Trabalho Prático

Compactação de Arquivos de Texto

Thales Augusto Rocha Fernandes - 2022043825

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

1. Introdução

Nesse trabalho é tratado o problema de compactação e de descompactação de arquivos através do algoritmo de Huffman. Esse algoritmo analisa a frequência das palavras em um dado texto e atribui uma codificação curta para os símbolos mais frequentes e uma codificação mais longa para símbolos menos frequentes, de maneira que há uma maior economia de espaço na codificação. Dessa maneira, o objetivo desse trabalho é criar um código em linguagem C/C++ que recebe como entrada dois nomes de arquivos, em que o primeiro é o nome do arquivo que iremos compactar ou descompactar e o segundo é o nome do arquivo em que essa compactação ou descompactação será armazenada, de forma que esses arquivos são de texto e estão codificados como UTF-8. O código desse trabalho realiza a compactação e descompactação com base nas palavras, de maneira que todas as palavras do texto possuem um código binário codificado. Essa implementação dos algoritmos não fará uso de estruturas de dados provenientes de bibliotecas específicas da linguagem C++. Assim, a representação da estrutura de dados usadas estão disponíveis no código e possuem funções e características específicas para este projeto.

2. Método

2.1.Configurações da máquina

Sistema operacional: WSL2 - Ubuntu 20.04 LTS

2.2. Estruturas de dados

Na implementação desse trabalho, as estruturas de dados usadas foram o No_Huffman, o MeuVector e a Dupla, sendo que essas três estruturas também são classes que serão melhor explicadas no tópico 2.3. Além disso, também foi criada a struct Compara_No que é uma estrutura que não possui atributos e apresenta dentro dela apenas uma função que recebe como parâmetro dois No_Huffman e compara a frequência desses No_Huffman. O motivo de usarmos uma estrutura que possui apenas uma função dentro dela é porque ela será usada dentro de outras classes.

2.3. Classes

A classe No_Huffman representa um nó da árvore de Huffman. Cada nó dessa árvore contém uma palavra do texto e a quantidade de vezes que essa palavra aparece, de forma que essa classe representa a informação da compactação do arquivo de texto. Esse No_Huffman possui 4 atributos que são uma string chamada palavra, um atributo inteiro chamado frequência, e dois ponteiros do tipo No_Huffman chamados esquerdo e direito que irão apontar para os nós esquerdo e direito. A única operação que essa classe realiza é a de inicialização em que atribuímos valores aos seus atributos.

A classe MeuVector é uma implementação de um vetor dinâmico de itens do tipo No_Huffman que será utilizado para armazenar os nós da árvore de Huffman durante sua construção e para manter uma certa organização desses nós através de uma fila de prioridade que será melhor explicada no tópico 2.4. Essa classe possui 4 atributos privados que são um vetor de itens de tipo ponteiro de No_Huffman que será utilizado como estrutura principal para o vetor dinâmico, dois atributos do tipo size_t para guardar o tamanho e a capacidade máxima do vetor e um atributo do tipo Compara_No que é utilizado para guardar informações sobre a fila de prioridade no processo de criação da árvore de Huffman. As operações que essa classe realiza são criar um novo MeuVector, destruir um MeuVector, inserir um No_Huffman, retornar o primeiro nó do vetor, remover um item do vetor, retornar se o vetor está vazio, retornar o tamanho do vetor e realizar uma mudança de capacidade do vetor.

A classe Dupla é uma implementação de um par de valores que podem ser de qualquer tipo visto que essa classe utiliza dois templates para cada um desses valores. Essa classe possui dois atributos privados que representam o par de valores, sendo que um chama primeiro e o outro chama segundo. As operações que essa classe realiza são criar a Dupla e um operador de atribuição para a Dupla.

2.4.Funções

O código possui 5 principais funções que vamos tratar nesse tópico.

main: Essa função lê o comando que o usuário digita no terminal e ativa funções de acordo com esse comando. Se o usuário desejar compactar o arquivo ele digitará o nome do arquivo que será compactado, o nome do arquivo onde a compactação estará contida e o comando ‘-c’. Se o usuário desejar descompactar o arquivo ele digitará o nome do arquivo que será descompactado, o nome do arquivo onde a descompactação estará contida e o comando ‘-d’. Se o usuário quiser compactar o arquivo, essa função main irá chamar a função Compactar e se o usuário quiser descompactar o arquivo a função Descompactar será chamada.

Arvore_Huffman: Essa função constrói a árvore de Huffman a partir de uma tabela de frequências que é passada por parâmetro. Essa tabela de frequências contém todas as palavras contidas no texto que queremos comprimir e descomprimir e a quantidade de vezes que as palavras foram repetidas. Essas informações da tabela estão guardadas em um vetor de Duplas, de forma que o tamanho desse vetor também é passado como parâmetro pra essa função. Dentro dessa função Arvore_Huffman é criado um MeuVector, de maneira que nós criaremos um No_Huffman para todas as palavras da tabela de frequência e iremos inserir dentro desse MeuVector. Como discutido no tópico 2.3. a classe MeuVector consegue realizar a comparação de frequência entre os No_Huffman, de forma que os nós dentro dessa classe estarão ordenados pela ordem da frequência. Após inserir todos os nós dentro do MeuVector, a função Arvore_Huffman combina os nós de menor frequência até que sobre apenas um nó dentro do MeuVector, de maneira que esse nó será o nó raiz da árvore de huffman e será retornado pela função.

Codifica_Bits: Essa função cria uma tabela de mapeamento de palavras para bits 0 ou 1. Os parâmetros passados para essa função são um No_Huffman que representa a

raiz da árvore huffman, uma string vazia chamada codigo, um vetor de Duplas em que ambos os itens da Dupla são strings e um atributo size_t que representa o tamanho desse vetor de Duplas. Para criar a tabela de mapeamento a função percorre a árvore de huffman de forma recursiva, de forma que se a função for chamada recursivamente para um nó esquerdo é adicionado o caracter '0' na string codigo e se a função for chamada recursivamente para um nó direito é adicionado o caracter '1' na string codigo. Caso o nó atual da função for um nó folha, nós adicionaremos um par de valores no vetor de Duplas passado por parâmetro, de forma que a primeira string desse par contém a palavra do nó folha e a segunda string do par contém o código em binário dessa palavra. Ao percorrermos toda a árvore, teremos o código em binário de todas as palavras contidas na árvore huffman armazenadas dentro do vetor de Duplas.

Compactar: Essa função irá compactar um arquivo em outro arquivo e receberá como parâmetro duas strings contendo o nome desses arquivos. A princípio, a função Compactar abre o arquivo de entrada através de um ifstream e cria um arquivo de saída através de um ofstream. Em seguida, é criado um vetor de Duplas, em que o primeiro item é uma string e o segundo é um inteiro, que conterá a tabela de frequências que é passada por parâmetro na função Arvore_Huffman, de maneira que nós iremos percorrer o arquivo que queremos compactar adicionando as palavras e a quantidade de vezes que essas palavras se repetem dentro desse vetor de Duplas. Após criarmos essa tabela de frequências, nós criamos um ponteiro No_Huffman e atribuímos a esse ponteiro o valor retornado pela função Arvore_Huffman que conterá a raiz da árvore de huffman criado por essa função. Depois de construir a árvore de huffman a função cria um vetor de Duplas, em que os dois itens são strings, de forma que nesse vetor será armazenado o código em bits de todas as palavras contidas nos nós da árvore de huffman através da função Codifica_Bits explicada anteriormente. Agora que a codificação em bits foi feita, o código imprime no arquivo compactado todos os pares de strings do vetor de Duplas que passou pela função Codifica_Bits, de forma a imprimir todas as palavras e seus códigos no arquivo de saída. Essa impressão ocorre para que, posteriormente, seja possível descompactar o arquivo compactado. Em seguida, nós voltamos para o início do arquivo e convertemos todas as palavras do texto em caracteres comprimidos com o uso de um buffer e inserimos esses caracteres comprimidos dentro do arquivo comprimido. Após realizar essa inserção a função fecha as duas streams abertas no início e termina.

Descompactar: Essa função irá descompactar um arquivo em outro arquivo e receberá como parâmetro duas strings contendo o nome desses arquivos. A princípio, a função Descompactar abre o arquivo de entrada através de um ifstream e cria um arquivo de saída através de um ofstream. Em seguida, a função cria um vetor de Duplas, em que os dois itens são strings, de forma que nesse vetor será armazenado o código em bits de todas as palavras contidas nos nós da árvore de huffman, através da leitura do arquivo compactado, já que, como vimos na função Compactar, os pares de string que passaram pela função Codifica_Bits são imprimidos no arquivo compactado. Agora que temos a tabela de mapeamento de palavras em bits é possível decodificar o texto comprimido. Essa decodificação é realizada através de uma estrutura bitset que é proveniente da biblioteca bitset e facilita a manipulação de bits e a conversão desses bits em string. Dessa maneira, nós iremos percorrer todo o texto de bits comprimidos do arquivo comprimido, converter esses bits em tipo string

através do bitset e adicionar o resultado de cada bit em uma string chamada resultado_leitura. Depois que a string resultado_leitura contém todo o texto comprimido, nós percorremos essa string através de um loop, decodificamos cada palavra e imprimimos no arquivo de saída descomprimido. Finalmente, nós fechamos as duas streams abertas no início da função.

3. Análise de Complexidade

Apesar das funções que vamos analisar manipular, principalmente, o texto palavra por palavra, nós iremos analisar a complexidade dessas funções a partir do número de caracteres do texto, já que diferentes palavras podem ter diferentes tamanhos e pode tornar a análise imprecisa. Dessa maneira, a variável n tratada nesse tópico condiz com o número de caracteres manipulados nos arquivos de entrada. A função main possui complexidade de tempo e de espaço constantes e não será analisada.

Arvore_Huffman – Complexidade de Tempo: Essa função possui 2 loops que percorrem todas as palavras da tabela de frequência passada por parâmetro. O pior caso dessa função ocorre quando todas as palavras do arquivo de entrada são diferentes e não se repetem. Devido a esse possível caso, a complexidade de tempo da função é O(n).

Arvore_Huffman– Complexidade de Espaço: Essa função cria uma estrutura MeuVector onde é inserido todos os No_Huffman da árvore de huffman. Estamos tratando do tipo char nessa análise e, sabemos que a estrutura No_Huffman explicada no tópico 2.3. ocupa aproximadamente 50 bytes e o tipo char ocupa apenas um byte. No entanto, os nós são formados por um conjunto de bytes que são as palavras, de forma que o espaço ocupado por essa função seja uma constante multiplicado pelo número de caracteres. Dessa maneira, a complexidade de espaço da função é O(n).

Codifica_Bits – Complexidade de Tempo: Essa função é recursiva e sua parte não recursiva tem complexidade constante. Dessa maneira, a complexidade de tempo da função é O(log n).

Codifica_Bits – Complexidade de Espaço: Essa função cria uma string que contém um código em bits para todas as palavras do texto de entrada. Supondo que essa string contém uma média de caracteres igual a das palavras, o que é um caso bem provável de acontecer, a complexidade de espaço da função é O(n).

Compactar – Complexidade de Tempo: Essa função possui 4 loops, em que 3 deles percorrem todas as palavras do texto de entrada e 1 deles percorre todos os itens da tabela de códigos contido na classe Dupla. Dessa maneira, a complexidade de tempo dessa função é O(n).

Compactar – Complexidade de Espaço: Essa função cria a árvore de huffman a partir dos No_Huffman, cria a tabela de frequência a partir de um vetor da estrutura Dupla, cria os códigos em bits para as palavras a partir de um vetor da estrutura Dupla e cria uma string adicional contendo todo o texto comprimido. O tamanho gerado pela função ao todo é bem grande. Dessa maneira, a complexidade de espaço da função é O(n^2).

Descompactar – Complexidade de Tempo: Essa função possui 4 loops. Um desses loops percorre a tabela de codificação no início do arquivo compactado e o outro loop percorre todo o texto comprimido no arquivo compactado. Os outros dois loops estão aninhados, de forma que o loop externo percorre todo o texto comprimido do arquivo compactado e o loop interno percorre toda a tabela de codificação do arquivo compactado. A tabela de codificação pode ser do tamanho do próprio texto quando todas as palavras são diferentes ou pode ser bem menor do que o tamanho do texto quando há muitas palavras repetidas. Devido ao pior caso em que todas as palavras são diferentes, a complexidade de tempo da função é O(n^2).

Descompactar – Complexidade de Espaço: Essa função cria um vetor da estrutura Dupla para armazenar a tabela de codificação, cria duas strings adicionais que vão armazenar o texto comprimido do arquivo compactado e cria uma estrutura bitmap auxiliar. Dessa forma, a complexidade de espaço da função é O(n).

4. Análise de Robustez

Os erros tratados nesse projeto foram os erros de comando no terminal ao executar o programa e erro de abertura de arquivo. Se o comando digitado pelo usuário no terminal possuir mais que 4 argumentos ou menos que 4 argumentos, o programa avisa o usuário e é encerrado. Se o comando digitado pelo usuário no terminal possuir uma flag diferente de ‘-c’ ou ‘-d’, o programa avisa o usuário e é encerrado. Se o arquivo especificado para leitura não existe, o programa avisa o usuário e é encerrado.

5. Análise Experimental

Iremos analisar o tempo gasto pelas funções Arvore_Huffman, Compactar e Descompactar. A função Codifica_Bits não será analisada, pois o tamanho da tabela de frequência que é passada por parâmetro pra essa função é limitado no nosso código e, devido a esse condicionamento, todos os testes de tempo utilizados nessa função resultam em um tempo gasto igual a 0 ms, de maneira que não é possível realmente avaliar experimentalmente. As outras 3 funções que vamos analisar podem ser avaliadas experimentalmente a partir do número de caracteres e, para isso, eu realizei 10 testes para cada função de forma a variar uniformemente o número de caracteres de 3155000 até 31550000. O resultado desses experimentos serão mostrados através de gráficos que serão mostrados em seguida. A partir da análise dos gráficos é possível perceber que a função Arvore_Huffman gasta bem menos tempo do que as outras funções mesmo contendo a mesma complexidade de tempo da função Compactar, já que a função Compactar chama a função Arvore_Huffman dentro dela. No entanto, mesmo gastando um tempo maior, a inclinação nos gráficos dessas duas funções possuem uma inclinação bem parecida e com aparência linear, de forma que podemos comprovar a análise de complexidade O(n) atribuída às duas funções no tópico anterior. O gráfico da função Descompactar é um pouco diferente do esperado, mas há uma explicação para isso. A análise de complexidade da função Descompactar no tópico anterior diz questão ao pior caso possível em que todas as palavras são diferentes no texto que estamos tentando descompactar. Porém, nos testes foram usados textos compactados com muitas palavras repetidas, de forma que a complexidade dos testes foi bem menor do que a esperada no pior caso. A partir da

análise do gráfico de testes dessa função Descompactar, a função parece apresentar um crescimento linear.

Grafico da função Arvore_Huffman:

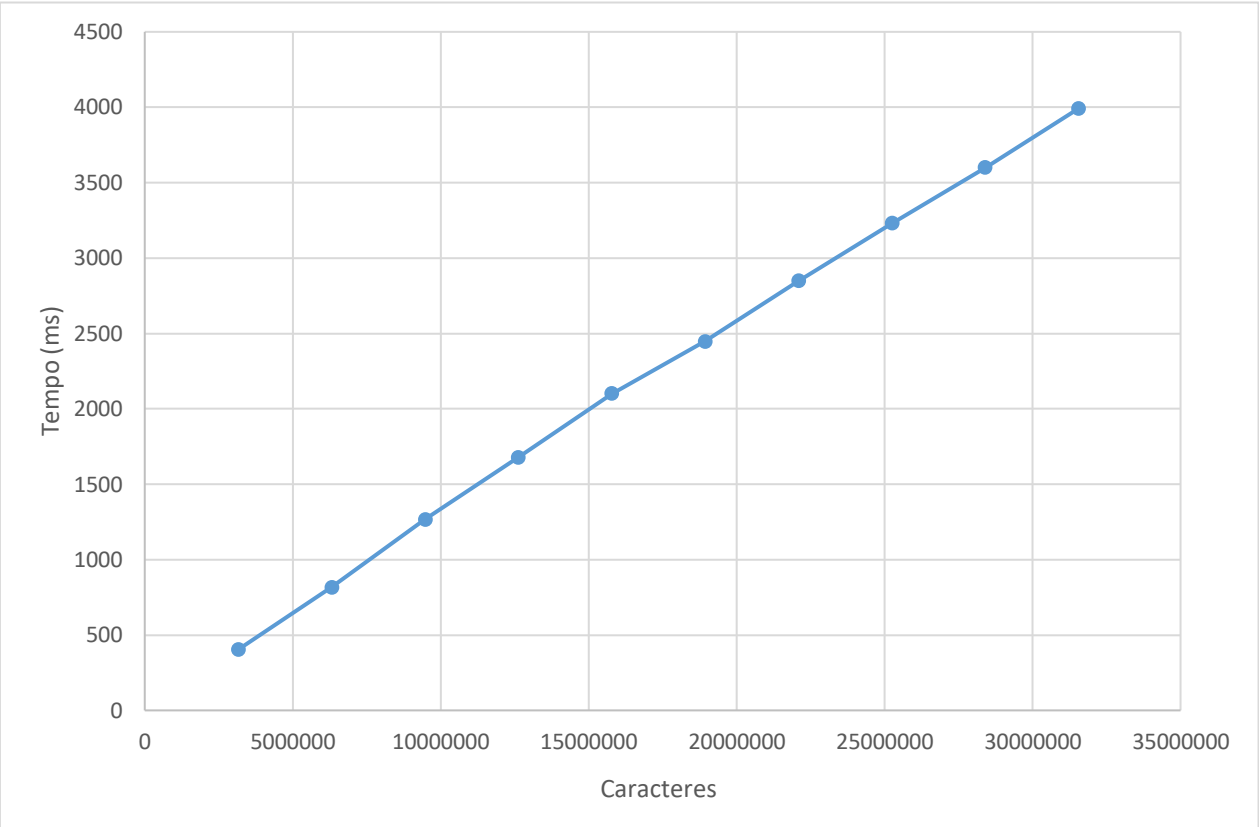


Gráfico da função Compactar:

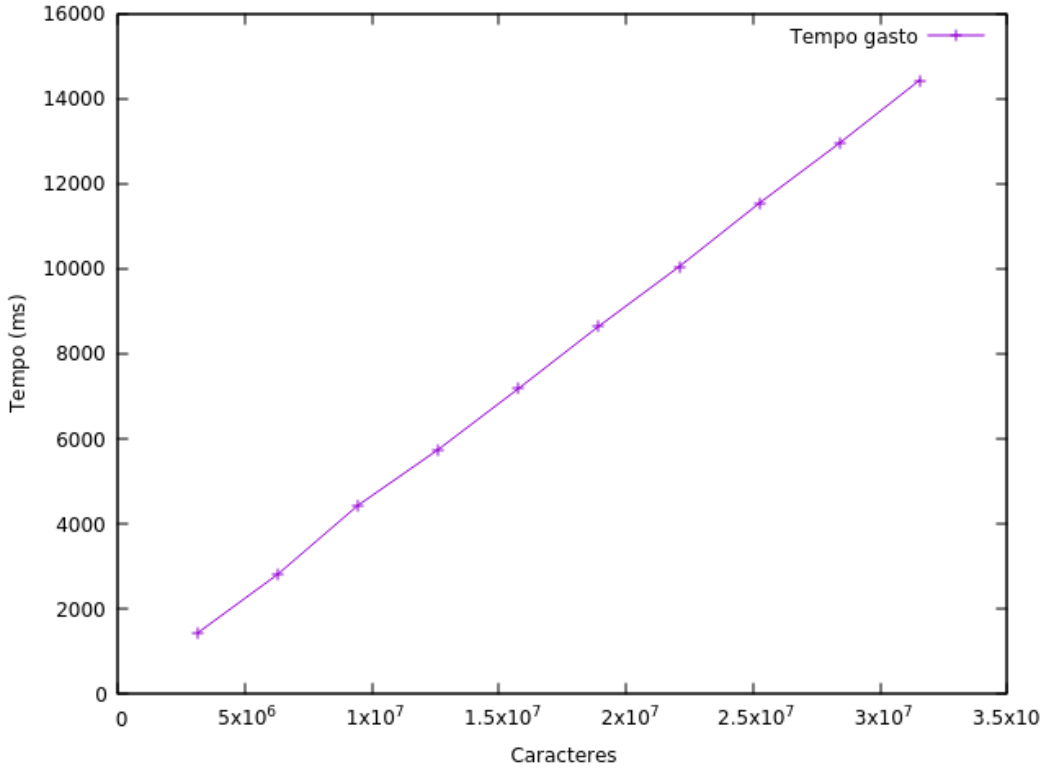
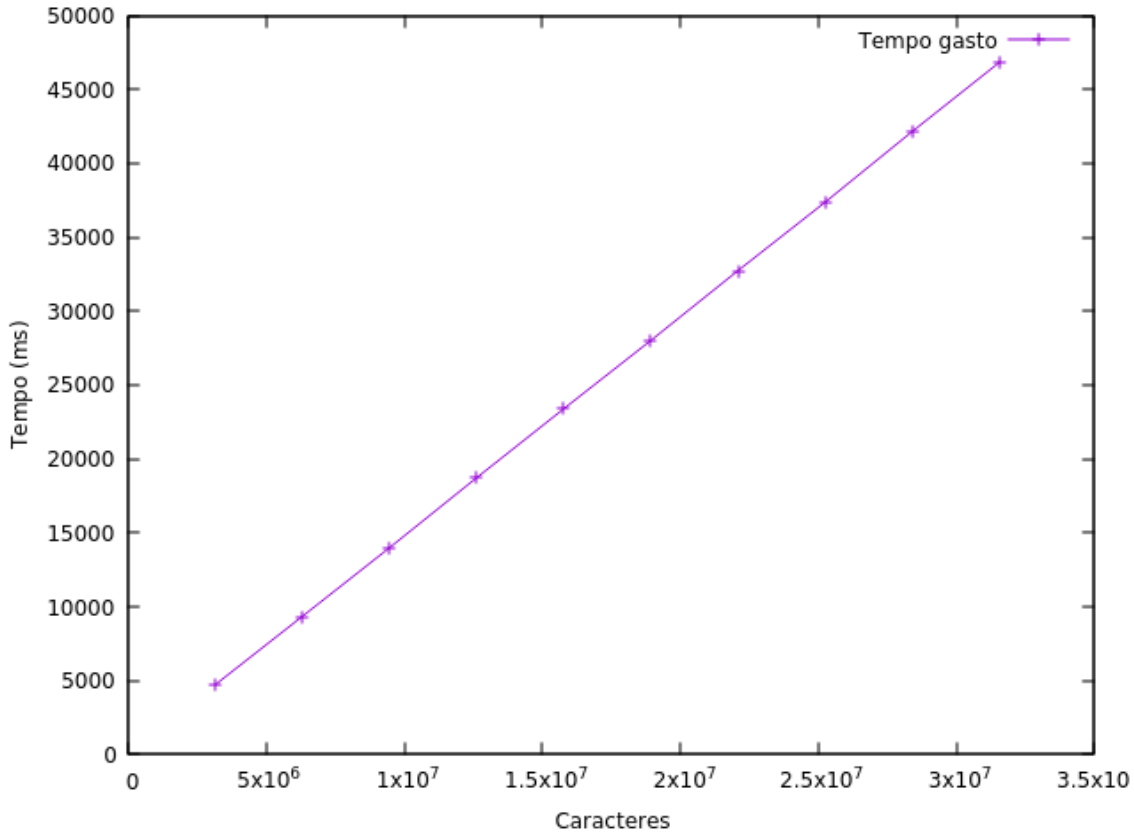


Gráfico da função Descompactar:



Conclusão:

Nesse trabalho foi realizado um código na linguagem C++ que compacta e descompacta arquivos através do algoritmo de Huffman. Esse programa foi realizado sem utilizar estruturas de dados provenientes de bibliotecas externas do C++, de forma que as estruturas No_Huffman, MeuVector e Dupla foram implementadas e possuem funções e características específicas para este projeto. Dessa maneira, o processo de formulação do código proporcionou um maior entendimento acerca dos tipos abstratos de dados e um maior entendimento sobre o algoritmo de compactação usado. Além disso, esse projeto demandou o uso de uma quantidade considerável de funções com um certo nível de dificuldade, o que exigiu o uso de ferramentas de depuração como o GDB e o valgrind. Outro componente exigido por esse trabalho foi a análise de complexidade, por meio de notações assintóticas, e a análise experimental, por meio de ferramentas que testam os desempenhos computacionais. A utilização de conceitos e ferramentas usados para compreender as estruturas de dados, as estratégias de depuração e as estratégias de análise experimental e de análise de complexidade foram ensinados nas aulas de Estrutura de Dados e, assim, esse trabalho prático foi essencial para fixar o conteúdo do curso e para aumentar a familiaridade com projetos de programação.

Bibliografia:

Slides virtuais da disciplina de Estrutura de Dados disponibilizados por Wagner Meira Jr.

Livro Thomas H. Cormen - Algoritmos: Teoria e Prática

https://cplusplus.com/reference/bitset/bitset/

https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman

Instruções para Compilação e Execução:

A compilação e execução é feita através do makefile.

O usuário deve deixar o arquivo txt que será compactado ou descompactado dentro da pasta TP que é a raiz do projeto. Ao inserir esse arquivo, ele deve digitar no terminal o comando ‘make run ARGS=”entrada.txt saída.txt –f’ ’, sendo que “entrada” do comando deve ser substituído pelo nome do arquivo de entrada e “saída” deve ser substituído pelo nome do arquivo que você quer guardar a compactação ou descompactação do arquivo de entrada, de forma que esse arquivo de saída pode existir ou não. A flag ‘-f’ também deve ser substituída por ‘-c’ se o usuário desejar compactar o arquivo de entrada ou por ‘-d’ se o usuário desejar descompactar o arquivo de entrada. Os arquivos podem ou não possuir a terminação ‘txt’.

Se o usuário desejar executar o código manualmente pelo terminal, após utilizar o comando make, ele deve digitar “./bin/tp entrada.txt saída.txt -c” e novamente substituir os nomes dos arquivos e da flag.