

Trabalho Prático 3

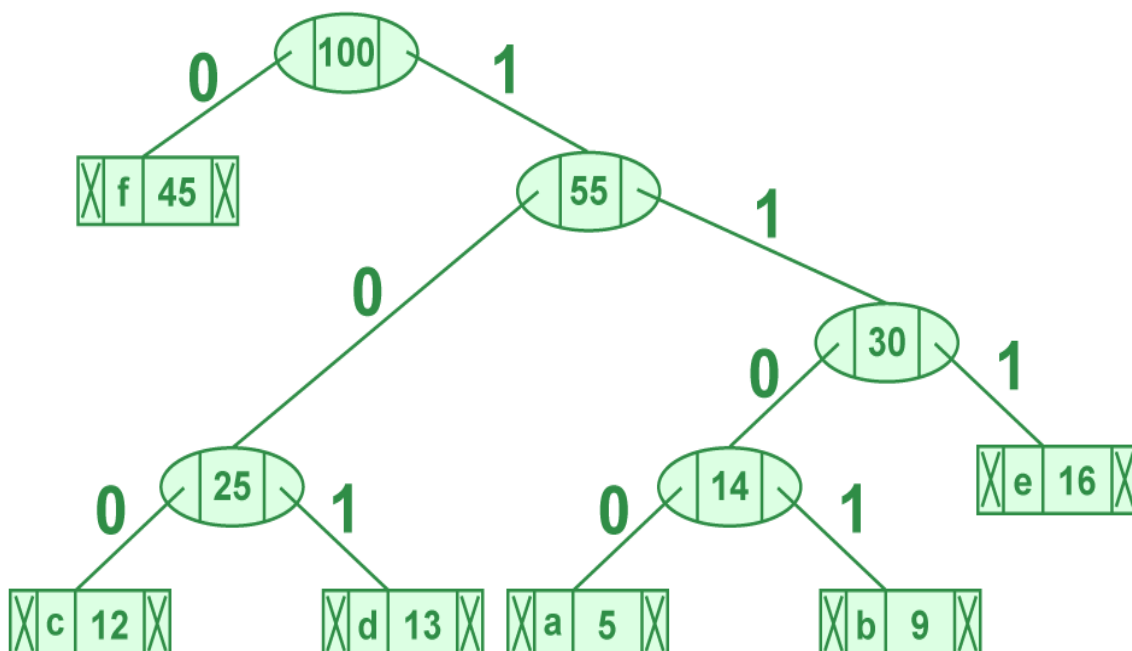
João Marcos Tomáz Silva Campos - 2022043728 Universidade Federal de Minas Gerais (UFMG) Belo Horizonte - MG - Brasil

joaomarcostsc2003@ufmg.br

1 Introdução

O problema a ser tratado é referente a implementação do algoritmo de Huffman. A ideia consiste em compactar arquivos de forma eficiente utilizando os métodos, demonstrando um pouco sobre formas eficientes de tratar armazenamento de dados. É importante deixar claro que para este trabalho foi optado por fazer a compressão e descompressão dos arquivos observando como parâmetro o número de caracteres.

A codificação de Huffman é um método de compressão que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser comprimido para determinar códigos de tamanho variável para cada símbolo. Foi desenvolvido em 1952 por David A. Huffman que era, na época, estudante de doutorado no MIT, e foi publicado no artigo "A Method for the Construction of Minimum-Redundancy Codes". Abaixo um exemplo de uma árvore de huffman.



Este documento se divide nos seguintes tópicos

2 Método: Descrição da implementação, detalhando as estruturas de dados, tipos abstratos de dados (ou classes) e funções (ou métodos) implementados.

3 Análise de Complexidade: Contém a análise da complexidade de tempo e espaço dos procedimentos implementados, formalizada pela notação assintótica.

4 Estratégias de Robustez: Contém a descrição, justificativa e implementação dos mecanismos de programação defensiva e tolerância a falhas implementados.

5 Análise Experimental: Apresenta os experimentos realizados em termos de desempenho computacional, assim como as análises dos resultados.

6 Conclusões

2 Método

2.1 Configurações dos ambientes utilizados

- Sistema operacional: macOS Ventura Versão 13.3.1
- O programa foi desenvolvido em C++
- Compilador g++ provido do Xcode
- Processador: Apple M2
- Memória: 8 GB RAM

2.2 Classes e estruturas

Para o desenvolvimento do trabalho foi utilizado uma única classe que possui as funções de implementação do algoritmo, chamada "Huffman" e foi utilizado uma estrutura "Nó" para que a estrutura da árvore binária fosse implementada. Também existem estruturas criadas para a implementação de uma programação defensiva para o código, contendo os erros possíveis.

2.3 Estrutura de dados

Neste trabalho utilizou-se a estrutura de dados árvore binária de Huffman. Uma árvore binária completa, chamada de árvore de Huffman é construída recursivamente a partir da junção dos dois símbolos de menor probabilidade, que são então somados em símbolos auxiliares e estes símbolos auxiliares recolocados no conjunto de símbolos. Também foi

utilizado para a implementação do código a estrutura Heap, para que fosse garantido a integridade da árvore de huffman.

2.4 Funções

void Huffman::findFrequency: Monta a tabela de frequências de caracteres.

void Huffman::trade: Troca dois nós de lugar.

void Huffman::heapify: Função que refaz a heap e garante sua integridade.

void Huffman::build_heap: Função que monta a heap.

Node* Huffman::get_min: Função que retorna o menor nó.

void Huffman::insert: Função para inserir um nó na árvore.

Node* Huffman::huffmanTree: Função que constrói a árvore de huffman.

void Huffman::build_table: Constrói o dicionário de frequências e caracteres representantes.

void Huffman::write_binary: Escreve em binário em um arquivo.

string Huffman::read_binary: Ler um arquivo binário.

void Huffman::compress: Função responsável pela compressão do arquivo.

void Huffman::decompress: Função responsável pela descompressão do arquivo.

2.5 Funcionamento

Para o funcionamento do código, é passado o nome dos arquivos a serem comprimidos ou descomprimidos, tendo o modo desejado especificado. Tendo esses parâmetros, chama-se a função responsável para aplicar o método escolhido. Para que a descompressão seja feita, é necessário o dicionário, portanto, ao comprimir, é gerado um arquivo chamado "table.txt" que auxiliará na descompressão. Caso seja passado um arquivo binário que tenha sido comprimido por outro código, não será possível fazer sua descompressão.

3 Análise de Complexidade

Para uma análise eficiente, devemos retornar em cada uma das funções e analisar suas complexidades separadamente. Em seguida, podemos somá-las e teremos a complexidade de tempo. Para a complexidade de espaço, basta analisar as alocações de memória feitas no código.

3.1 Complexidade de tempo

void Huffman::findFrequency: Monta a tabela de frequências de caracteres, para isso é necessário passar por todos os caracteres, assim, $O(n)$.

void Huffman::trade: Troca dois nós de lugar em tempo constante. $O(1)$.

void Huffman::heapify: Função que refaz a heap e garante sua integridade. $O(\log n)$.

void Huffman::build_heap: Função que monta a heap. $O(n \log n)$.

Node* Huffman::get_min: Função que retorna o menor nó. Por utilizar a função heapfy, ela se torna $O(\log n)$.

void Huffman::insert: Função para inserir um nó na árvore. $O(\log n)$

Node* Huffman::huffmanTree: Função que constrói a árvore de huffman. $O(n \log n)$.

void Huffman::build_table: Constrói o dicionário de frequências e caracteres representantes. $O(\log n)$

void Huffman::write_binary: Escreve em binário em um arquivo. $O(n)$.

string Huffman::read_binary: Ler um arquivo binário. $O(n)$.

void Huffman::compress: Função responsável pela compressão do arquivo. Utiliza várias das funções auxiliares, porém sua complexidade é determinada pela função dominante. Assim, $O(n \log n)$.

void Huffman::decompress: Função responsável pela descompressão do arquivo. Utiliza várias das funções auxiliares, porém sua complexidade é determinada pela função dominante. Assim, $O(n \log n)$.

Temos que a complexidade de tempo final do código será a soma das complexidades ou então tem como limite superior a maior complexidade das funções, portanto a complexidade do código final é de $O(n \log n)$.

3.2 Complexidade de espaço

O código utiliza de alocações de memória e manipulações de memória para que seja construído a árvore de Huffman. Fora isso, é utilizado apenas alocações estáticas. Com isso, por depender do número de nós a serem alocados, a complexidade de espaço do código é de $O(n)$.

4 Estratégias de robustez

O código utiliza como estratégia de robustez algumas práticas de programação defensiva que param a execução caso alguma das exceções sejam chamadas e avisa o erro para o usuário poder consertar. Essas exceções são:

file_not_open: Caso não seja encontrado o arquivo que o usuário passou para o programa, ele será encerrado e retornará que não foi encontrado o arquivo.

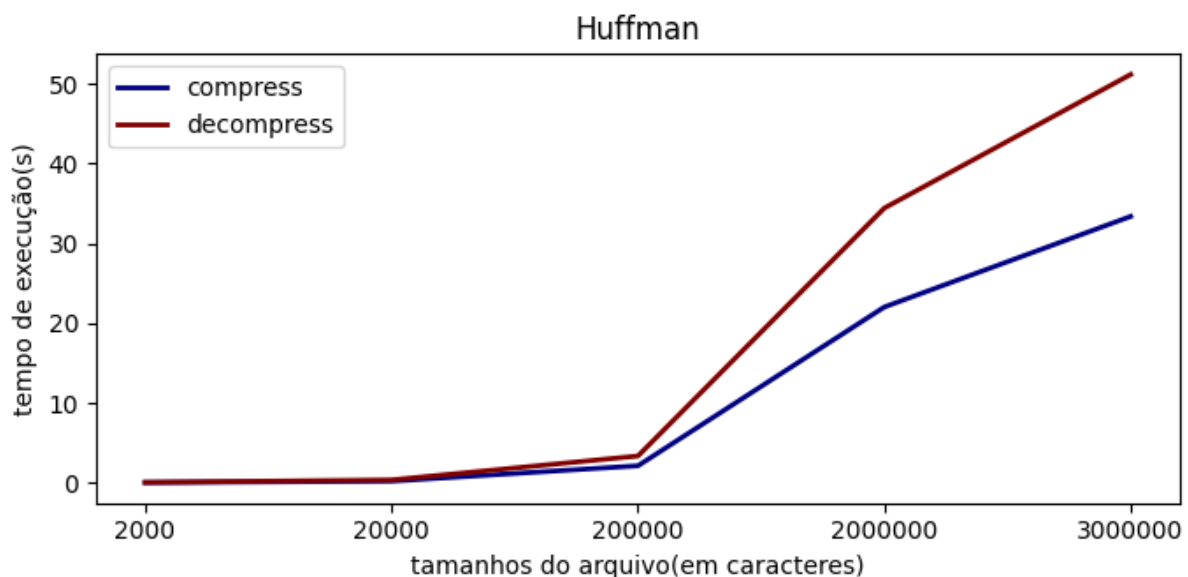
invalid_mode: Caso o usuário passe qualquer tipo de modo diferente de compressão e descompressão, o programa irá avisá-lo dos modos possíveis.

invalid_file: Caso seja passado condições de arquivos inválidos, o programa é encerrado. Um exemplo é arquivos com mesmo nome.

5 Análise Experimental

Para a análise experimental foram utilizadas algumas bibliotecas de c++ para que fosse possível analisar os tempos de execução e fazer as comparações. Também foram utilizadas ferramentas de análise de desempenho como o valgrind e a leitura do gmon.out para ter uma melhor leitura.

A análise foi feita para tamanhos consideráveis de entrada para analisar o desempenho de cada um dos algoritmos. Para essa análise foi feito um arquivo .csv e um código de plotagem em python para que fosse possível entender os desempenhos dos códigos. Os arquivos de teste foram gerados por um comando que gera arquivos aleatórios a partir de um intervalo de letras e números e de um número específico de caracteres.

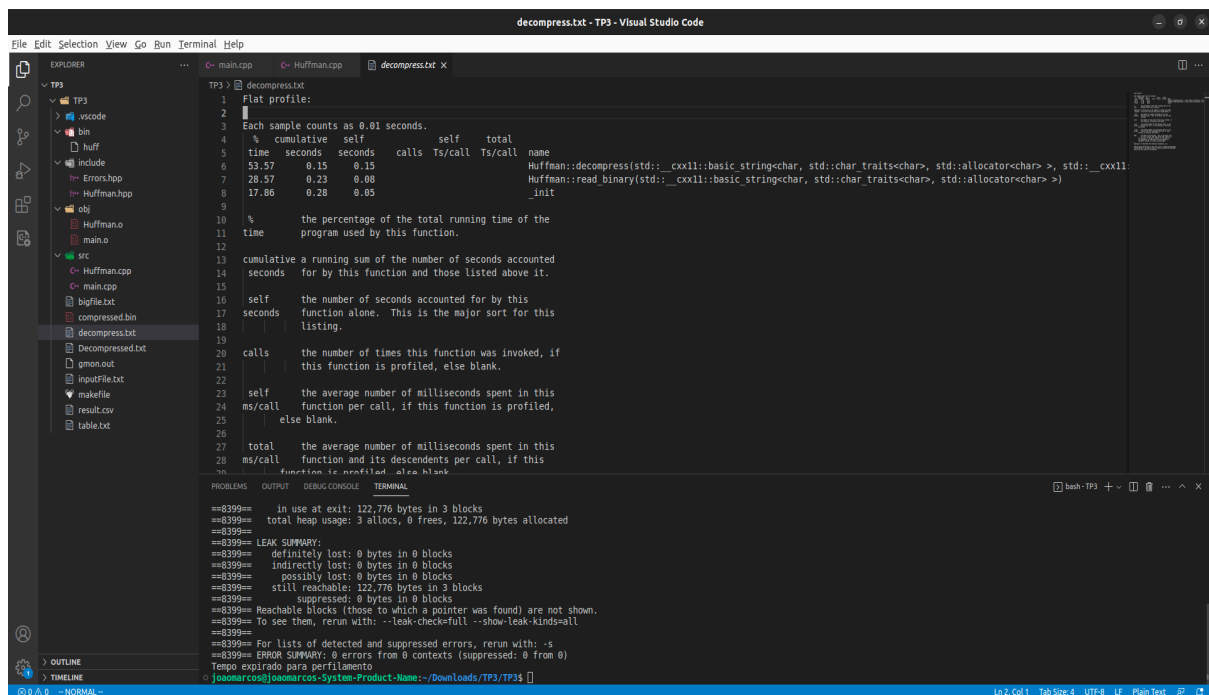


Como é possível observar pelo gráfico, o código possui um desempenho eficiente para compressão, tendo um tempo de execução quase linear para arquivos com um número

baixo de caracteres. Porém conforme aumenta-se o número de caracteres por arquivo, é necessário um tempo alto para que seja feito a compressão e um tempo maior ainda para que seja descomprimido.

A descompressão necessita de um tempo maior devido ao fato que necessita da leitura do arquivo do dicionário e também da leitura em binário e conversão a partir do caminhamento na árvore.

Além dessa análise de acordo com o tamanho dos arquivos, foram utilizadas ferramentas para observar o comportamento do código com relação a acessos de memória.



```
decompress.txt - TP3 - Visual Studio Code
File Edit Selection View Go Run Terminal Help

TP3
  bin
  huff
  include
  Errors.hpp
  Huffman.hpp
  obj
  Huffman.o
  main.o
  src
    Huffman.cpp
    main.cpp
    bigfile.txt
    compressed.bin
    decompress.txt
    Decompressed.txt
    gmon.out
    inputFile.txt
  makefile
  result.csv
  table.txt

TP3 > decompress.txt
1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4
5 % cumulative self total
6 time seconds calls Ts/call Ts/call name
7 53.57 0.15 0.15 Huffman::decompress(std::cxll::basic_string<char, std::char_traits<char>, std::allocator<char>>, std::cxll
8 17.86 0.28 0.05 Huffman::read_binary(std::cxll::basic_string<char, std::char_traits<char>, std::allocator<char>>)
9 _init
10
11 % the percentage of the total running time of the
12 program used by this function.
13
14 cumulative a running sum of the number of seconds accounted
15 seconds for by this function and those listed above it.
16
17 self the number of seconds accounted for by this
18 seconds function alone. This is the major sort for this
19 listing.
20
21 calls the number of times this function was invoked, if
22 this function is profiled, else blank.
23
24 self the average number of milliseconds spent in this
25 ms/call function per call, if this function is profiled,
26 else blank.
27
28 total the average number of milliseconds spent in this
29 ms/call function and its descendents per call, if this
30 function is profiled, else blank.
31
32 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
33
34 ==8399== in use at exit: 122,776 bytes in 3 blocks
35 ==8399== total heap usage: 3 allocs, 0 frees, 122,776 bytes allocated
36
37 ==8399== LEAK SUMMARY:
38 ==8399== definitely lost: 0 bytes in 0 blocks
39 ==8399== indirectly lost: 0 bytes in 0 blocks
40 ==8399== possibly lost: 0 bytes in 0 blocks
41 ==8399== still reachable: 122,776 bytes in 3 blocks
42 ==8399== suppressed: 0 bytes in 0 blocks
43 ==8399== Reachable blocks (those to which a pointer was found) are not shown.
44 ==8399== To see them, rerun with: --leak-check=full --show-leak-kinds=all
45
46 ==8399== For lists of detected and suppressed errors, rerun with: -s
47 ==8399== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
48
49 Tempo expirado para perfilamento
50
51 João Carlos de Jesus - System-Product-Name - /Downloads/TP3/TP3$
```

Como é possível observar acima, o código não possui defeitos quanto ao acesso de memória. Além disso, temos pela leitura do arquivo gmon.out, feita pelo gprof, que as funções rodam sem nenhum defeito. Um detalhe importante, para arquivos com números imensos de caracteres, maiores que os passados nessa documentação, o algoritmo não suporta e retorna "Morto", necessitando de melhorias. Uma forma possível, assim como visto em sala, fazer uma intercalação polifásica.

6 Conclusões

Após a conclusão do trabalho e dos estudos feitos para sua execução, foi possível perceber como o algoritmo de Huffman é uma boa opção para muitos casos em que seja necessário a compressão de arquivos. Ainda que antigo, é um algoritmo eficiente em certos casos na atualidade. Com a resolução desse trabalho foi possível colocar em prática muitos dos conceitos vistos em sala na parte final da matéria de Estrutura de Dados, além de contribuir muito com os conhecimentos acerca do tema específico.

Bibliografia

Cormen, Thomas . Algoritmos: Teoria e Prática, 3ª edição. São Paulo: GEN LTC, 2012.

Wikipédia. Wikipédia, a enciclopédia livre: Codificação de Huffman. Disponível em https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman. Acesso em: 01 de julho de 2023.

Instruções para compilação e execução

O código foi estruturado para que fique da maneira mais fácil possível para sua utilização para várias entradas de teste. Para isso, siga os passos abaixo.

1- Compile o programa pelo comando "make" no terminal;

2- Depois de compilado, deve-se executar pelo terminal da seguinte forma:

Caso queira comprimir um arquivo.

```
./bin/huff <nome_arquivo.txt> <nome_comprimido.bin> -c
```

Caso queira descomprimir um arquivo.

```
./bin/huff <nome_comprimido.bin> <nome_arquivo.txt> -d
```

O código possui em seu início um parse args para que seja feita a leitura dos comandos feitos pelo terminal e execute de acordo como passado pelo usuário.

Caso queira ser feita análises de desempenho, pode ser efetuado os seguintes comandos abaixo:

```
gprof ./bin/huff gmon.out > saida.txt
```

```
valgrind --leak-check=full ./bin/huff
```

Para limpar os arquivos executáveis gerados, pode-se utilizar o comando "make clean".