RELATÓRIO DO PROJETO DE COMPRESSÃO DE ALGORITMOS

Estrutura de Dados Básica II

INTEGRANTES DO GRUPO

Pedro Vinícius Barbosa Pereira

Theo Henrique da Silva Borges

Relatório do código desenvolvido para implementação de um sistema de gerenciamento de catálogo de elementos.

Relatório apresentado pelos integrantes descritos acima, os quais fazem parte do curso superior em Tecnologia da Informação da UFRN, cuja disciplina é Estrutura de Dados Básica II, lecionado pelo docente André Maurício Cunha Campos.

SUMÁRIO

1. INTRODUÇÃO

O presente relatório visa apresentar o código desenvolvido na disciplina de Estrutura de Dados Básica II. Pretende-se relatar de forma didática o processo de implementação do código referente a um algoritmo de compressão por meio do algoritmo de huffman.

a. O que é esse projeto?

Visando consolidar o estudo do assunto de árvores binárias, o projeto requer que os discentes implementem o algoritmo de huffman para comprimir um arquivo de texto, mas especializado para linguagens de programação. No caso do presente projeto, os discentes resolveram implementar para a linguagem c + +.

O algoritmo foi dividido em duas partes: A parte de gerar uma tabela de frequência baseada no arquivo analisado, e a parte de comprimir e descomprimir o arquivo.

O presente grupo resolveu fazer o código da maneira *"hard coded"*, ou seja, a tabela de frequência dos símbolos foi armazenada dentro do próprio arquivo que foi comprimido.

b. Como rodar o projeto?

Como citado anteriormente, o projeto é dividido em dois. Dentro do diretórios *"src"* há duas pastas, uma com os arquivos de cada parte do projeto: "compression" e "freqCounter".

i. Como rodar o FreqCounter

Para rodar o contador de frequência, digite na pasta root:

- ***g++ src/freqCounter/mainFreqCounter.cpp***

Isso irá gerar o arquivo executável "*a.out*". Execute esse arquivo com a seguinte linha de comando:

- ***./a.out [nomedoarquivocpp]***

Depois de chamar o executável, insira o nome do arquivo a ser analisado. O código vai imprimir a tabela de frequência no seu terminal.

ii. Como rodar o compressor

Para rodar o compressor de arquivos, digite na pasta root:

- ***g++ src/compressor/main.cpp***

Isso irá gerar o arquivo executável "*a.out*". Execute esse arquivo com a seguinte linha de comando:

- ***./a.out [nomedoarquivocpp]***

Depois de chamar o executável, insira o nome do arquivo a ser comprimido. O código escreverá um arquivo com extensão "*tp*" (personalizada), que é o seu arquivo comprimido.

2. DIFICULDADES ENCONTRADAS

A parte de descompressão do arquivo não foi realizada. Se deve ao fato de grande confusão por conta de operações com bits. A compressão foi feita com base nisso, mas mesmo que os discentes houvessem pensado que haveria como decodificar, não foram capazes de realizar tal feito. Ademais, pensar na estrutura para montar a árvore e atribuir um código binário para cada nó foi uma dificuldade, mas que foi superada pelos discentes.

3. ANÁLISE DE COMPLEXIDADE

O compressor de arquivos possui diversas funções que, neste projeto, foram feitas recursivamente. Dito isto, cabe uma análise sobre a complexidade de cada uma delas, principalmente as que envolvem árvore:

a. readFile()

Essa é a função responsável por ler o arquivo selecionado pelo usuário e gerar a tabela de frequência. Por analisar cada linha do código, a complexidade já começa com $O(L)$, onde L é a quantidade de linhas do arquivo.

A próxima operação crucial que fazemos é checar todos os elementos de WordsTable, mas este é um vetor de strings que tem tamanho constante, então desprezaremos a complexidade na busca nele, visto que será $O(1)$.

O que **não** é $O(1)$ é a linha "size_t indexOfS = line.find(s);", que busca o índice de um certo elemento no vetor, e a linha "line.erase(indexOfS, s.length());", que apaga um certo range do mesmo vetor. Ambas as operações são $O(n)$. Sabendo disso, e sabendo também que ambas as linhas de complexidade $O(n)$ são dependentes entre si, e que ambas são dependentes do código com complexidade $O(L)$, temos que a complexidade geral do código é $O(L \times N^2)$.

b. generateTree()

Essa é a função responsável por gerar a árvore com os nós que representam o algoritmo de huffman, onde suas folhas representam os símbolos e estão associados à sua frequência e ao seu código.

Como o loop while deste código é chamado $n - 1$ vezes (seja n o número de elementos na fila de prioridade), podemos desprezar a constante e dizer que a complexidade vai ser $n \times Alguma\ Coisa$.

Essa incógnita pode ser descoberta analisando a complexidade do que é analisado dentro do while: um "pop()" e um "push()" em uma fila de prioridade, que são dois códigos

de complexidade log_n. Portanto, como nenhum desses códigos dentro do "while" depende do outro, podemos dizer que a complexidade dessa função é $O(n \times log_n)$, onde n é o número de elementos na fila de prioridade.

c. codifyTree()

Essa é a função mais simples de ser analisada, visto que o objetivo dela é percorrer todos os nós da árvore, atribuindo um código para cada um deles. A partir dessa análise, basta apenas verificar se a complexidade do que é feito dentro desta busca: Verificação da existência de filhos e concatenação de strings, sendo ambas as operações de complexidade $O(1)$. Dado este fato, a função em questão é de complexidade $O(n)$.

d. compress()

Por ser a maior função, parece que essa vai ser a mais complexa, mas não é verdade! A função tem o objetivo de ler todas as linhas de um arquivo (O que já vimos que tem complexidade $O(L)$) e todos os caracteres dessa linha (Complexidade $O(M)$, sendo m a quantidade de caracteres da linha). É notório que analisar a quantidade de elementos da linha depende diretamente de analisar cada linha, pois faremos isso para todas as linhas do arquivo. Logo, é seguro dizer que a complexidade já começa sendo $O(L \times M)$.

Ademais, temos a complexidade da função "getNodeBySymbol", presente nesta função é $O(H)$, sendo H a altura da árvore. Entretanto, essa função é chamada uma vez para cada palavra da linha em questão. Com isso, temos que o trecho que procura o símbolo de cada palavra da linha tem complexidade $O(W \times H)$. Onde W é o número de palavras na linha.

Dadas todas as observações, podemos então inferir que a complexidade dessa função é $O(L \times H \times M)$.

4. ANÁLISE DE COMPRESSÃO

Por se tratar de um código que guarda a tabela de frequência no final do arquivo, há uma pequena chance (caso o arquivo seja muito pequeno) de não haver uma compressão. Entretanto, caso o arquivo seja grande, é notório que a compressão será mais eficiente do que se fosse uma codificação "hard-coded". Isso se deve pelo fato da tabela de frequência se moldar à frequência dos símbolos no próprio código, ao invés de ser baseada em arquivos externos ao que deve ser comprimido.

5. CONCLUSÃO

O algoritmo é eficiente, mas houveram dificuldades na hora de entender como implementar, especialmente na hora de fazer operações com binários. Apesar das dificuldades, é certo que o código trouxe demasiada experiência para ambos os discentes do grupo, tanto ganhando experiência com implementação e busca em árvores, quanto em operações com bits.

6. REFERÊNCIAS:

- GEEKSFORGEEKS. **Huffman Coding | Greedy Algo3**. Disponível em: <https://www.geeksforgeeks.org/dsa/huffman-coding-greedy-algo-3>. Acesso em: 15 out. 2025.
- **Estruturas de Dados: Algoritmo de Huffman**. Disponível em: <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html>.