Trabalho Prático 3
Compactação de Arquivos de Texto

Bernardo Dutra Lemos - 2022043949

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
bernardolemos@dcc.ufmg.br

● Introdução:

Um problema comum no meio de armazenamento de dados é relacionado ao espaço limitado para guardar informações, nesse cenário surge a necessidade de conseguir salvar informação ocupando a menor quantidade de espaço possível e essa é a temática deste trabalho.

A ideia principal do trabalho é realizar a compactação de arquivos de texto. Nesse contexto, o método que será usado é conhecido como Algoritmo Guloso de Huffman e é dito ser um algoritmo guloso, pois o código relacionado a cada caractere depende da frequência de ocorrência dele no arquivo de entrada.

O algoritmo de Huffman consiste em gerar códigos para cada caractere do arquivo de entrada com base na frequência que ele aparece no conteúdo. De forma que os mais recorrentes possuem um código menor, ou seja, que ocupe menos espaço e os menos recorrentes ocupem mais bits.

● Método:

A realização deste trabalho foi dividida em três partes, sendo elas: o entendimento do algoritmo de huffman, a implementação das estruturas necessárias e, por fim, a implementação do algoritmo de Huffman em si.

Durante as pesquisas sobre esse método de compreensão de arquivos foram encontradas diversas formas de implementação, mas a forma escolhida é a forma como é ensinada no livro do Cormen, que é utilizando uma fila de prioridade mínima. Diante disso, o primeiro passo realizado foi implementar a classe de uma fila de prioridade mínima que será descrita a seguir.

- **MinHeap:**
  Tipo abstrato de dados, criado utilizando polimorfismo paramétrico, que possui métodos de inserção(*Insert*), extração do valor mínimo(*ExtractMin*), impressão e de obter o tamanho, também há métodos privados, mas eles são usados apenas para manter a

propriedade da fila de prioridade. O construtor dessa classe recebe um vetor qualquer e seu tamanho e constrói um heap a partir desse vetor.

Após resolver a questão da fila de prioridade surgiu a questão de como seria a forma que a compactação seria realizada, e foi decidido que seria no nível de letras. Com isso surgiu a questão de como essas letras seriam armazenadas. Para isso foi criado um tipo abstrato de dados chamado “Node” que possuiria quatro atributos privados, sendo eles o char que representa aquele nó, a frequência com que ele aparece no arquivo de entrada e dois ponteiros. Esses ponteiros são necessários, pois durante a montagem do algoritmo de Huffman é criada uma árvore baseada nesses nós. Abaixo segue uma descrição do TAD “*node*”.

- **Node:**
  Tipo abstrato de dados, criado para representar o nó da árvore de Huffman, possui diversos métodos, como *setLeft* e *setRight,* que recebem ponteiros de nós e definem eles para serem nós filhos, também a métodos que definem a letra e a frequência. Também há métodos que retornam os atributos privados, como nós filhos, letra e frequência. Para tornar o processo de comparação entre elementos dessa classe mais fácil, também houve a sobrecarga dos operadores de igualdade(==), menor(<), maior(>) e de escrita em outputs(<<).

Com os dois TADs descritos acima prontos, os pré-requisitos para implementar o algoritmo de Huffman estão completos. A classe referente a esse código possui diversos métodos e todos eles serão abordados logo abaixo.

- **Huffman:**
  A classe, além do construtor, possui dois métodos públicos, um para realizar a codificação(encode) e outro para decodificação(decode), a forma como os dois funcionam serão descritas e durante a explicação os métodos privados serão citados.

  Encode: Recebe duas strings referentes ao nome do arquivo de entrada e de saída, respectivamente. Lê o arquivo de entrada e salva a quantidade de ocorrências de cada char em um vetor de forma constante(*hashing*) e, ao mesmo tempo, salva o texto do arquivo em uma string. Após esse processo, é criado uma fila de prioridade mínima a partir do vetor de ocorrências e esse heap é passado como parâmetro para a função de criação da árvore de Huffman(*build_tree*), essa é salva internamente na classe. Após esse processo, com o

auxílio da função chamada “*fulfill_table*” e “*get_code*”, o código associado a cada caracter é salvo em um vetor de strings, esse processo é essencial para aumento na velocidade do programa, pois cada código pode ser acessado de forma constante na hora de codificar o texto ao invés de realizar n, sendo n o número de caracteres, caminhamentos na árvore de Huffman. Para finalizar, é chamada a função “*code*”, que retorna o conteúdo do arquivo de texto codificado e essa string de retorno é passada para a função “*write_bits*” que realiza a escrita em bits no arquivo de saída, além disso é criado um arquivo chamado “NomeSaida+Table.txt”, que armazena o tabela de ocorrência de cada char, arquivo este que **deve** ser usado para construir a árvore de Huffman na decodificação. No final da execução é imprimido o tempo gasto pela operação.

Decode: Recebe duas strings referentes ao nome do arquivo codificado e de saída. Primeiro é realizada a leitura do arquivo da tabela, que é criado na compactação, e é criado a fila de prioridade mínima e em seguida a criação da árvore de Huffman. Após isso o arquivo codificado é aberto e passado para a função “*read_bits*” que lê o arquivo bit a bit e devolve uma string de 0s e 1s que é a mensagem codificada. Para finalizar essa string obtida no passo anterior é passada para a função *decodedstring,* que realiza a decodificação do código compactado e retorna o texto original. Para finalizar, o texto original é escrito no arquivo de saída indicado. No final da execução é imprimido o tempo gasto pela operação.

O construtor da classe possui duas possibilidades de funcionamento, decodificar e codificar e deve ser passada como parâmetro(c ou d). Além disso, é importante ressaltar que para alguns casos “sobravam” alguns bits no último byte de compactação e eles adicionavam algumas letras a mais no final do arquivo, e a forma como isso foi resolvido foi adicionando um char extra no começo do arquivo codificado para saber quantos bits extras o código possuía.

● Análise de Complexidade:

De acordo com o capítulo 6 do livro “Algoritmos - Teoria e prática” de Thomas Cormen, a construção de um fila de prioridade mínima é realizada em tempo O(n), enquanto todas as operações são realizadas em tempo O(log(n)). Também de acordo com o mesmo livro do Cormen. A algoritmo de Huffman possui um tempo de execução de O(n*log(n)), sendo n o número de

caracteres. Mas o programa sempre considera o número de caracteres distintos sendo os 128 da tabela ASCII, portanto para realizar a montagem da árvore de Huffman sempre são necessários um número constante de passos, independente do tamanho da entrada.

Outro ponto importante para analisar a complexidade total do programa é considerar que para saber a quantidade de ocorrências de cada uma dos 128 possíveis caracteres é necessário ler o arquivo de entrada uma vez e para codificar é necessário ler mais uma vez, logo o processo de codificação é na ordem de O(n), sendo n o número de caracteres presente no arquivo de entrada. Uma melhoria realizada para melhorar o tempo de codificação foi que uma vez gerado a árvore os códigos referentes a cada caractere não mudam, portanto podem ser salvos em um vetor na posição correspondente ao valor ASCII dele. Tornando assim, o acesso do código em tempo constante ao invés de gerar o código percorrendo a árvore n vezes, o que aumentaria a complexidade de O(n) para O(n*log(n)).

Já no processo de decodificação é preciso montar a árvore de Huffman, que já foi dito anteriormente gastar um tempo constante e além disso é necessário percorrer o arquivo codificado uma única vez e enquanto caminha na árvore de Huffman para cada bit lido. Logo, para cada caracter presente no arquivo original é preciso realizar a busca dele na árvore, que leva em média uma complexidade de O(log(n)), portanto temos que para decodificar é necessário O(n*log(n)).

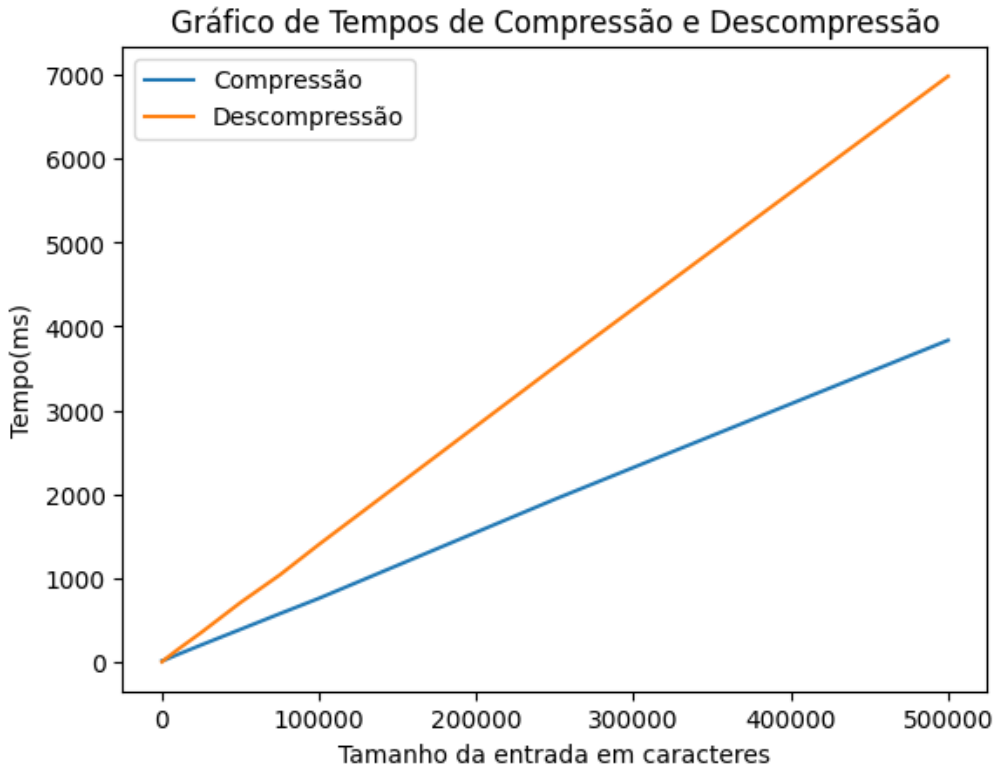
- ## Estratégias de Robustez:

O programa em si não abre margem para muitos erros relacionados à entrada, considerando que os caracteres especificados foram apenas os 128 da tabela ASCII não existem muitos possíveis erros a serem tratados. Portanto os erros tratados foram em relação a erros de leitura de arquivos e a opção escolhida entre codificar ou decodificar, ambos passados por linha de comando.

Caso nenhuma das opções escolhidas sejam válidas ou nenhum arquivo seja passado, é lançado um erro de argumentos inválidos. Caso o arquivo de entrada não seja encontrado ou tenha tido algum problema para ser aberto, é lançado um erro de arquivo não encontrado. Além disso, é considerado o caso em que é feita a tentativa de descompactar um arquivo que não foi codificado. A forma como esse erro é identificado é que durante a compactação é criado um arquivo que contém a tabela de frequências, caso esse arquivo não seja encontrado pelo programa, é lançado um erro dizendo que a tabela de ocorrências não foi encontrada. Caso o processo de codificação/decodificação ocorra sem problemas, é mostrado uma mensagem dizendo que a operação foi completada com sucesso.

- **Análise Experimental:**

As análises de tempo foram feitas utilizando um comando do linux que gera um arquivo de texto aleatório utilizando todos os caracteres de A-Z, a-z e 0-9. O comando em questão é “*tr -dc "A-Za-z 0-9" < /dev/urandom | fold -w100|head -n 10000 > bigfile.txt”*, onde o valor depois de -n é o número de caracteres que o arquivo criado vai possuir.



Os resultados de tempo são exatamente o que foi comentado na seção de análise de complexidade, que é um tempo linear para a compactação e um tempo na ordem de O(n*log(n)) para a descompactação.

Quanto a taxa de compressão, ela não tem como ser especificada exatamente, pois depende muito do arquivo de entrada, caso seja um arquivo que repete várias vezes os mesmos caracteres é esperado que a compactação tenha uma taxa melhor, entretanto quanto mais caracteres possuírem uma taxa de frequência parecida, pior vai ser a compactação, mas em média o valor mínimo do peso arquivo compactado é de 70% do peso original.

- ## Conclusões:

O trabalho inteiro consistia em escrever um programa que tivesse suas opções de execução, compactar ou descompactar, e o método de compactação dos arquivos deveria ser o algoritmo de Huffman. E para descompactar um arquivo, ele deve ser compactado pelo próprio programa anteriormente e a tabela gerada durante esse processo é essencial para a descompressão do arquivo.

Se mostrou um projeto bastante desafiador, principalmente na parte da escrever e ler os bits que demandou bastante pesquisa, e também exigiu que uma estrutura de dados muito útil fosse feita, a fila de prioridade mínima. Além de mostrar um método de compactação muito útil. No geral foi um trabalho que agregou bastante conhecimento e se utilizou de vários conhecimentos adquiridos ao longo da disciplina.

- ## Bibliografia:

  - CORMEM, Thomas. Algoritmos - Teoria e Prática. 3.ed. GEN LTC, 2012
  - Disponível em:<https://stackoverflow.com/questions/1856514/writing-files-in-bit-form-to-a-file-in-c?rq=1> Acesso em 30 de Junho de 2023
  - Disponível em:<https://stackoverflow.com/questions/50224823/c-how-to-read-in-a-single-bit-from-a-file> Acesso em 30 de Junho de 2023

- ## Compilação:

1. Abrir o terminal
2. Entrar na pasta tp
Comando: cd / <caminho para a pasta /TP2 >
3. Compilar o programa com o Makefile disponibilizado na pasta
Comando: make
4. Executar o programa
Compactar:
Comando: ./bin/main -c <arquivo de entrada> <arquivo de saída>
Descompactar
Comando: ./bin/main -d <arquivo compactado> <arquivo de saída>
5. Após terminar de utilizar o programa, delete os arquivos desnecessários
Comando: make clean