Trabalho Pratico 3

Compactação de Arquivos de Texto

João Victor Evangelista Cruz

Universidade Federal de Minas Gerais (UFMG)

Matrícula: 2021019670

Belo Horizonte – MG – Brasil

joaovecruz@ufmg.br

1. Introdução

A situação problema desse trabalho era estar trabalhando em um jornal e esse jornal precisa cortar custos. Dessa forma, foi necessário reduzir a quantidade de memória disponível para armazenar as edições passadas do jornal. Assim, para conseguir manter as informações de todas as informações do jornal, é necessário criar uma solução utilizando o método de compactação textual conhecido como Algoritmo de Huffman.

O algoritmo de Huffman verifica quantos símbolos existem no texto a ser compactado e a frequência de cada símbolo. Após isso, cria-se uma árvore binária chamada de Árvore de Huffman, na qual, nos nós folha dessa árvore ficam os símbolos e nos nós pais, ficam a junção das frequências dos nós filhos. A partir dessa árvore, é possível atribuir códigos binários para cada símbolo e substituir cada símbolo pelo seu código. Por fim, transforma-se essa grande sequência de 0s e 1s em outros bytes, fazendo com que se gaste menos memória do que no início.

2. Método

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estruturas de Dados

A implementação do programa teve como base as estruturas de dados Árvore, Lista e Vetor.

O vetor utilizado foi um vetor de nós. Esses nós armazenavam um determinado símbolo e a frequência desse símbolo no texto. O vetor era necessário para armazenar os nós que eram criados no processo de leitura do texto.

Já a Lista era uma lista de nós e armazenava inicialmente os nós contidos no vetor. Esses nós eram inseridos ordenadamente de acordo com sua frequência, de forma que nós com menor frequência ficavam no início da lista e nós com frequência maior ficavam no final.

Por fim, a Árvore era criada através dessa lista, de forma que eram removidos da lista os dois primeiros nós, e a partir deles, formava-se uma sub-árvore, com esses dois nós sendo filhos de um outro nó criado com frequência igual à soma dos dois nós. O nó raiz dessa sub-árvore retorna a lista. Dessa forma, a árvore é criada de baixo para cima e esse processo é repetido até que sobre somente um nó e esse nó é a raiz da árvore de Huffman criada.

Essa árvore é importante para o processo, pois é a partir dela que são gerados os códigos de cada símbolo, da seguinte maneira: Contando a partir da raiz até o nó que se quer obter o código, a cada nó que passa, se foi necessário ir para a esquerda, adiciona-se um 0 no código. Se foi necessário ir para a direita, adiciona-se um 1 e faz isso até chegar no determinado nó. Por exemplo, se para chegar em um determinado nó, partindo da raiz, é necessário ir para o nó da direita, depois ir para a esquerda e por fim de novo, para a direita, o código desse nó será 101.

2.2. Classes

Foram implementadas três classes: A classe Nó, que armazena o símbolo, a frequência desse símbolo e mais três ponteiros de nós, sendo dois que auxiliam na formação das árvores e um que auxilia no funcionamento da lista. Também há a classe Lista, que promove o funcionamento das listas. E também há a classe Huffman, que possui como atributos o vetor de nós e a lista e possui a maioria dos métodos utilizados pelo programa para a compactação e descompactação dos textos.

2.3. Funções

No:

bool No::Folha(): Função que verifica se o nó é um nó folha ou não.

Além dessa função, a classe No possui três construtores e funções de Get e Set para definição e obtenção de todos os seus parâmetros.

Lista:

void Insere(No* no): Função que insere o nó na lista, com sua posição sendo definida de acordo com o atributo frequência desse nó. Quanto menor a frequência, mais perto do início estará posicionado esse nó.

No* Remove(): Função que remove o primeiro nó da lista e retorna esse nó.

bool Vazia(): Função que verifica se a lista está vazia ou não.

void Limpa(): Função que deleta todos os elementos da lista.

Além dessas funções, há o construtor e destrutor da lista, além das funções que retornam o tamanho e o primeiro elemento da lista.

Huffman

Além do construtor, a classe possui os seguintes métodos:

void Comprime(std::string arq_entrada, std::string arq_saida): Função que recebe o nome dos arquivos de entrada e saída e utiliza funções auxiliares para comprimir o texto do arquivo de entrada e armazenar no arquivo de saída, junto com as informações necessárias para descompactá-lo.

std::string Ler_Arquivo(std::string nome_arq): Função que recebe o nome do arquivo de entrada e retorna o conteúdo desse arquivo em forma de string.

void CriaVetor(std::string conteudo): Função que recebe o texto a ser compactado e preenche um vetor com a frequência de cada símbolo contido nesse texto.

void CriaLista(): Preenche a lista atributo da classe Huffman inserindo os nós presentes no vetor obtido na função acima.

No* CriaArvore(): Utiliza a lista criada na função anterior para construir uma árvore de Huffman do texto.

std::string* CriaTabela(No *raiz): Utiliza a Árvore de Huffman criada na função acima para criar um vetor de strings, contendo o código de cada símbolo.

void CriaTabelaRecursivo(std::string* tabela, No* no, std::string codigo): Função recursiva auxiliar da CriaTabela que passa pelos nós da árvore obtendo o código de cada nó.

std::string Codifica(std::string conteudo, No* arvore): Utiliza a tabela de códigos criada na última função para fazer uma string substituindo cada simbolo por seu código.

std::string StringArvore(No* arvore): Função que transforma a árvore para o formato de string, de forma que seja possível ler essa string e criar essa árvore novamente quando for descompactar o arquivo.

std::string StringArvoreRecursivo(No* no, std::string str): Função recursiva auxiliar a StringArvore que passa pelos nós da árvore complementando a string formada.

std::string CriaBytes(std::string texto_codificado): Utiliza o texto codificado na função Codifica e cria um byte a cada 8 dígitos presentes no texto codificado, criando assim um caractere. Armazena os caracteres formados em uma string e retorna essa string.

Void PreencheArquivoComprimido(std::string arvore, int tamanho, std::string texto_codificado, std::string arq_saida): Recebe a árvore em formato de string, o tamanho do texto original, o texto codificado e o nome do arquivo de saída e utiliza a função CriaBytes para criar o texto compactado e armazena a string árvore, o tamanho e o texto compactado no arquivo de saída.

void Descomprime(std::string arq_entrada, std::string arq_saida): Função que recebe o nome dos arquivos de entrada e saída e utiliza funções auxiliares para ler as informações do arquivo de entrada e descompactar o arquivo, armazenando o texto original no arquivo de saída.

No* Huffman::LeArvore(std::string arv_string): Função que lê a árvore em formato de string do arquivo do texto compactado e recria a Árvore de Huffman desse texto.

No* LeArvoreRecursivo(std::string arv_string, int* posicao): Função recursiva auxiliar a LeArvore que passa por cada caractere da string criando cada nó da árvore.

std::string CriaBits(std::string bytes): Função contrária a CriaBytes, essa função tranforma os caracteres do texto compactado nos códigos binários que elas representam, formando o texto codificado novamente.

std::string Descodifica(std::string texto_codificado, int tamanho, No* raiz): Utiliza o texto codificado obtido na função anterior, a Árvore de Huffman do texto e o tamanho do texto original para transformar o texto codificado de volta ao texto original.

void PreencheArquivoDescomprimido(std::string conteudo, int tamanho, No* arvore, std::string arq_saida): Utiliza as funções CriaBits e Descodifica e obtém o texto original e armazena ele no arquivo de saída.

3. Análise de Complexidade

A parte da compactação tem funções com as seguintes complexidades assintóticas:

Ler_Arquivo: Essa função tem complexidade O(n/k), em que n é a quantidade de caracteres do texto e k é a quantidade de símbolos / linha, do texto, já que o texto é lido de linha em linha e armazenado na string.

CriaVetor: Possui complexidade O(n), pois a função itera sobre a string que armazena o texto, caractere por caractere, e cria um novo nó ou atualiza esse nó, dependendo do caso. Ambas ações são O(1).

O(1) * O(n) = O(n)

CriaLista: A primeira parte da função possui complexidade O(128), já que a função passa pelas 128 posições do vetor, e se houver algum nó não nulo, insere na lista. A inserção da lista tem melhor caso O(1), quando a frequência do nó a ser inserido é a menor da lista. E tem pior caso quando a frequência é a maior, assim, há comparações com todos os nós presentes na lista, por isso, tem de 1 a 127 comparações, dependendo da quantidade de nós já existentes na lista, que dá uma média de 64.

Por isso, a função tem melhor caso O(128) e pior caso O(128 * 64).

Esse pior caso é muito difícil, já que é difícil ter um texto com todos os 128 caracteres diferentes da tabela ASCII. Para um texto de tamanho considerável, o caso médio, tendo 30 símbolos diferentes no texto e a 80 comparações nas inserções na lista fica aproximadamente O(90 + 30 * 80) que é aproximadamente O(2500).

CriaArvore: Tem complexidade O(p-1), em que p é a quantidade de símbolos diferentes no texto, já que cada elemento da lista se une com outro elemento, e a sub-árvore produzida volta pra lista, fazendo assim, p-1 operações. p é sempre menor ou igual a n.

CriaTabela: A função passa por todos os nós da árvore uma vez. Por isso, a complexidade é O(p + p-1) = O(2p – 1), sendo p a quantidade de símbolos diferentes, já que todos os símbolos são nós folha e a quantidade de nós pai é um a menos do que a quantidade de folha.

Codifica: Possui complexidade O(n), pois passa por todos os caracteres do texto codificando-o.

StringArvore: Assim, como a CriaTabela, ela também passa por todos os nós uma vez, por isso, também é O(2p - 1).

PreencheArquivoComprimido: É O(1), pois somente abre o arquivo e armazena a string nele.

CriaBytes: Tem complexidade O(m), em que m é a quantidade de caracteres do texto codificado, já que ela passa por todos os caracteres dessa string. O valor de m depende da quantidade de símbolos diferentes que o texto possui, e é quase sempre maior que n, só é igual a n quando o texto possui apenas 2 símbolos diferentes e m nunca é menor que n. Apesar disso, para a maioria dos textos, $m < n^2$.

Somando tudo, a complexidade da parte de compactação é O(m), que geralmente é $O(n) < O(m) < O(n^2)$.

Descompactação:

Ler_Arquivo: É O(n/k), como já foi mostrado.

Le_arvore: É O(p), sendo p a quantidade de símbolos diferentes do texto original, já que ela passa pela string reconstruindo cada nó da árvore.

CriaBits: É O(8v), em que v é o tamanho do texto compactado, já que passa por dois laços, aninhados, de 8 e v repetições. v é menor que n, já que é o texto compactado naturalmente é menor do que o texto original.

Descodifica: É O(m), em que m é a quantidade de caracteres do texto codificado, já que ela passa por todos os caracteres da string, decodificando-a.

Portanto, a descompactação também é O(m), já que m é maior que n, p, e v.

4. Estratégias de Robustez

Foi implementado no texto os seguintes mecanismos para a robustez e tolerância a falhas do programa:

No arquivo principal do programa, é pedido como argumentos de entrada dois nomes de arquivos, o arquivo de entrada e o de saída. Se não tiver –c ou –d seguidos dos nomes do arquivo de entrada e do arquivo de saída, o programa não funciona e gera uma mensagem de erro.

Além disso, se o arquivo de entrada não existir ou se não for possível abrir algum dos arquivos, o programa para de executar e também é gerado um erro.

Quando se por um acaso acontecer de tentar remover algum elemento de uma lista vazia, também aparece na tela uma mensagem de erro.

5. Análise Experimental

Foram feitos testes comprimindo arquivos de textos, de formato parecido com a maioria dos textos normais, com parágrafos, palavras e espaços, de diferentes tamanhos e foi calculado a taxa de compressão (porcentagem do tamanho do arquivo original que foi diminuída) de cada caso (Tamanho dos arquivos em bytes):

Tamanho do arquivo original	Tamanho do arquivo depois da compressão	Taxa de compressão
100	129	-
200	187	6,5%
500	368	26,4%
1000	653	34,7%
5000	2806	46,88%
10000	5.498	45,02%
50000	26.970	46,06
100000	53.717	46,29%

A partir dos resultados dos testes, é possível perceber que quando o texto é muito pequeno, tendo um tamanho menor ou igual a 100 caracteres, o arquivo comprimido fica maior do que o arquivo original, já que no arquivo de saída da compressão, é necessário armazenar informações para a descompressão dele.

A partir de cerca de 200 bytes, o arquivo formado já começa a ter tamanho menor do que o original, mas com uma taxa de compressão pequena, também pelas informações que tem que ser adicionadas no arquivo de saída.

Com o aumento do tamanho, as informações de descompressão se tornam quase insignificantes para o tamanho total do arquivo, e a taxa de compressão de arquivos de tamanhos significativos, acima de 5000 bytes, foi de cerca de 45%.

Por isso, concluímos que o programa não funciona tão bem para arquivos de texto de tamanhos pequenos. Porém, isso não é um fator tão significante, já que os textos já são pequenos, então teoricamente, eles não necessitam de serem comprimidos.

Já quando comprimimos arquivos maiores, que realmente podem precisar de serem comprimidos, obtemos boas taxas de compressão, chegando a quase metade do tamanho do arquivo original.

6. Conclusões

Nesse trabalho, foi feito um programa capaz de fazer a compressão de arquivos de texto, utilizando a linguagem C++.

Para a conclusão desse trabalho e a obtenção de uma solução, foi importante o conhecimento das estruturas de dados, principalmente a árvore e a lista, que tiveram um papel fundamental para que fosse possível executar o Algoritmo de Huffman.

Além disso, foi necessário obter conhecimentos sobre como mudar os bits das variáveis, de forma que fosse possível compactar as strings codificadas obtidas nos métodos do programa.

Com esse trabalho, fica nítido a importância da organização do código através da modularização, principalmente quando se tem códigos grandes e com várias classes, como esse, e também a utilização do makefile, para poupar tempo e esforço na hora de compilar o programa e fazer testes.

7. Bibliografia

- Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011.

- Cormen, T., Leiserson, C, Rivest R., Stein, C. Algoritmos – Teoria e Prática, 3a. Edição, Elsevier, 2012.

- GeeksForGeeks. 2023. Huffman Coding | Greedy Algo-3 Disponível em: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/ Acesso em: 22 de jun. de 2023.

8. Instruções para compilação e execução

Para a compilação do programa, deve-se utilizar o comando “make all”, no terminal, estando na raiz da pasta TP.

Para a execução, deve se colocar os arquivos de teste na pasta TP, na raiz do projeto e digitar o comando estando na raiz do projeto, no terminal:

“./bin/main –c arquivo_entrada.txt arquivo_saida”, para a compactação do arquivo de entrada e

“./bin/main –d arquivo_entrada.txt arquivo_saida” para a descompactação do arquivo de entrada.

Por exemplo, se o nome do arquivo de entrada for “ent.txt” e o arquivo de saída for “saida.txt” o comando deverá ser

./bin/main –c ent.txt saida.txt ou

./bin/main –d ent.txt saida.txt