Project4 Huffman Code

- Lin Juyi - 3180103721
- Lu Tian - 3180103740
- Gu Yu - 3190105872

1. Project Description

In 1953, David A. Huffman published his paper "A Method for the Construction of Minimum-Redundancy Codes", and hence printed his name in the history of computer science. As a professor who gives the final exam problem on Huffman codes, I am encountering a big problem: the Huffman codes are NOT unique. For example, given a string "aaaxuaxz", we can observe that the frequencies of the characters 'a', 'x', 'u' and 'z' are 4, 2, 1 and 1, respectively. We may either encode the symbols as {'a'=0, 'x'=10, 'u'=110, 'z'=111}, or in another way as {'a'=1, 'x'=01, 'u'=001, 'z'=000}, both compress the string into 14 bits. Another set of code can be given as {'a'=0, 'x'=11, 'u'=100, 'z'=101}, but {'a'=0, 'x'=01, 'u'=011, 'z'=001} is NOT correct since "aaaxuaxz" and "aazuaxax" can both be decoded from the code 00001011001001. The students are submitting all kinds of codes, and I need a computer program to help me determine which ones are correct and which ones are not.

Input Specification

Each input file contains one test case. For each case, the first line gives an integer N (2≤N≤63), then followed by a line that contains all the N distinct characters and their frequencies in the following format:

```
c[1] f[1] c[2] f[2] ... c[N] f[N]
```

where c[i] is a character chosen from {'0' - '9', 'a' - 'z', 'A' - 'Z', '_'}, and f[i] is the frequency of c[i] and is an integer no more than 1000. The next line gives a positive integer M (≤1000), then followed by M student submissions. Each student submission consists of N lines, each in the format:

```
c[i] code[i]
```

where c[i] is the i-th character and code[i] is an non-empty string of no more than 63 '0's and '1's.

Output Specification

For each test case, print in each line either "Yes" if the student's submission is correct, or "No" if not.

Note: The optimal solution is not necessarily generated by Huffman algorithm. Any prefix code with code length being optimal is considered correct.

2. Project Architecture

We use CMake to control the software compilation process in this project. The "main.cpp" contains the code of Input/Output. We put the main logical of our program part inside class "HuffmanTree" within "HuffmanTree.cpp". The

test code is in "test.cpp".The test case and result is in "testData.txt" and "result.csv". You can also execute task Proj4 to interactively test our program or run Proj4_test task and reproduce the test case&result if you want.

3. Algorithm Description

We divide the problem into two parts.

In first part, we check whether the input code system is a valid prefix code system or not. A prefix code is a type of code system (typically a variable-length code) distinguished by its possession of the “prefix property”, which requires that there is no whole code word in the system that is a prefix (initial segment) of any other code word in the system. [1] We first make sure there is no duplicate code in the code system. And then we construct a binary tree which represents the code system. After the construction of the tree, we examine every code by iterate the code in the tree by their encoding. If the iteration ends up in a leaf node, the code is not a prefix of other code, if not, the code is the prefix of some code and the prefix check failed.

```
def is_valid_prefix_code(code_list):
    if has_duplicate_code(code_list):
        return false
    tree_represent_the_code_system = construct_tree(code_list)
    for code in code_list:
        end_node = iterate(tree_represent_the_code_system, code)
        if end_node is not LEAF_NODE:
            return false
    return true;
```

In second part, we generate the Huffman Tree corresponds to the characters and their frequencies

```
def construct_huffman_tree(char_to_code_map, char_to_frequency_map):
    for char in char_to_frequency_map:
        create a tree Ti of a single node char
        frequency of Ti is the value of the char in char_to_frequency_map
    let trees be the set of tree
    while there are more than 2 tree in trees:
        let T1 and T2 be the two tree with min weight
        remove T1 and T2
        T3 = merge T1, T2
        frequency of T3 is T1.frequency + T2.frequency
        add T3 to trees
    return the unique tree in trees
```

In the main function, we compare the length of the code generated by the huffman tree and the input code system. If input code system perform as good as the huffman coding system, we recognize it's optimal.

```
def is_optimal_code_system(code_system)
    let code_list be the chars in the code_system
    if not is_valid_prefix_code(code_list):
        return false
    get char_to_code_map, char_to_frequency_map from the input code system
    huffman_tree = construct_huffman_tree(char_to_code_map, char_to_frequency_map)
    let len_huffman_code be the code length from applying the huffman tree to the input string
    let len_code_system be the code length from applying the code system to the input string
    if len_code_system <= len_huffman_code:
        return true
    else:
        return false
```

4. Algorithm Analysis

In out approach, we first make sure that the code system is a valid prefix code system and then compare the coding efficiency of the code system against the huffman coding to see if it is optimal by the definition of the problem, which is, "The optimal solution is not necessarily generated by Huffman algorithm. Any prefix code with code length being optimal is considered correct."[2]

In procedure "is_valid_prefix_code", we have to construct and iterate the binary tree of the code system. In the worst case, the tree will degenerate to a linked list and each iteration/insertion would have to iterate through the whole list. As a result, this procedure has $\theta(n^2)$ time complexity.

In procedure "construct_huffman_tree", it takes us $\theta(n)$ time to construct the forest. After that, we use priority_queue of stl to store and sort the trees, which will take $\theta(nlog(n))$ time.

After that we use dfs to iterate through the HuffmanTree and store the character and its coding in a unordered_map(hashmap), the iterate takes $\theta(num\ of\ the\ nodes\ in\ huffman\ tree)$ time.

The overall time complexity of this algorithm is $\theta(n^2)$

5. Experiment

You can see that our program will fail fast/dropout if the prefix validation check fail. We think it is "unfair" to compare two test case if one of them fail fast and the other doesn't. So, we make sure that the fail fast wouldn't happen in our performance test, by making every character has different 8-bit encoding and impossible to overlap in the prefix.

```
// 1. build the test case
vector<char> theChars;
vector<unsigned long> theFrequencies;
vector<string> theCodes;
unsigned char theCode= -1;
unsigned long theFrequency = 0;

for (char i = '0'; i <= '9'; i++) {
    theChars.push_back(i);
    theFrequencies.push_back(theFrequency);
    theCodes.push_back(convertToStringCode(theCode));
    theCode--;
    theFrequency++;
}

// ......

string convertToStringCode(unsigned char theCode) {
    string ret = "";

    while (theCode != 0) {
        if ((theCode & 1) == 1) {
            ret = "1" + ret;
        } else {
            ret = "0" + ret;
        }
        theCode = theCode >> 1;
    }

    while (ret.size() < 8) {
        ret = "0" + ret;
    }

    return ret;
}
```

We increase the input scale(the number of different character) from 2 to 63. Within each input scale, we run our program 10000 times and combine the total time.

We use the "chrono::high_resolution_clock" to get the delta time with high precision.

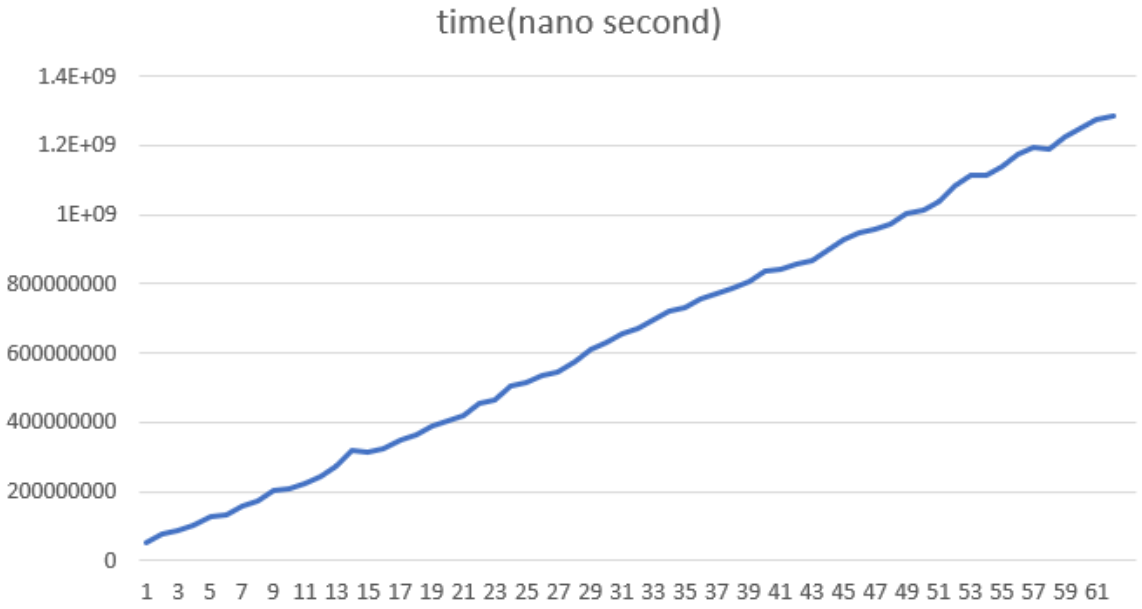
```
for (unsigned int i = 1; i < MAX_CHAR_NUM; i++) {
    chrono::time_point<chrono::system_clock, chrono::nanoseconds>  begin_time, end_time;
    // ...... generate the test data
    writeTestData(dataFile, i, codeToCheck, currentChars, currentFrequencies);
    // run the program and test
    begin_time = std::chrono::high_resolution_clock::now();
    for (int time = 0; time < DEFAULT_TEST_TIMES; time++) {
        assert(!isCodeLegal(codeToCheck, currentChars, currentFrequencies));
    }
    end_time = std::chrono::high_resolution_clock::now();
    long long deltaTime = (end_time- begin_time).count();

    // write test result and test data to the output file
    resultFile << i << ";" << deltaTime << endl;
}
```

We ran our test on a machine with

- Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz
- 16.0 GB RAM

And we get the result

input size(number of character)	running time(ms)
3	74
13	243
23	457
33	672
43	859
53	1083
63	1288



6. Discussion and Conclusion

In this project, we implement an algorithm to generate a Huffman Tree of some string and check if some encoding is as optimal as huffman coding.

In our implementation, the procedure of checking if the code system is valid prefix code system is the main drawback of our algorithm. We think we can optimize this by:

1. Add another filter method before this check. If the code length of the input code system is not isomorphic with any huffmanCode, drop out right away.
2. Iterate and construct the binary coding tree at the same time, no need to iterate twice.

However, these can't change the $\theta(n^2)$ complexity substantially.

In construction huffman tree procedure, we can replace the priority queue with two plain queue and a sort operation. But we didn't adopt this approach, because the complexity of it is also $\theta(n^2)$ and it will make our code very

disgusting.

```
// We don't want this in our code
queue<HuffmanTree*> queue1;
queue<HuffmanTree*> queue2;

for (auto & huffmanTree : huffmanTrees) {
    queue1.push(&huffmanTree);
}

while (queue1.size() + queue2.size() > 1) {
    HuffmanTree *tree1 = nullptr;
    HuffmanTree *tree2 = nullptr;
    if (queue1.empty()) {
        tree1 = queue2.front();
        queue2.pop();
        tree2 = queue2.front();
        queue2.pop();
    } else if (queue2.empty()) {
        tree1 = queue1.front();
        queue1.pop();
        tree2 = queue1.front();
        queue1.pop();
    } else {
        if (queue1.front()->getTreeWeight() < queue2.front()->getTreeWeight()) {
            tree1 = queue1.front();
            queue1.pop();
        } else {
            tree1 = queue2.front();
            queue2.pop();
        }
        if (queue1.empty()) {
            tree2 = queue2.front();
            queue2.pop();
        } else if (queue2.empty()) {
            tree2 = queue1.front();
            queue1.pop();
        } else {
            if (queue1.front()->getTreeWeight() < queue2.front()->getTreeWeight()) {
                tree2 = queue1.front();
                queue1.pop();
            } else {
                tree2 = queue2.front();
                queue2.pop();
            }
        }
    }
    queue2.push(tree1->merge(tree2));
}
```

7. References

1. https://en.wikipedia.org/wiki/Prefix_code
2. https://pintia.cn/problem-sets/1512345633586028544