UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

TRABALHO PRÁTICO 3
COMPACTAÇÃO DE ARQUIVOS TEXTO

ALUNO: FELIPPE VELOSO MARINHO
MATRÍCULA: 2021072260
DISCIPLINA: ESTRUTURA DE DADOS
PROFESSOR: MARCIO COSTA SANTOS

1 - Introdução

O objetivo deste trabalho é explorar algoritmos de compactação e aplicar os conhecimentos adquiridos para desenvolver algoritmos eficientes que reduzam o tamanho de arquivos de texto sem comprometer a informação contida neles. Para alcançar esse objetivo, foi desenvolvido um sistema baseado no algoritmo de Huffman, utilizando as linguagens C/C++, e incorporando os conceitos estudados na disciplina de Estruturas de Dados.

O sistema implementado recebe, por linha de comando, dois nomes de arquivos e uma flag indicando a operação a ser realizada: -c (compactar) ou -d (descompactar). No caso da compactação, o primeiro arquivo contém o texto a ser compactado no formato UTF-8, enquanto o segundo arquivo está vazio. Já no caso da descompactação, o primeiro arquivo contém o texto previamente compactado pelo sistema, e o segundo arquivo conterá a descompactação do primeiro arquivo.

2 - Método

2.1 - Configurações da Máquina

- Windows 10 Pro x64
- WSL 2 com Ubuntu 20.04.06 LTS
- Processador: AMD Ryzen 5 5500U 2100 Mhz
- RAM: 8,00 GB (utilizável: 7,80 GB)
- Compilador: MinGW GCC Build 2 - 9.2.0
- Linguagem: C++

2.2 - Estruturas de Dados

A estrutura de dados que é necessária para utilização do algoritmo de huffman é a árvore usada na compactação dos dados onde é composta por Node(nós), que possuem um caractere, uma frequência, ponteiros para os filhos esquerdo e direito, e um ponteiro para o próximo nó na lista.

O construtor da classe inicializa a raiz da árvore como nulo. O destrutor verifica se a raiz não é nula e, caso não seja, libera a memória ocupada pela raiz e todos os seus nós filhos.

A função **createTree** é responsável por criar a árvore a partir de uma lista de nós. Ela recebe a lista como parâmetro e, enquanto a lista tiver mais de um elemento, remove os dois primeiros nós, cria um novo nó que será pai desses dois nós removidos, atualiza seus atributos e insere o novo nó de volta na lista. O processo continua até restar apenas um nó na lista, que será a raiz da árvore.

A função **printTree** somente é utilizada para imprimir a estrutura da árvore e a função **treeHeight** retorna a altura da árvore.

Outra estrutura utilizada de maneira a facilitar a manipulação e armazenar os nós da árvore de Huffman foi uma lista encadeada. A lista conta com o construtor da classe inicializa o início da lista como nulo e define o tamanho inicial como zero, o destrutor que percorre a lista e libera a memória ocupada por cada nó, a função **getSize** que retorna o tamanho da lista a função **getInicio** que retorna o ponteiro para o primeiro nó da lista, a função **insertNode** que é utilizada para inserir um novo nó na lista, a função **fillList** é responsável por preencher a lista encadeada com os nós da árvore de Huffman. Ela recebe uma tabela de frequências e a lista como parâmetros. Para cada posição da tabela em que a frequência é maior que zero, um novo nó é criado e inserido na lista utilizando a função **insertNode**.

Também há a função **removeInBegin** remove o primeiro nó da lista e o retorna. Caso a lista esteja vazia, é lançada uma exceção, a função **printList** é utilizada para imprimir os elementos da lista encadeada. Ela percorre a lista e imprime o caractere e a frequência de cada nó.

2.3 - Classes e suas Funções

Para a implementação do programa, além das classes provindas das estruturas de dados foram utilizadas as classes, Compress, Decompress, Dictionary, FrequencyTable, Tmp, Exceptions e a super classe onde são chamadas todas as funções do sistema, Program.

As classes Dictionary e FrequencyTable são modularizações de partes do algoritmo de huffman onde irei citar suas principais funções. Na primeira temos as funções **allocDictionary** que é responsável por alocar a memória para o dicionário que é implementado como uma matriz de caracteres. A função **createDictionary** é utilizada para criar o dicionário a partir da árvore de Huffman. Em FrequencyTable temos os construtores e destrutores, a função para setar a tabela de frequência com o valor de passado no parâmetro, outra para setar a tabela de frequência com o número 0 e a função pra preencher a tabela de frequência, todas sendo funções onde é percorrida toda tabela e feita a inserção. As funções citadas são, respectivamente, **setTable**, **setTableWithZero** e **fillTable**.

As classes de Exceptions e Tmp são classes secundárias, sendo uma onde estão armazenadas todas as exceções personalizadas e a outra onde estão as funções relacionadas ao retorno dos tempos de usuário e de sistema (**diffUserTime** e **diffSystemTime**). Na classe Tmp são utilizadas as bibliotecas <sys/resource.h> e <threads.h> e são passados como parâmetros o tempo de início e o tempo final de execução das funções.

As classes Compress e Decompress são responsáveis pelas principais funções do sistema onde em ambas temos os construtores e destrutores vazios e na primeira temos **compress** e **compact**, onde em **compress** é feita a compressão de uma string passada pelo parâmetro junto ao dicionário que é utilizado para substituir cada caractere da string pelo seu respectivo código de compressão ao final é retornado a string codificada. A função

compact compactamos o arquivo de texto e escrevemos em um arquivo binário. Para isso é percorrido a string codificada bit a bit. Para cada bit, a função atualiza o byte em construção (byte) utilizando uma máscara bitwise e um deslocamento (shift) para a esquerda. Quando o byte está cheio, ou seja, quando os 8 bits foram preenchidos, a função escreve o byte no arquivo binário. Ao final, a função verifica se o último byte não está completo e escreve no arquivo, se necessário. Por fim, o arquivo é fechado.

Em Decompress, suas principais funções são **decompress**, onde é recebido uma sequência de caracteres comprimidos e a raiz da árvore de huffman, percorre a sequência de caracteres comprimidos e se sor 0 move pro nó da esquerda, se for 1 pro nó da direita, se não tiver filhos, adiciona o caractere ao nó e redefine a raiz. No final é retornada a string descomprimida, já a função **descompactar** recebe o arquivo compactado, percorre o arquivo byte a byte e para cada byte, converte-o em uma sequência de 8 bits representada por '0's e '1's. No fim é retornada a sequência de caracteres descompactados.

Para finalizar, para a classe Program acho necessário para compreensão de todo o sistema, comentários sobre todas as funções.

sizeFile: Calcula quantos caracteres tem o arquivo, percorrendo o caminho passado no parâmetro, usada para alocação de memória necessária.

readFile: Função utilizada para leitura de arquivos e no retorno do texto em array de char.

processArguments: A função é utilizada para processar os argumentos que vem da main, argc e argv.

sizeFileBytes: Função para calcular quantos bytes tem o arquivo. É utilizada a biblioteca ifstream para isso e foi feita para retornar a taxa e o fator de compressão ao final da compressão do arquivo.

salvarArquivoDescompactado: Função que salva o texto descompactado em um arquivo .txt.

salvarTabelaFrequencia: Nessa função é feita um truque para que a descompressão seja possível após a compressão. É salvo a tabela de frequência logo após a compressão na pasta tmp como um arquivo binário,para assim ser utilizada na descompressão do arquivo binário posteriormente.

carregarTabelaFrequencia: Função auxiliar para carregar a tabela de frequência de um arquivo binário.

printMenu: Nessa função é feita todas as chamadas para que o sistema rode. É processado os argumentos e a partir disso se for compressão ele cria a tabela de frequência, salva a tabela em um arquivo auxiliar para ser utilizado posteriormente na descompressão, cria a lista de nós para a criação da árvore de huffman. Assim é criado o dicionário, alocando a memória necessária e com isso podemos codificar o texto em outro array de caracteres de “0s” e “1s”. Depois compactamos passando o texto codificado e salvamos em um arquivo binário. Por fim, imprimimos o fator e a taxa de compressão e os tempos de usuário e de sistema na execução das funções.

No caso de passar a flag para descompactar, descompactamos o texto em um array de char utilizando a função descompactar que retorna os bytes agora em caracteres de 0 e 1. Depois lemos a tabela que foi salva em um arquivo temporário durante a compactação e repetimos o processo de criar uma lista de nós, a árvores de huffman, o dicionário e por fim realizamos a descodificação do texto e salvamos em um arquivo.

3 - Análise de Complexidade

Para meio de economia das páginas serão documentadas as ordens de complexidade de tempo e espaço somente das principais funções do sistema.

Nas funções de FrequencyTable, temos setTable e setTableWithZero com loops com números fixos de iterações (MAX_SIZE 256), sendo assim o tempo de execução é constante O(1). A complexidade de espaço de ambas também é constante visto que não há alocação de memória.
Na função fillTable temos a ordem de complexidade de tempo sendo O(N), visto que a função percorre o texto até encontrar o caractere nulo (‘\0’). A complexidade de espaço é O(1) já que também não há alocação de memória.

Em Dictionary a complexidade de tempo é O(MAX_SIZE * columns), já que a função possui um loop com MAX_SIZE de iterações e dentro dele é alocado espaços para cada coluna. A complexidade de espaço é a mesma de tempo, já que é alocado um array de ponteiros de tamanho MAX_SIZE e o espaço para cada coluna em cada posição.

Na classe Decompress temos a função decompress com complexidade de tempo sendo O(N), que possui um loop que percorre o texto até encontrar o caractere nulo. A complexidade de espaço é a mesma, sendo O(N) pois a função aloca espaço para a string descomprimida que tem tamanho igual ao texto de entrada (N). A outra função relevante é a descompactar, que possui a ordem de complexidade de tempo O(M), pois possui um loop que percorre o arquivo binário até o final. E a complexidade de espaço igual, sendo alocado espaço suficiente para o buffer de saída que tem tamanho igual ao tamanho do arquivo (M).

Por fim a classe Compress possui duas funções relevantes para análise, compress e compact. Na primeira citada, a ordem de complexidade de tempo e de espaço são O(N) por possuir dois loops aninhados que percorrem até o (‘\0’) e o segundo percorre cada caractere da string na tabela de dicionário, e por alocar espaço para a string comprimida que tem tamanho dependente do tamanho da string de entrada (N). Por fim a função compact possui complexidade de tempo O(N), possui um loop que percorre a string de entrada até o caractere nulo (‘\0’). A ordem de complexidade de espaço é O(1) pois a função não chega a alocar nenhum espaço.

4 - Estratégias de Robustez

De modo a garantir a robustez e a corretude do programa após a entrada do usuário passada pelo argumento, foi empregado o uso de “exceptions”, semelhante aos “msgasserts” visto em C.

Foi criada uma classe com diversas exceções personalizadas onde são disparadas caso ocorra uma exceção no código.

Algumas das exceções colocadas incluem:
As exceções de arquivo não encontrado, caso os ponteiros passados como parâmetros sejam apontados para o nulo;
Exceção de erro na escrita ou leitura do arquivo;
Exceção de erro na compressão de um arquivo;
Exceção de erro na descompressão de um arquivo;
Erro na alocação de memória, usada principalmente em Dictionary;
Exceção de lista vazia;

Um possível ponto fraco analisado é que o sistema só roda com toda a eficiência em linux, isso foi observado a partir de testes porém para o intuito do TP, que será corrigido em máquinas Linux, o código funciona perfeitamente como o planejado. Além disso, o código foi configurado para o português, portanto ele pode até funcionar na compressão e descompressão de outras línguas do latim, porém é aconselhado o uso somente para textos em português brasileiro.

Por último, é necessário a compactação antes da descompactação de um arquivo. Porque somente durante a compactação é salvo a tabela de frequência em um arquivo temporário localizado na pasta tmp, que será utilizado na descompactação posteriormente.

5 - Análise Experimental

A análise de desempenho computacional foi realizada usando a ferramenta gprof, com o objetivo de identificar os métodos que mais impactaram o desempenho do programa, bem como observar como o tamanho e o tipo da expressão afetam o tempo de execução por meio do número de chamadas.

No exemplo abaixo, um arquivo de 4.055.830 bytes foi utilizado como texto para compactação. Ao analisar as funções com a ajuda da função memlog, é possível observar que o número de chamadas da função escreveMemlog é exatamente igual à quantidade de bytes do arquivo. Também é evidente que as principais funções usadas na codificação e compressão de arquivos são responsáveis pela maior parte do tempo de execução do usuário.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
47.04	0.47	0.47	1	470.38	470.38	Compress::compress(char**, unsigned char*)
26.02	0.73	0.26	1	260.21	260.21	Compress::compact(unsigned char*, char*)
16.01	0.89	0.16	4055830	0.00	0.00	escreveMemLog(long, long, int)
7.01	0.96	0.07	1	70.06	230.19	FrequencyTable::fillTable(unsigned char*)
4.00	1.00	0.04	1	40.03	40.03	Program::readFile(char*)
0.00	1.00	0.00	204	0.00	0.00	List::removeInBegin(List*)
0.00	1.00	0.00	104	0.00	0.00	leMemLog(long, long, int)
0.00	1.00	0.00	103	0.00	0.00	List::insertNode(List*, node*)
0.00	1.00	0.00	103	0.00	0.00	List::getSize()
0.00	1.00	0.00	2	0.00	0.00	Decompress::decompress(unsigned char*, node*)
0.00	1.00	0.00	2	0.00	0.00	Dictionary::~Dictionary()
0.00	1.00	0.00	2	0.00	0.00	List::~List()
0.00	1.00	0.00	2	0.00	0.00	Tree::~Tree()
0.00	1.00	0.00	2	0.00	0.00	Program::sizeFileBytes(char*)
0.00	1.00	0.00	2	0.00	0.00	Compress::stringSize(char**, unsigned char*)
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN10DecompressC2Ev
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN10DictionaryC2Ev
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN14FrequencyTableC2Ev
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN3Tmp12diffUserTimeEP6rusageS1_
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN4ListC2Ev
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN4TreeC2Ev
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN7Program8sizeFileEPc
0.00	1.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN8CompressC2Ev

Imagem 1 - Análise do gprof ao executar a compilação de um arquivo

Em relação a memória, utilizando o valgrind é possível observar alguns erros quanto ao uso de memória. Na função salvarTabelaFrequencia(), a inicialização do array da tabela de frequência é definida com um valor fixo de MAX_TAM 256, o que acaba gerando um erro de Syscall param write(buf) points to uninitialised byte(s). Nas funções de createTree e fillList também são gerados alguns erros de valores não devidamente inicializados, porém isso acontece de maneira semelhante ao primeiro caso citado, onde por definir o MAX_SIZE no loop, além disso por ser necessário que os valores do retorno dessas funções se mantenham alocados é gerado o erro. Na função compress é necessário que se mantenha alocado o array de char com o texto codificado para ser usado posteriormente nas operações bit a bit para compressão.

```
==1609==
==1609== HEAP SUMMARY:
==1609==     in use at exit: 18,664,327 bytes in 205 blocks
==1609==   total heap usage: 524 allocs, 320 frees, 22,879,967 bytes allocated
==1609==
==1609== 6,528 (32 direct, 6,496 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 4
==1609==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1609==    by 0x10DAA6: List::fillList(unsigned int*, List*) (list.cpp:100)
==1609==    by 0x10BF4A: Program::printMenu(int, char**) (program.cpp:285)
==1609==    by 0x10AB22: main (main.cpp:20)
==1609==
==1609== 18,657,799 bytes in 1 blocks are possibly lost in loss record 4 of 4
==1609==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1609==    by 0x10CF47: Compress::compress(char**, unsigned char*) (compress.cpp:61)
==1609==    by 0x10BFBA: Program::printMenu(int, char**) (program.cpp:299)
==1609==    by 0x10AB22: main (main.cpp:20)
==1609==
==1609== LEAK SUMMARY:
==1609==    definitely lost: 32 bytes in 1 blocks
==1609==    indirectly lost: 6,496 bytes in 203 blocks
==1609==      possibly lost: 18,657,799 bytes in 1 blocks
==1609==    still reachable: 0 bytes in 0 blocks
==1609==         suppressed: 0 bytes in 0 blocks
==1609==
==1609== Use --track-origins=yes to see where uninitialised values come from
==1609== For lists of detected and suppressed errors, rerun with: -s
```

Imagem 2 - Análise do valgrind: HEAP SUMMARY e LEAK SUMMARY

Foi feita a tentativa da utilização da biblioteca Analisamem para a observação da localidade de referência de modo que foi definido somente uma fase. Dessa forma o gráfico de acesso gerado possui alguns saltos que provavelmente são provenientes da utilização de matrizes para o armazenamento dos dicionários, visto que são acessadas linha a linha.

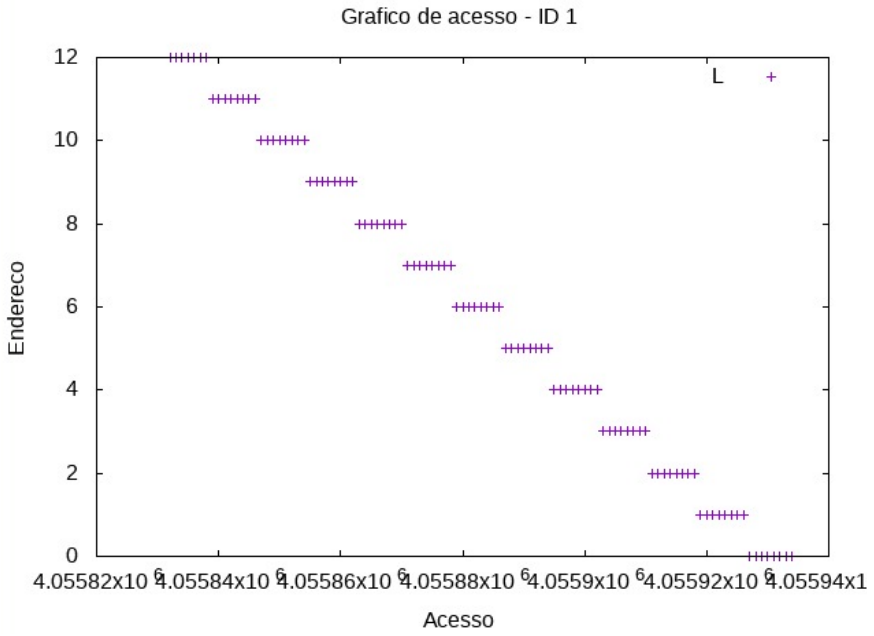

Imagem 3 - Análise do Analisamem: Gráfico de acessos

A imagem 4 mostra que a distância de pilhas nas funções de lista tiveram distâncias constantes, porém a distância de pilhas mostra que o programa está acessando locais de memória que não estão próximos uns dos outros em termos de localização física. Isso provavelmente está resultando em um impacto negativo no desempenho do programa devido à natureza da hierarquia de memória do sistema. Isso pode levar a um maior número de falhas de cache, onde a CPU precisa buscar os dados na memória principal, o que é um processo mais lento. Como resultado, a execução do programa pode ficar mais lenta devido

a essas falhas de cache e ao tempo adicional necessário para buscar os dados em uma distância maior na memória.

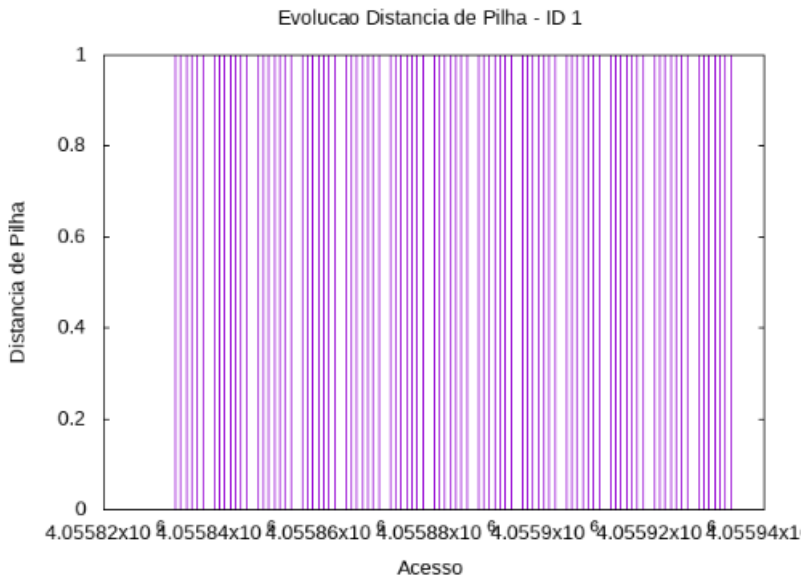

Imagem 4 - Análise Analisamem: Evolução da distância de Pilha

6 - Conclusões

O Trabalho Prático 3 abordou uma grande parte dos conhecimentos que estudamos ao longo do curso. Foi necessário aprimorar especialmente nossa compreensão da estrutura de árvore para implementar o algoritmo de Huffman. Além disso, a temática do trabalho foi muito interessante e apresentou um nível de dificuldade adequado, mesmo considerando o final do período letivo. A documentação, mais uma vez, desempenhou um papel fundamental na consolidação dos conhecimentos sobre análise de complexidade e desempenho computacional.

Em resumo, o trabalho mais uma vez foi altamente benéfico para a aplicação prática dos conhecimentos adquiridos na disciplina.

7 - Bibliografia

Algoritmo de Huffman em C **-** Canal Programe seu futuro**.** Utilizado para compreensão da teoria por trás do código e na implementação.

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8 - Instruções para compilação e execução

Para a compilação do arquivo é utilizado o comando make, que irá compilar todos os arquivos do programa e gerar os arquivos fontes e objetos na estrutura de pasta pedida no enunciado do Trabalho Prático. O comando make clean faz a limpeza dos arquivos objetos e do executável.

Para a execução é necessário compilar o programa usando o comando make onde serão compilados os arquivos e gerado o executável em ./bin/main.

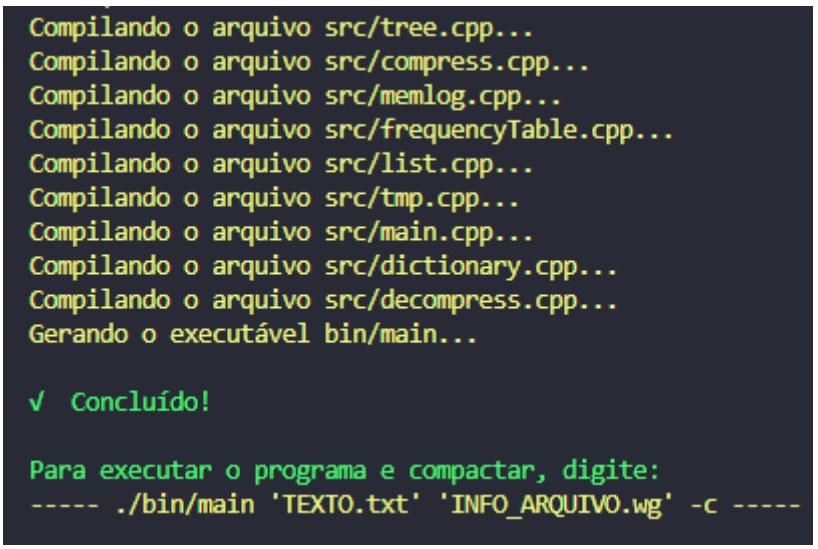

Imagem 5 - Instruções do terminal para compactação

Logo após, como indicado no próprio terminal, é necessário compactar o arquivo passando como argumentos após o comando ./bin/main, o caminho do texto que será usado para a compressão, o arquivo para salvar as informações da compactação, todos em .txt, e por fim a tag '-c-' para a compactação ser realizada.

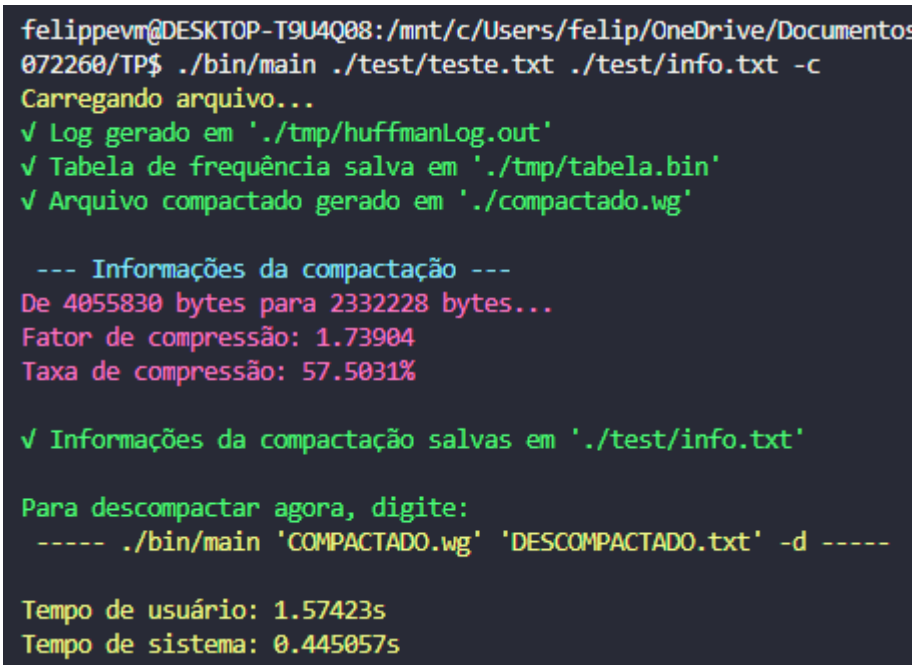

Imagem 6 - Instruções do terminal para descompactação

Após a compactação do arquivo ser feita com sucesso, as instruções para descompactação serão informadas no terminal novamente acima dos tempos de usuário e de sistema. Para a descompactação, após o './bin/main' é necessário passar como argumento o arquivo anteriormente compactado, em geral a extensão será .wg, o arquivo onde ficará armazenado o texto descompactado e a flag '-d' para a descompactação.

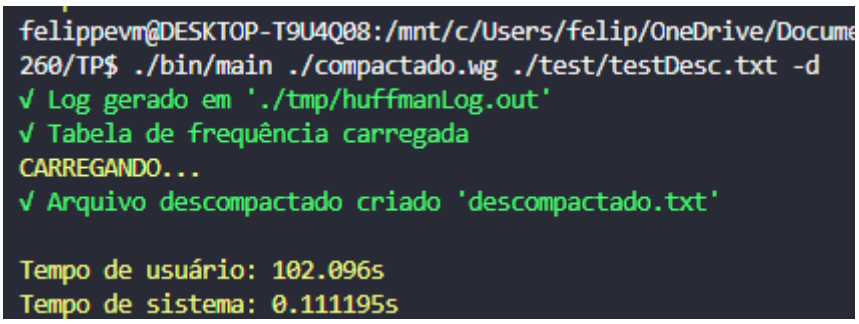

Imagem 7 - Informações do terminal após o término da descompactação