input_1.txt");
    temp_text=" ";

    while (getline (MyReadFile, original_text))
    {
        for_each(original_text.begin(), original_text.end(), [](char & c){
            c = tolower(c);
        });

        if (line>=1){
            if(encoder.count("\\") > 0)
            {
                encoder.at("\\")++;
            }
            else{
                encoder.insert(pair<string, int>("\\", 1));
            }
            if(encoder.count("n") > 0)
            {
                encoder.at("n")++;
            }
            else{
                encoder.insert(pair<string, int>("n", 1));

            }

        }
        while (original_text.size()>0){

            temp_text = original_text.substr(0,1);
            original_text= original_text.substr(1);

            if(encoder.count(temp_text) > 0)
            {
                encoder.at(temp_text)++;
            }
            else{
                encoder.insert(pair<string, int>(temp_text, 1));

            }
        }
        line++;
    }
    MyReadFile.close();

    for (std::pair<std::string, int> element : encoder)
    {

        encoder_list.push_back(make_pair(element.second,element.first));
    }

    return (encoder_list);
}
```

4- Tree Add Algorithm

First I check my trees leftchild if it is empty then I add item to leftchild. If it is not then I checked rightchild and same thing happens here too. If it is not empty too then I add this two node a parent and make this parent a leftchild or rightchild depends on total frequency. You can see my code below.

```
void Tree::Add(std::string xitem,int xfrequency)
{
    if (tail->leftchild==NULL){
        tail->leftchild= new TreeNode();
        tail->leftchild->parent = tail;
        tail->leftchild->item = xitem;
        frequency+=xfrequency;
        tail->leftchild->leftchild = nullptr;
        tail->leftchild->rightchild = nullptr;
    }
    else if(tail->rightchild==NULL) {
        tail->rightchild = new TreeNode();
        tail->rightchild->parent=tail;
        tail->rightchild->item = xitem;
        frequency+=xfrequency;
        tail->rightchild->leftchild = nullptr;
        tail->rightchild->rightchild = nullptr;
    }
    else{
        if (frequency>xfrequency){
            tail->parent = new TreeNode();
            tail->parent->rightchild =tail;
            tail = tail->parent;
            tail->item = "";
            tail->leftchild = new TreeNode();
            tail->leftchild->parent = tail;
            tail->leftchild->item = xitem;
            frequency+=xfrequency;
            tail->leftchild->leftchild= nullptr;
            tail->leftchild->rightchild= nullptr;
        }
        else if (frequency<=xfrequency){
            tail->parent = new TreeNode();
            tail->parent->leftchild =tail;
            tail = tail->parent;
            tail->item = "";
            tail->rightchild = new TreeNode();
            tail->rightchild->parent = tail;
            tail->rightchild->item = xitem;
            frequency+=xfrequency;
            tail->rightchild->leftchild= nullptr;
            tail->rightchild->rightchild= nullptr;
        }
    }
}
```

5- Traverse Algorithm

I use this function a lot of times because it has recursion. This function purposes are traversing in my tree and making dictionary.txt and tree.txt and you will understand this 2 files are very important for my algorithm. First I look at leftchild then rightchild if they are Null then function returns nothing.

If they are not Null then I look their item. If it is not empty then I append that items name 0 or 1 depends on which side is the letter. And printing it into Tree.txt and Dictionary.txt.

```
void Tree::printInorder(struct TreeNode* node,string location)
{


    ofstream mytree;
    mytree.open ("tree.txt",std::ios_base::app);

    ofstream myfile;
    myfile.open ("dictionary.txt",std::ios_base::app);

    if (node == NULL){
        return;}



    /* first recur on left child */
    if (node->item!="")
    {

        if(location == "left")
        {
            movea.append("0");
            node->item=node->item.append(movea);
        }
        else if (location == "right")
        {

            movea.append("1");
            node->item=node->item.append(movea);

        }
        if (node==tail&&node->item!=""){
            mytree<<"    |  +- grandchild-"<<node->item<<endl;
            myfile << node->item <<endl;
            movea.clear();
            return;
        }
    }
    else{
        if (location=="left")
        {
            movea.append("0");
        }
        else if(location =="right")
        {
            movea.append("1");
        }
    }


    if (node!=tail&&location =="left"&&movea.substr(0,1)=="1"){
        if (node->parent->leftchild->item!=""||node->parent->rightchild->item!=""){
            mytree<<set_text<<"  +- Parent-"<<movea.substr(0,movea.size()-1)<<endl;
            set_text.append("  |");
        }
    }

    if (node!=tail && location =="left"&&movea.substr(0,1)=="0"){

        if (node->parent->leftchild->item!=""||node->parent->rightchild->item!=""){
            mytree<<set_text<<"  +- Parent-"<<movea.substr(0,movea.size()-1)<<endl;
            set_text.append("  |");
        }
```

```
    }
    if (node->item!="" && movea.substr(0,1)=="0")
    {
        mytree<<set_text<<"  +- "<<"grandchild-"<<node->item<<endl;
    }


    printInorder(node->leftchild,"left");

    if (node->item!="")
    {
        myfile << node->item <<endl;
    }

    printInorder(node->rightchild,"right");

    if (node->item!="" && movea.substr(0,1)=="1")
    {
        mytree<<set_text<<"  +- "<<"grandchild-"<<node->item<<endl;
    }

    if (location=="right"){
        set_text=set_text.substr(0,set_text.size()-3);
    }

    if (!movea.empty()) {
        movea.erase(std::prev(movea.end()));
    }


    myfile.close();
    mytree.close();
}
```

6- Encoding Algorithm

I calculate all letters frequencies with map in my code. You can see how I do it above. Then I return them to vector pair. After that I sort this vector with their frequencies. Then I make a sub_vectors where all less letters goes and more letters goes. If subvector1's frequency is bigger or equal to subvector2 then I put another letter to subvector2. So I can separate them into 2 branch in tree. Then I put them into tree one by one from bottom to top. I use recursion for this. Then I traverse in this tree and while I traversing I put letters copy and their values copy to dictionary.txt and save tree in tree.txt. For encoding, first I take all letters and their values into output_letters_vector then I take input.txt and find all letters one by one in output_letters_vector and print their values. Below pictures are all my encoding algorithm code. And I think it is more efficient because we are not traversing whole tree from top to bottom. We are just searching what we want in 1-dimensional vector.

```
void Control::Encode(string inputfile) {
    Frequency frequency;
    Tree branchlower,branchbigger,root;
    vector<pair<int,string>> encoder_list,sub_listlower,sub_listbigger;



    ofstream myfile;
    myfile.open ("dictionary.txt");
    myfile.close();

    encoder_list = frequency.calculate_frequency();
    sort(encoder_list.begin(), encoder_list.end());

    int totalfrequencylower=0,totalfrequencybigger=0;
    for (int i = 0; i < encoder_list.size() ; i++) {
        sub_listlower.push_back(encoder_list[i]);
        totalfrequencylower= totalfrequencylower + encoder_list[i].first;
        if (totalfrequencylower >= totalfrequencybigger && encoder_list[i] != encoder_list[encoder_list.size(
            totalfrequencybigger+=encoder_list[encoder_list.size() - 1].first;
            sub_listbigger.insert(sub_listbigger.begin(), encoder_list[encoder_list.size() - 1]);
            encoder_list.erase(encoder_list.end()-1);
        }
    }

    for (int i = 0 ; i < sub_listlower.size(); i++){
        branchlower.Add(sub_listlower[i].second, sub_listlower[i].first);
    }
    for (int i = 0 ; i < sub_listbigger.size(); i++){
        branchbigger.Add(sub_listbigger[i].second, sub_listbigger[i].first);
    }

    branchlower.TailCalibration();
    branchbigger.TailCalibration();

    ofstream myreset;
    myreset.open ("tree.txt");
    myreset.close();
    ofstream mytree;
    mytree.open ("tree.txt",std::ios_base::app);
    mytree<<"+- root"<<endl;
    if (branchlower.getFrequency() > branchbigger.getFrequency())
```

```
mytree<<"+- root"<<endl;
if (branchlower.getFrequency() > branchbigger.getFrequency())
{
    //branchlower is 1 branchbigger is 0 then
    mytree<<"   +- branch-left"<<endl;
    branchbigger.printInorder(branchbigger.tail, location: "left");
    mytree<<"   +- branch-right"<<endl;
    branchlower.printInorder(branchlower.tail, location: "right");
}
else{
    //branchlower is 0 branchbigger is 1 then
    mytree<<"   +- branch-left"<<endl;
    branchlower.printInorder(branchlower.tail, location: "left");
    mytree<<"   +- branch-right"<<endl;
    branchbigger.printInorder(branchbigger.tail, location: "right");
}

mytree.close();


vector<string> output_letters;
string Dictionary_text;
ifstream MyDictionary("dictionary.txt");
while (getline ( &: MyDictionary, &: Dictionary_text)){
    output_letters.push_back(Dictionary_text);
}
```

```
    string original_text,temp_text;
    ifstream MyEncodeFile(inputfile);
    int line=0;
    while (getline ( &: MyEncodeFile,  &: original_text))
    {

        if (line>=1){

            for (int i = 0; i < output_letters.size() ; i++)
            {
                if (output_letters[i].substr( _Off: 0, _Count: 1)=="\\")
                {
                    cout<<output_letters[i].substr( _Off: 1);
                    break;
                }

            }
            for (int i = 0; i < output_letters.size() ; i++) {
                if (output_letters[i].substr( _Off: 0, _Count: 1)=="n")
                {
                    cout<<output_letters[i].substr( _Off: 1);
                    break;
                }
            }
        }
        while (original_text.size()>0){
            temp_text = original_text.substr( _Off: 0, _Count: 1);
            original_text= original_text.substr( _Off: 1);
            for (int i = 0; i < output_letters.size() ; i++)
            {
                if (output_letters[i].substr( _Off: 0, _Count: 1)==temp_text)
                {
                    cout<<output_letters[i].substr( _Off: 1);
                    break;
                }
            }
        }
        line++;
    }
    line=0;
}
```

7- Decoding Algorithm

For this algorithm I use my saved file dictionary.txt then I put all letters output_letters_vector. Then I take all numbers from decode.txt and appended them to a string. if I can find any number values in output letters vector I clear string and print that letter.If I can't find any letter I append another number to string. Below Picture is my decoding algorithm. It is short because I used my saved files for this function. And I think it is more efficient because we are not traversing whole tree from top to bottom. We are just searching what we want in 1-dimensional vector.

```
void Control::Decode(string inputfile) {

    string original_text,temp_text;
    vector<string> output_letters;
    temp_text.clear();
    ifstream MyDecodeFile(inputfile);

    string Dictionary_text;
    ifstream MyDictionary("dictionary.txt");
    while (getline ( &: MyDictionary,  &: Dictionary_text)){
        output_letters.push_back(Dictionary_text);
    }
    while (getline ( &: MyDecodeFile,  &: original_text))
    {
        while (original_text.size()>0){
            temp_text.append( _Right: original_text.substr( _Off: 0, _Count: 1));
            original_text= original_text.substr( _Off: 1);
            for (int i = 0; i < output_letters.size() ; i++)
            {
                if (output_letters[i].substr( _Off: 1)==temp_text)
                {
                    cout<<output_letters[i].substr( _Off: 0, _Count: 1);
                    temp_text.clear();
                    break;
                }
            }
        }
    }
    cout<<endl;

}
```

8- List Tree Algorithm

For this algorithm I used my saved file tree.txt.Then I print that file to console. Below picture is my list tree algorithm. It is short Because I used my saved files for this function. And I think it is more efficient because we are not traversing whole tree we are just printing a tree.txt file.

```
void Control::ListTree() {
    string original_text,temp_text;
    ifstream MyTreeFile("tree.txt");
    while (getline ( & MyTreeFile,  & original_text))
    {
        cout<<original_text<<endl;
    }
}
```

9- Find Letter Algorithm

For this function first I create a vector then put all dictionary.txt letters in this vector then I search what you want from this vector.

```
void Control::Letters(string input) {
    vector<string> output_letters;
    string Dictionary_text;
    ifstream MyDictionary("dictionary.txt");
    while (getline (MyDictionary, Dictionary_text)){
        output_letters.push_back(Dictionary_text);
    }

        for (int i = 0; i < output_letters.size() ; i++)
        {
            if (output_letters[i].substr(0,1)==input)
            {
                cout<<output_letters[i].substr(1)<<endl;
                break;
            }
        }
}
```

10- How to Run My Program

My program execute for every command separately and takes only one command in one line .If you want to test my program you just need to write to console this codes.

- **make (This code will execute my makefile and create a exe file)**
- **./Main -i [input file name] -encode (This code will encode input and create tree.txt and dictionary.txt which my program needs)**
- **./Main -i [input file name] -decode (This code will decode input with dictionary.txt)**
- **./Main -s [letter] (This code will find letter value with dictionary.txt)**
- **./Main -l (This code will print tree with tree.txt)**