ADS_Lab0_Haffman Compression

1. Realization of Huffman Tree

1.1 Members

Node Structure: Represents a single node in the Huffman tree.

- `data`: The character or string being encoded.
- frequency`: The occurrence count of the data.
- `left`: Pointer to the left child node.
- `right`: Pointer to the right child node.

hfTree Class: Implements the Huffman tree.

- `root`: A pointer to the root node of the Huffman tree.
- `hfTree Constructor`: Builds the Huffman tree based on the input string and encoding option.
- `getCodingTable`: Generates a map of data to their Huffman codes.

```cpp
class hfTree
{
private:
    struct Node
    {
        std::string data;
        int frequency;
        Node *left;
        Node *right;
        Node(){};
        std::string getMinData() const {}
        static bool compare(const Node* a, const Node* b){};
    };

    Node *root;

public:
    hfTree(const std::string &text, const Option op);
    std::map<std::string, std::string> getCodingTable();
};
```

1.2 Huffman Tree Construction Process:

1. **Frequency Calculation**: Depending on the `Option` (SingleChar or MultiChar), the constructor calculates the frequency of each character or pair of characters in the input text.
2. **Node Creation**: For each unique piece of data, a `Node` is created with the corresponding frequency and added to a `deque` (forest) of nodes.
3. **Tree Building**: Nodes are repeatedly sorted by frequency (and lexicographically if frequencies are equal) using the `compare` function. The two nodes with the lowest frequency are removed from the forest, combined into a new node (with their frequencies summed and the two nodes as children), and the new node is added back to the forest.
4. **Iteration**: This process continues until there is only one node left in the forest, which becomes the root of the Huffman tree.

1.3Functional Method Implementation:

- **getCodingTable**: This method generates a map representing the Huffman encoding table. It uses a breadth-first traversal of the tree starting from the root. For each leaf node encountered, it records the path taken as a binary string (0 for left, 1 for right) and associates it with the node's data in the coding table.

1.4 MultiChar Process

1. **Character Pair Frequency Calculation**: The constructor starts by initializing two maps, `singleCount` and `multiCount`. The `singleCount` map is used to keep track of single characters, while `multiCount` is for pairs of characters. It then iterates over the input text, updating the `multiCount` map with the frequency of each character pair.
2. **Selecting Top Combinations**: A vector of pairs, `topCombinations`, is created from the `multiCount` map and sorted by frequency (and lexicographically if frequencies are equal). The top combinations (up to a limit, e.g., 3) are then selected to be encoded as multi-char units.
3. **Updating Single Character Frequencies**: The code iterates over the text again, updating frequencies in `singleCount` based on whether a current character pair is in the top combinations. If a character pair is found in `singleCount`, its frequency is incremented and the loop skips the next character (since the pair is treated as a single unit). Otherwise, it increments the frequency of the single character.

4. **Removing Zero Frequency Entries**: Any entries in `singleCount` with a frequency of zero are removed. This step ensures that only relevant characters and character pairs are considered for tree construction.
5. **Node Creation**: For each entry in the `singleCount` map, a new `Node` is created with the character or character pair as data and the corresponding frequency. These nodes are added to the `forest` deque.
6. **Tree Building**: Similar to the `Option::SingleChar` process, nodes in the `forest` are sorted and combined into a Huffman tree. The two nodes with the lowest frequency are merged into a new node, which is then added back to the forest. This is repeated until only one node remains, which becomes the root of the Huffman tree.
7. **Root Assignment**: The last node in the `forest` deque is assigned as the `root` of the Huffman tree.

2. Analysis of Extra 3 files

To study the efficiency differences between two compression methods, I additionally selected three sets of txt documents: taming.txt, merry.txt, and shakespeare.txt. Here are the experimental results:

file name	bytes of origin	bytes of sinzip	compression rate	bytes of mulzip	compression rate	rate diff
shakespeare	647	374	57.81%	372	57.50%	0.31%
merry	17,564	10,999	62.62%	10,917	62.16%	0.47%
taming	340,912	204,347	59.94%	203,750	59.77%	0.18%

Overall, whether in small-scale or large-scale texts, the compression rate of multi zip is slightly better than that of single zip, but the difference is not significant, all within one percentage point. However, it can be foreseen that such dictionary expansion optimization methods have great potential. Choosing the appropriate dictionary size may improve the text compression rate to some extent.

3. A Possible More Efficient Method

I believe that in text compression, it is possible to expand the dictionary scope, not limiting it to two characters but considering more characters. For instance, constructing probability tables for 1 to 10 characters, selecting appropriate strategies to analyze and extract information reflected in the probability tables, and capturing long repetitive structures such as commonly used vocabulary, roots, and affixes to enhance compression efficiency. This approach is similar to block coding, but it requires more flexibility than block coding's fixed segmentation method. Instead, it adopts a dynamic segmentation approach, necessitating a more complex generation algorithm. This compression concept is expected to yield excellent results for large files and can more effectively utilize the statistical characteristics present in the data.