Lab 0: 哈夫曼压缩

赵楷越 522031910803

1 哈夫曼树的实现方式

1.1 成员变量定义

在所给代码的基础上,本次实现定义了私有变量 hfTreeNode 结构体用于表示哈夫曼树的节点,并定义了 hfTreeNode 类型指针用于保存构建好的哈夫曼树的根节点。最后, 定义了一个 string 映射的 map 容器 codingTable 用于保存生成的编码表。在公有函数中, 也定义了 generatecodingtable 函数根据生成的哈夫曼树递归得到哈夫曼编码表。对于不同的编码方式 Option,分别用 singlebuild 和 multibuild 方法构建哈夫曼树。具体代码如下图所示。

```
11  class hfTree{
12  private:
13      struct hfTreeNode{ //define the structure of the node
14          std::string data;
15          int weight;
16          hfTreeNode *left;
17          hfTreeNode *right;
18          hfTreeNode(std::string data,int weight) : data(data), weight(weight), left(nullptr), right(nullptr) {}
19      };
20      hfTreeNode *root; //the root of the tree
21      std::map<std::string, std::string> codingTable;
22
23  public:
24      enum class Option
25      {
26          SingleChar,
27          MultiChar
28      };
29      void generatecodingtable(const hfTreeNode *node, std::string code);
30      ~hfTree();
31      void deleteNode(hfTreeNode *node);
32      void singlebuild(const std::string &text);
33      void multibuild(const std::string &text);
34      hfTree(const std::string &text, const Option op);
35      std::map<std::string, std::string> getCodingTable();
36  };
```

1.2 哈夫曼树的建树过程

```
3   hfTree::hfTree(const std::string &text, const Option op){
4       switch (op){
5       case Option::SingleChar:
6           singlebuild(text);
7           break;
8       case Option::MultiChar:
9           multibuild(text);
10          break;
11      default:
12          break;
13      }
14  }
```

如上图所示，在哈夫曼树构造函数 hfTree 中，我们首先根据不同的 Option 值，选择不同的建立哈夫曼树的方法，分别为如下的单个字符编码方式和考虑多个字符编码方式两种。

1.2.1 单个字符编码

```
31  void hfTree::singlebuild(const std::string &text){
32      std::map<std::string, int> freqMap;    // Build frequency map
33      for (int i = 0; i < text.size(); i++){
34          std::string key = text.substr(i, 1);
35          freqMap[key]++;
36      }
37      struct cmp{ // Custom comparator for priority queue
38          bool operator()(hfTreeNode *a, hfTreeNode *b){
39              if (a->weight == b->weight)
40                  return a->data > b->data;
41              else
42                  return a->weight > b->weight;
43          }
44      };
45      std::priority_queue<hfTreeNode *, std::vector<hfTreeNode *>, cmp> nodeQueue;
46      for (const auto &pair : freqMap){
47          nodeQueue.push(new hfTreeNode(pair.first, pair.second));
48      }
49      while (nodeQueue.size() > 1){ // Build huffman tree
50          hfTreeNode *left = nodeQueue.top();
51          nodeQueue.pop();
52          hfTreeNode *right = nodeQueue.top();
53          nodeQueue.pop();
54          hfTreeNode *parent = new hfTreeNode(min(left->data, right->data), left->weight + right->weight);
55          parent->left = left;
56          parent->right = right;
57          nodeQueue.push(parent);
58      }
59      root = nodeQueue.top();
60      generatecodingtable(root, ""); // Generate coding table
61  }
```

在单个字符编码的情况下，本次实现构建了一个基本的哈夫曼树。

如上图所示，我们先按单个字符读取输入的字符串文本，并用 freqMap 映射存储从该输入文本中得到的频率表，保存了每一个字符在该输入文本中出现的次数。

接着，为了达到哈夫曼树构建过程中，能够快速取出剩余节点中权值最小的两个节点的效果，此处引入了一个带有自定义排序方法 cmp 的优先级队列 nodeQueue，保存的是 hfTreeNode 的指针。其中，自定义比较器 cmp 也保证了当节点的权值相同时，最先从队列中取出的节点是字典序最小的节点。这样，我们就能根据频率表 freqMap 中的映射关系，创建新的 hfTreeNode 空间，并将其指针存入 nodeQueue。

最后，我们便能根据哈夫曼树的构建逻辑，直观方便地构建哈夫曼树。每次先取出两个权值最小的节点，再构建一个权值等于两节点之和的父节点，最后将父节点指针存入 nodeQueue，循环直到 nodeQueue 中只剩下唯一一个节点。最后一个节点便代表了此棵哈夫曼树的根节点，将其保存到 root 中，便完成了单个字符编码情况下的哈夫曼树构建。其中，要注意的是当创建父节点的时候，其字符串值应取为两个子节点中字符串值字典序较小的那一个，否则会导致创建的哈夫曼树与预期结果不符。

1.2.2 考虑多个字符编码

```cpp
void hfTree::multibuild(const std::string &text){
    std::map<std::string, int> multiCharFreqMap; // consider multi-char
    for (int i = 0; i < text.size() - 1; i++){
        std::string key = text.substr(i, 2);
        multiCharFreqMap[key]++;
    }
    struct cmp4Multi{
        bool operator()(hfTreeNode *a, hfTreeNode *b){
            if (a->weight == b->weight)
                return a->data > b->data;
            else
                return a->weight < b->weight;
        }
    };
    std::priority_queue<hfTreeNode *, std::vector<hfTreeNode *>, cmp4Multi> multiQueue;
    for (const auto &pair : multiCharFreqMap){
        if (pair.second >= 1)
            multiQueue.push(new hfTreeNode(pair.first, pair.second));
    }
    std::set<std::string> multiCharSet; // Get the top 3 most frequent multi-char
    int cnt = 3;
    while (multiQueue.size() >= 1 && cnt >= 1){
        std::string multiNode = multiQueue.top()->data;
        multiQueue.pop();
        multiCharSet.insert(multiNode);
        cnt--;
    }
    // Build frequency map
    std::map<std::string, int> freqMap;
    for (int i = 0; i < text.size(); i++){
        if(i < text.size() - 1){
            std::string key = text.substr(i, 2);
            if(multiCharSet.find(key) != multiCharSet.end()){
                freqMap[key]++;
                i++;
                continue;
            }
        }
        std::string key = text.substr(i, 1);
        freqMap[key]++;
    }
    /* ... */
```

multibuild 函数基于 singlebuild 函数的基础，对文本进行了多个字符编码的处理，如上图所示。其余部分代码与 singlebuild 函数相同。

为了处理多个字符编码的情况，首先构建了频率表 multiCharFreqMap 存储输入文本中连续两个字符的出现次数。接着，为了得到出现频率最大的 k 个字符组合（此处为 3 个），类似地采用了优先级队列 multiQueue 存储 hfTreeNode 指针。此处的比较器 cmp4Multi 也保证了我们先取出频率最大的字符组合，并在频率权值相同时，取出字典序较小的字符组合。

然后，为了便于取出排序前三的字符组合（不足 3 个时只取 2 个），我们定义了 multiCharSet 存储对应的字符组合。最后，重新扫描一遍输入文本，和单个字符的情况不同，我们优先根据 multiCharSet 中的字符组合计算频率表，如果当前没有扫描到该组合，才记录此处单个字符的频率。

1.3 生成编码表

```cpp
void hfTree::generatecodingtable(const hfTreeNode *node, std::string code){
    if(node == nullptr) return;
    if (node->left == nullptr && node->right == nullptr){
        codingTable[node->data] = code;
        return;
    } // dfs to generate coding table
    generatecodingtable(node->left, code + "0");
    generatecodingtable(node->right, code + "1");
}
```

如上图所示，构建编码表的方式可以采取递归实现的方法。不断去遍历节点，模拟哈夫曼编码的实际情况：从根节点出发，左子节点的编码值加上字符 0，右子节点的编码值加上字符 1，当遍历至叶节点的时候，便能得到单个字符对应的哈夫曼编码值，并将该字符或字符组合与编码的对应关系存到编码表 codingTable 中。当递归结束，编码表便完成了生成并存储在了 hfTree 类中。

```
145  std::map<std::string, std::string> hfTree::getCodingTable(){
146      generatecodingtable(root, "");
147      return codingTable;
148  }
```

最后，程序可以调用 getCodingTable 函数返回给外界构建好的编码表。

2 额外实验结果

根据相关资料的搜索，本次实验使用了命令“tr -dc ”A-Za-z 0-9” < /dev/urandom | fold -w100|head -n NUM > bigfile.txt”在 linux 环境下生成随机的文本，用于进行不同规模下两种编码方式压缩效果的测试。我们分别令命令中 NUM=10，100，1000，5000，得到了不同大小的初始文本 sample1.txt，sample2.txt，sample3.txt，sample4.txt。以下两表记录了不同编码方式的压缩效果。（压缩前，压缩后的单位为 byte）

压缩前	压缩后	压缩率
1010	764	0.756
10100	7583	0.751
101000	75758	0.750
505000	378758	0.750

Figure 1: SingleChar 压缩结果

压缩前	压缩后	压缩率
1010	762	0.754
10100	7583	0.751
101000	75830	0.751
505000	379172	0.751

Figure 2: MultiChar 压缩结果

实验发现，在随机文本的大小不同的情况下，两种编码方式对该方式生成的随机文本的压缩效果没有显著差别，最终压缩率均在 0.75 左右。分析可能原因是因为在文本规模足够大时，只取前三个频率最高的字符组合的优化方式对压缩结果的影响相对来说足够小。

3 可能更优的压缩策略

根据先前的实验结果，我们修改了程序中 cnt 的值为 100，意义为取出前 100 个频率最高的字符组合进行压缩，并重新对 sample1.txt 进行压缩，得到的压缩后文件大小为 727bytes，压缩效果优于原有的 764bytes，此时字符组合编码的影响就对压缩效果足够大了。但要注意的是，这种方法也会导致压缩的效率降低，因此我认为可以根据不同文本规模的大小，和其中连续字符重复出现的频率数，动态地去调节选择加入编码的字符组合的个数，这样可能会是一种更优的压缩策略。