中期文档

问题解答

问题 1: 哈夫曼编码算法是否总能保证最优压缩？如果不一定，请举例说明在哪些情况下它可能不是最优的。

哈夫曼编码算法在大多数情况下能保证最优压缩，但在某些情况下可能不是最优的。例如，当字符频率分布非常不均匀时，哈夫曼编码可能不是最优的。

问题 2: 如何根据文件的字节流，构建哈夫曼树?

- 1. 统计每个字节的出现频率。
- 2. 创建一个优先队列，将每个字节作为一个节点插入队列，优先级为其频率。
- 3. 从队列中取出两个频率最低的节点，合并成一个新节点，新节点的频率为两个节点频率之和。
- 4. 将新节点插入队列。
- 5. 重复步骤 3 和 4，直到队列中只剩一个节点，该节点即为哈夫曼树的根节点。

问题 3: 对于一组频率已经给定的字符，如果哈夫曼树已经构建完成，是否能快速找到一个新的字符的编码，而不重新构建整棵树？如何实现?

可以通过调整现有的哈夫曼树来快速找到新字符的编码。具体实现方法是：

- 1. 将新字符作为一个新节点插入优先队列。
- 2. 从队列中取出两个频率最低的节点，合并成一个新节点。
- 3. 重复上述步骤，直到队列中只剩一个节点。

问题 4: 假设对一个文件进行了哈夫曼编码压缩，如何通过编码表和压缩后的字节流，准确还原出原始文件内容?

- 1. 根据编码表构建哈夫曼树。
- 2. 从压缩后的字节流中逐位读取数据，通过哈夫曼树进行解码。
- 3. 遇到叶子节点时，输出对应的字符。
- 4. 重复上述步骤，直到读取完所有数据。

问题 5: 构建哈夫曼树的过程中，如何每次高效、便捷地选出出现频率最低的两个节点?

可以使用优先队列（最小堆）来高效地选出频率最低的两个节点。插入和删除操作的时间复杂度均为 O(log n)。

问题 6: 如何完成文件夹的压缩并保留内部文件名等信息的一致性?

- 1. 遍历文件夹，记录每个文件的路径和内容。
- 2. 对每个文件进行哈夫曼编码压缩,并记录每个文件压缩的字节数。
- 3. 将压缩后的数据和文件路径一起存储，形成一个压缩包，并将每个文件的字节数写到压缩文件末尾便于查看。

- 4. 解压时，根据存储的路径信息还原文件夹结构，根据文件末尾的信息来分别读取数据进行还原。

问题 7: 如果需要对大量的小文件进行压缩，而不是单个大文件，哈夫曼编码的效率如何？是否有优化的空间?

对大量小文件进行压缩时，哈夫曼编码的效率可能较低。可以通过以下方法优化：

- 合并小文件，将所有的看做一个大文件构建哈夫曼树，然后再压缩。

项目进度简述

已完成的项目需求

- 文件的压缩和解压缩部分全部完成
- - 能够正常压缩/解压给定的 一个 非空文件
  - 能够正常压缩/解压 一个 空文件
  - 压缩时，当能够指定压缩包的名称； 解压时，能够还原出原本的文件名 ， 即便压缩包名称与文件名不
    同
- 文件夹的压缩和解压缩部分全部完成
- - 能够正常压缩/解压给定的 一个 非空文件夹
  - 能够正常压缩/解压 一个 空文件夹
  - 解压时能够还原出原本的文件名、文件夹名
- 此外，我对**压缩解压缩时间**进行了一定程度的**优化**，提高了工作效率

未完成的项目需求

- 设置压缩密码
- 文件覆盖问题
- 鲁棒性以及用户交互规则（现在用户根据命令行提示可以完成压缩解压缩操作）

接下来的工期计划

- 实现为压缩设置压缩密码
- 解决文件覆盖问题
- 解决鲁棒性并为用户设置使用参数

附录

此外主体代码如下：

压缩文件代码

```
// 压缩单个文件
void FileIO::compressFile(const string &filename, const string &outputFileName)
{
    fs::path entry(filename);
    // 对于空文件，直接写入0
    if (std::filesystem::file_size(entry) == 0)
    {
        // 读取原文件
        ifstream inputFile(filename, ios::in | ios::binary);
        // 以追加模式写入，不知道会不会有影响
        ofstream outputFile(outputFileName, ios::out | ios::binary | ios::app);

        // 写入文件头信息
        fileHead filehead;
        filehead.originBytes = 0;
        filehead.alphaVarity = 0;
        filehead.nameLength = filename.length();
        strncpy(filehead.name, filename.c_str(), sizeof(filehead.name) - 1); // 将文件名存入字符数组

        outputFile.write(reinterpret_cast<char *>(&filehead), sizeof(filehead));

        outputFile.write("\n",1);

        inputFile.close();
        outputFile.close();
    }
    else
    {
        // 读取文件并构建字符频率表，在后面可以改到前一层，把这4行作为参数传进来
        // 这样应该可以解决解压缩文件代码中的错误（好像不太行，这是独立的）
        map<char, long long> freqTable = makeCharFreq(filename);
        HuffmanTree tree(freqTable);
        tree.createHuffmanTree();
        unordered_map<char, string> charCode = tree.createHuffmanCode();

        // 写入压缩文件
        ifstream inputFile(filename, ios::in | ios::binary);
        // 以追加模式写入，不知道会不会有影响
        ofstream outputFile(outputFileName, ios::out | ios::binary | ios::app);

        // 写入文件头信息
        fileHead filehead;
        filehead.originBytes = inputFile.seekg(0, ios::end).tellg();
        inputFile.seekg(0, ios::beg);
        filehead.alphaVarity = charCode.size();
        filehead.nameLength = filename.length();
        strncpy(filehead.name, filename.c_str(), sizeof(filehead.name) - 1); // 将文件名存入字符数组
```

```
        outputFile.write(reinterpret_cast<char *>(&filehead),
sizeof(filehead));

        // 写入字符频度信息
        for (auto &entry : freqTable)
        {
            alphaCode af(entry);
            outputFile.write(reinterpret_cast<char *>(&af), sizeof(af));
        }

        // 写入主内容
        char inputBuffer[BUFFER_SIZE];
        char outputBuffer[BUFFER_SIZE];
        int outputIndex = 0;
        unsigned char bits = 0;
        int bitcount = 0;
        long long filesize = fs::file_size(filename);
        int times = filesize / BUFFER_SIZE;
        for(int i = 0; i < times; i++){
            inputFile.read(inputBuffer, BUFFER_SIZE * sizeof(char));
            for(size_t i = 0; i < BUFFER_SIZE; i++){
                string currentChar = charCode[inputBuffer[i]];
                int length = currentChar.length();
                for(size_t j = 0; j < length; j++){
                    bits <<= 1;
                    bits |= (currentChar[j] == '1');
                    bitcount++;
                    if(bitcount == 8){
                        outputBuffer[outputIndex++] = bits;
                        bits = 0;
                        bitcount = 0;
                    }
                    if(outputIndex == BUFFER_SIZE){
                        outputFile.write(outputBuffer, outputIndex);
                        outputIndex = 0;
                    }
                }
            }
        }
        // 对于不满BUFFER_SIZE的部分处理
        long long others = filesize % BUFFER_SIZE;
        inputFile.read(inputBuffer, others * sizeof(char));
        for(size_t i = 0; i < others; i++){
            string currentChar = charCode[inputBuffer[i]];
            int length = currentChar.length();
            for(size_t j = 0; j < length; j++){
                bits <<= 1;
                bits |= (currentChar[j] == '1');
                bitcount++;
                if(bitcount == 8){
                    outputBuffer[outputIndex++] = bits;
                    bits = 0;
                    bitcount = 0;
                }
                if(outputIndex == BUFFER_SIZE){
                    outputFile.write(outputBuffer, outputIndex);
```

```
                    outputIndex = 0;
                }
            }
        }
        if (!!bitcount)
        {
            // 补齐八位
            for(int i = bitcount; i < 8;i++){
                bits <<= 1;
                bits |= 0;
            }
            outputBuffer[outputIndex++] = bits;
        }
        outputFile.write(outputBuffer, outputIndex);
        inputFile.close();
        outputFile.close();
    }
}
```

解压缩文件代码

```
// 解压缩文件
streampos FileIO::decompressFile(const string &filename, string &outputFileName, long long filesize,streampos startIndex)
{
    // 读取头文件信息
    auto [filehead,currentPos] = readFileHead(filename,startIndex);

    // 恢复文件名,将输出路径更新
    string outputFilename(filehead.name, filehead.nameLength);
    outputFileName = outputFilename;
    if(filehead.originBytes == 0){
        ofstream outputFile(outputFileName, ios::out | ios::binary |ios::app);
        outputFile.close();
        return currentPos;
    }
    // 读取字符频度信息
    auto [freqTable, newPos] = readCompressTFileFreq(filename, filehead.alphaVarity,currentPos);

    // 构建哈夫曼树
    HuffmanTree tree(freqTable);
    tree.createHuffmanTree();
    // 返回子节点
    HuffmanNode *root = tree.getHuffmanRoot();
    HuffmanNode *current = root;

    ifstream inputFile(filename, ios::in | ios::binary);
    // 这个可能也要处理
    ofstream outputFile(outputFileName, ios::out | ios::binary |ios::app);
    // 定位到存储文件的位置
    inputFile.seekg(newPos);
```

```
    // 缓冲区
    char inputBuffer[BUFFER_SIZE];
    char outputBuffer[BUFFER_SIZE];
    int outputIndex = 0;
    int writeByte = 0;
    // 主题内容的字节数
    long long mainSize = filesize - (newPos - startIndex);
    int times = mainSize / BUFFER_SIZE;
    long long others = mainSize % BUFFER_SIZE;
    for(int i = 0; i < times; i++) {
        inputFile.read(inputBuffer, BUFFER_SIZE * sizeof(char));
        for (size_t i = 0; i < BUFFER_SIZE; i++)
        {
            char byte = inputBuffer[i];
            for (int j = 0; j < 8; j++)
            {
                bool bit = byte & (1 << (7 - j));
                current = bit ? current->right : current->left;
                if (!current->left && !current->right)
                {
                    outputBuffer[outputIndex++] = current->data;
                    writeByte++;
                    current = root;
                    if(outputIndex == BUFFER_SIZE ){
                        outputFile.write(outputBuffer, outputIndex);
                        outputIndex = 0;
                    }
                    // 处理多余的字符
                    if(writeByte >= filehead.originBytes){
                        goto finish;
                    }
                }
            }
        }
    }
    // 对于不满BUFFER_SIZE的部分
    inputFile.read(inputBuffer, others * sizeof(char));
    for (size_t i = 0; i < others; i++)
    {
        char byte = inputBuffer[i];
        for (int j = 0; j < 8; j++)
        {
            bool bit = byte & (1 << (7 - j));
            current = bit ? current->right : current->left;
            if (!current->left && !current->right)
            {
                outputBuffer[outputIndex++] = current->data;
                writeByte++;
                current = root;
                if(outputIndex == BUFFER_SIZE ){
                    outputFile.write(outputBuffer, outputIndex);
                    outputIndex = 0;
                }
                // 处理多余的字符
                if(writeByte >= filehead.originBytes){
                    goto finish;
```

```
                }
            }
        }
    }
finish: ;
    if(outputIndex > 0){
        outputFile.write(outputBuffer, outputIndex);
    }
    streampos nowPos = inputFile.tellg();
    // 关闭文件
    inputFile.close();
    return nowPos;
}
```

压缩文件夹代码

```
// 压缩文件夹
void Features::compressDirectory(const string &dirPath, const string &outputFileName)
{
    // 获取目录下的文件夹信息以及文件信息
    vector<string> dirname;
    vector<string> filename;

    // 将原文件夹路径完整的记录下来
    dirname.push_back(dirPath);
    // 遍历这个dirPath文件夹,将文件加入到filename，将文件夹加入到dirname,相对路径了
    // 这里存储的是 /什么什么 注意“/”
    for (const auto &entry : fs::recursive_directory_iterator(dirPath))
    {
        if (fs::is_directory(entry.path()))
        {
            dirname.push_back(entry.path().string());
        }
        else if (fs::is_regular_file(entry.path()))
        {
            filename.push_back(entry.path().string());
        }
    }

    // 打开文件开写
    ofstream output(outputFileName, ios::out | ios::app | ios::binary);
    // 写入目录名长度
    int dirnameSize = dirname.size();
    output.write(reinterpret_cast<char *>(&dirnameSize), sizeof(dirnameSize));

    // 写入目录名内容
    for (const auto &dir : dirname)
    {
        int dirLength = dir.size();
        output.write(reinterpret_cast<char *>(&dirLength), sizeof(dirLength));
        output.write(dir.c_str(), dirLength);
    }
```

```
37
38      // 写入文件名长度
39      int filenameSize = filename.size();
40      output.write(reinterpret_cast<char *>(&filenameSize),
    sizeof(filenameSize));
41
42      // 写入文件名内容
43      for (const auto &file : filename)
44      {
45          int fileLength = file.size();
46          output.write(reinterpret_cast<char *>(&fileLength),
    sizeof(fileLength));
47          output.write(file.c_str(), fileLength);
48      }
49      output.close();
50
51      // 记录每个压缩文件的大小
52      long long filesize[filenameSize];
53      // 挨个压缩文件
54      int i = 0;
55      for (const auto &file : filename)
56      {
57          long long size = compressFile(file, outputFileName);
58          filesize[i++] = size;
59      }
60
61      // 写压缩后的大小,文本文件形式打开
62      ofstream output_1(outputFileName, ios::out | ios::app);
63      output_1 << "\n";
64      for (int i = 0; i < filenameSize; i++)
65      {
66          output_1 << filesize[i] << " ";
67      }
68  }
```

解压缩文件代码

```
1   void Features::decompressDir(const string &filename)
2   {
3       ifstream inputFile(filename, ios::in | ios::binary);
4       int dirnameSize, filenameSize;
5       string path;
6
7       // 读取目录名长度
8       inputFile.read(reinterpret_cast<char *>(&dirnameSize),
    sizeof(dirnameSize));
9
10      // 读取目录名内容
11      for (int i = 0; i < dirnameSize; i++)
12      {
13          int pathLength;
14          inputFile.read(reinterpret_cast<char *>(&pathLength),
    sizeof(pathLength));
15          path.resize(pathLength);
16          inputFile.read(&path[0], pathLength);
```

```
        fs::create_directories(path); // 创建目录
    }

    // 读取文件名长度
    inputFile.read(reinterpret_cast<char *>(&filenameSize),
sizeof(filenameSize));

    // 读取文件名内容
    vector<string> filepath;
    filepath.reserve(filenameSize);
    for (int i = 0; i < filenameSize; ++i)
    {
        int fileLength;
        inputFile.read(reinterpret_cast<char *>(&fileLength),
sizeof(fileLength));
        path.resize(fileLength);
        inputFile.read(&path[0], fileLength);
        filepath.push_back(path);
    }
    streampos startIndex = inputFile.tellg();
    inputFile.close();
    // 获得每个文件的压缩文件的大小
    long long *filesize = getCompressDirSize(filename, filenameSize);
    // 对各个文件进行解压
    for (int i = 0; i < filenameSize; i++)
    {
        FileIO fileIO;
        startIndex = fileIO.decompressFile(filename, filepath[i],
filesize[i], startIndex);
    }
}
```