Hacettepe University – BBM203 – Assignment 4 Report

Berke Bayraktar

21992847

- Instructions for execution are given at the end of report
- Some of the code snippets given below for explaining the algorithms may have minor differences with the ones in the actual source code. This is because I have excluded some things that are not related to actual algorithm in order to make the code snippets more concise and easy to go through

Encode Operation:

In an abstract way the operation goes as, given a **message** to encode

A – First we need to generate a huffman tree for the message;

1- Create a node for each unique character in the message string
2- Assign frequencies to these character nodes according to the their appreances in message
3- While there is more than one (freq, node) pair left, keep merging the 2 nodes with lowest frequencies
4- Return the only left node (this will be the huffman tree)

After we obtain the huffman tree we will create a **look-up table for every character** so it is easier to encode.

We should observe that because of how the merging works characters will be always at leaf nodes and other internal nodes will be empty

B – To create this table, we will traverse the tree starting from the root as current node,

1- If the current node is NOT an empty node this means we are at a leaf node so we will map the character at this node to our current encoding
2- Else we will first visit the left sub-tree and since we are travelling to left we will add a "0" character to our current encoding
3- Then we will visit the right sub-tree and since we are travelling to right we will add a "1" character to our current encoding

After this part is done we now have a look-up table that maps every character in the message to an encoding

C – Using this map, we will generate an encoding for the whole message

In implementation encoding operation is a multi-stage operation that uses various classes and methods.

First we have, **Node2Freq** class

This class is responsible for storing nodes and their corresponding frequencises and applying merge operation on these nodes. It also keeps track of how many unmerged trees/nodes are left. It is essentially a wrapper class around the data strucure `list<int, pair<int, Node*>>` extended with methods we need for huffman encoding.

```
class Node2Freq {
private:
    std::list<std::pair<int,BTNode*>> f2n;
public:
    void merge();
    int size();
    BTNode* tree();
```

To create the huffman tree in part A:

```
BTNode* Huffman::generate_tree(std::string& message) {
    Node2Freq n2f(message);

    while(n2f.size() != 1) {
        n2f.merge();
    }
    return n2f.tree();
}
```

To create the table in part B:

```cpp
void Huffman::generate_encoding_from_tree
(BTNode *root, std::string& encoding, std::map<char, std::string>& encodings) {

    if (root->left == nullptr && root->right == nullptr) { // leaf node
        encodings.emplace(root->val, encoding);
    }
    if (root->left != nullptr) {
        generate_encoding_from_tree(root->left, encoding.append("0"), encodings);
        encoding.pop_back();
    }
    if (root->right != nullptr) {
        generate_encoding_from_tree(root->right, encoding.append("1"), encodings);
        encoding.pop_back();
    }
}
```

To create the encoded string in part C:

```cpp
std::string Huffman::encode(std::string& message, BTNode* huffman_tree) {
    std::string encoding;
    std::map<char, std::string> encodings;

    generate_encoding_from_tree(huffman_tree, encoding, encodings);

    std::string encodedMessage;
    for(char c : message) {
        encodedMessage.append(encodings[c]);
    }
    return encodedMessage;
}
```

Decode Operation

In an abstract way the operation goes as, given a **encoding** to decode and it's corresponding **huffman tree**,

Encoding will give us information about which direction should we go. And since non-empty nodes are leaf nodes, once we reach a leaf node we can say that we have decoded a character and add this character to the new decoded string we are constructing. Then continue from rest of the encoding string to decode the next character. Once the whole encoding string is iterated, the whole string will have been decoded.

We will iterate over the encoded string,

1- While the whole string is NOT iterated
    1- If current node is NOT empty then we are at a leaf node and have decoded a character we can return this decoded character
    2- Else, Using the encoding, decide which direction we should go (if current char is "0" we should go left, if "1" we go right)

This handles the inner (indented) recursive part of the algorithm, every time it finds a charcter (reaches a leaf node it will return the character)

int* idx is for keeping track of at which index of encoded message we are. The caller will terminate the search once we have iterated all encoding

```
char Huffman::read_char_from_encoding_tree(BTNode *root, std::string &encoded_message, int*
idx) {
    if (root->val != '*') { // then we are at a leaf node
        return root->val;
    }

    char direction = encoded_message[*idx];
    *idx = *idx + 1;

    if (direction == '0') { // go left
        return read_char_from_encoding_tree(root->left, encoded_message, idx);
    }
    else { // go right
        return read_char_from_encoding_tree(root->right, encoded_message, idx);
    }
}
```

This handles the outer iterative part of the algorithm, once the condition inside while is satisfied we will have been decoded the whole message

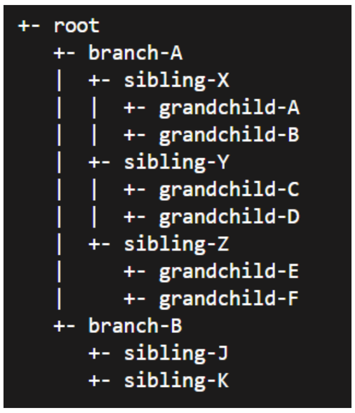
```
std::string Huffman::decode(std::string &encoded_message, BTNode* huffman_tree) {
    int current_idx = 0;
    std::string decoded_message;
    while (current_idx < encoded_message.size()) {
        char c = read_char_from_encoding_tree(huffman_tree, encoded_message, &current_idx);
        decoded_message.push_back(c);
    }
    return decoded_message;
}
```

List Operation

The goal for this operation is to print the huffman tree just like the other methods it is also a recursive algorithm since it requires tree traversal which is easy to implement using recursion

There are 4 parameters that we pass down during the recursion:

1- root of the current sub-tree
2- **indent**, for keeping track of how much space we should leave before printing out the element in the current root so that the tree looks better when we print it
3- **listing**, which is the actual string representation of the tree
4- **right**, a boolean value to determine if the current node is the right node of the previous tree, this is important because as we can see from the picture below if the current node is a right child (second child) of the previos root node then we have to decide for the indentation of the next node to be printed, accordingly.



```
+- root
   +- branch-A
   |  +- sibling-X
   |  |  +- grandchild-A
   |  |  +- grandchild-B
   |  +- sibling-Y
   |  |  +- grandchild-C
   |  |  +- grandchild-D
   |  +- sibling-Z
   |     +- grandchild-E
   |     +- grandchild-F
   +- branch-B
      +- sibling-J
      +- sibling-K
```

Caller of this method will pass 2 empty strings and a boolean value to represent if it is the right child (or last child) of the previous root node

And an edge case to consider that if the current node holds a line break character we cant just print this as it will actually cause a line break in the output and disturb the structure of the tree in string

The algorithm uses pre-order traversal,

1- Append the current node to the listing
2- If the current node is the right (last) child of the previous node then do not append the "|" character. This is because if the current node is the last child the next node will be at one indentation level back in the string representation (becuase we are using preorder traversal)
3- Print the left subtree
4- Print the right subtree

```
void Node2Freq::list(BTNode *root, std::string indent, std::string& listing, bool right) {
    if (root == nullptr) return;

    else {
        listing.append(indent + "+- " + root->val + "\n");
    }

    if (right)
        indent.append("   ");
    else
        indent.append("|  ");


    list(root->left, indent, listing, false);
    list(root->right, indent, listing, true);
}
```

The initial caller for this method is a wrapper for the listing method that will be given to client code. Here the first value for right child is passed as true. This is because just like the right child nodes, we do not need a "|" character for the very next node to be printed (which is its left node because of preorder traversal)

```
std::string Huffman::list(BTNode* huffman_tree) {
    std::string listing, indent;
    Node2Freq::list(huffman_tree, indent, listing, true);
    return listing;
}
```

Program Execution

Once the makefile is run, an executable named **output** will be created

The command line arguments program requires is same with the ones specified in assignment pdf.

For encoding a message

- ./output -i [INPUT_FILE.txt] -encode

this command will print the encoded text to console AND create 2 new files at the same directory with .exe file named;

1- encoding.txt : used for saving huffman encoded text
2- tree.txt : used for saving the huffman tree generated in encoding process

For decoding a message

- ./output -i [INPUT_FILE.txt] -decode

this command will decode the the encoded text in the previous step and print it out to console therefore it requires the encoded text (saved in encoding.txt) and the huffman tree (saved in tree.txt) encoding.txt is passed as a command line argument however tree.txt is not so it should be in the same directory with .exe file **this command should be executed as: ./output -i encoding.txt -decode**

Character Huffman encoding

- ./output –s [character]

this command will deserialize the Huffman tree generated in encoding and print the huffman code for that character to console so it should be present in the same directory with .exe (same with decoding operation)

List Tree

- ./output -l

this command requires the huffman tree file as well for deserializing and printing it to the console