UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Compressão de arquivos de texto

Aluno: Lucas de Oliveira Araújo

Matrícula: 333

Belo Horizonte, MG
2023

Sumário

---

1 Introdução

No mundo contemporâneo, informação digital é uma das principais formas de criação, transmissão e uso de dados. Com a proliferação de dispositivos eletrônicos, redes de comunicação e a internet, a geração e o compartilhamento de dados digitais se tornaram ubíquos em praticamente todas as esferas da sociedade.

Por conseguinte, a compressão, isto é, a redução do tamanho dos dados originais, se torna um ponto-chave para permitir o armazenamento e troca dessas informações de forma eficiente. Nesse sentido, existem duas abordagens principais na compressão de dados: a compressão *lossless* (sem perdas) e a compressão *lossy* (com perdas). O método de compressão com perdas se baseia na ideia de remover do arquivo original os dados que são menos perceptíveis para o usuário, como certas frequências de som em arquivos de áudio, resultando em uma redução de tamanho. Por outro lado, a compressão sem perdas adota uma abordagem de reescrever os dados do arquivo de forma que seja possível reconstruí-lo com completa integridade, garantindo que nenhum dado seja perdido durante o processo.

Neste trabalho, apresentaremos um programa que utiliza o algoritmo de Huffman para compactar arquivos de texto, preservando a integridade dos dados e garantindo que nenhuma informação seja perdida [1].

2 Método

2.1 Tipos abstratos de dados

Dado que a natureza do problema é a manipulação de arquivos no disco, o primeiro TAD implementado foi o `Compress`. Essa classe comporta os métodos de compressão e descompressão dos arquivos, bem como métodos auxiliares para a escrita de dados no disco.

A classe `Parser` é a classe que auxilia os métodos da Compress. Ela é responsável por realizar as validações necessárias para compactar ou descompactar um determinado arquivo disponibilizado pelo usuário.

Em geral, todo o programa gira em torno da classe Compress, o que a torna o coração de todo o código.

2.2 Estruturas de dados

A ideia básica do algoritmo de Huffman é reescrever os dados de forma que eles ocupem menos espaço. Isso pode ser feito utilizando diferentes abordagens de compactação: ao nível de letra, ao nível de sílaba ou ao nível de palavra. Em todas essas abordagens, a ideia fundamental é reescrever os dados mais frequentes com a menor representação possível [2]. Nesse trabalho, o método escolhido foi a compactação ao nível de letra.

Diante disso, o código de Huffman para a abordagem escolhida consiste em criar uma árvore denominada Trie, a qual é construída de forma que as letras mais frequentes fiquem mais próximas da raiz e as letras menos frequentes fiquem mais distantes. Consequentemente, a primeira estrutura de dados implementada em nosso programa foi a `Trie`. A trie é uma árvore binária, e o caminho até um nó folha qualquer determina a sequência de bits que o representará. Por exemplo, na figura 1 a letra D é representada pela sequência 100, o qual é o caminho da raiz até o nó folha que a representa. Assim, no momento de escrita dos dados codificados no disco, gravaríamos a sequência

de bits 100 em vez da sequência 01000100 (D em binário), o que evidentemente resulta em uma economia de espaço.

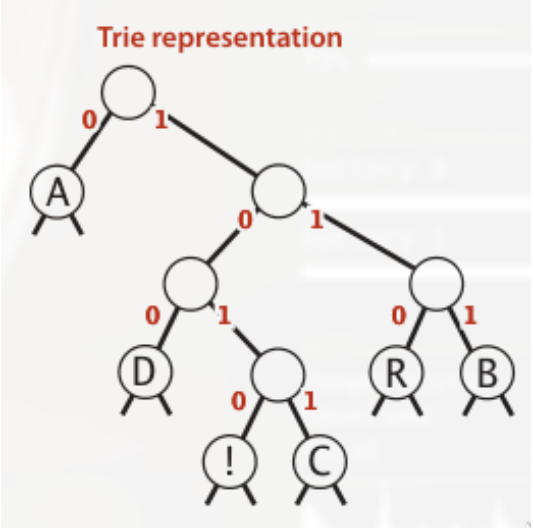

Figura 1: O código de Huffman em um trie

Além do mais, dada uma determinada string, para tornar a contagem da frequência de cada símbolo mais eficiente, implementados um `map`, o qual é baseado em uma árvore Red-Black para tornar os processos de inserção, remoção e busca mais eficientes.

Outras estruturas de dados de suporte implementadas foram a `MinPQueue` (fila de prioridade mínima), para auxiliar no momento da construção da Trie a partir das frequências de cada caractere, além do `Vector`, para manipular dados que precisam ser acessados em tempo $O(1)$.

2.3 Compressão

A compressão do programa abrange toda a codificação UTF-8. Dessa maneira, caracteres de diversos alfabetos como o árabe, japonês, russo e o latim podem ser compactados de forma que não haja perdas.

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Figura 2: Tabela de codificação UTF-8

Na codificação UTF-8, o número de bytes que representa cada caractere varia entre 1 a 4 bytes [3]. Para saber quantos bytes representam o caractere atual, os primeiros bits do primeiro byte de cada caractere informam quantos bytes o representam, conforme a figura 2.

Dessa maneira, durante o processo de compressão, o programa utiliza-se dessa lógica para conseguir interpretar cada caractere de uma string.

Por fim, durante o processo de compressão, um cabeçalho de tamanho variado é reservado no início do arquivo compactado. Este cabeçalho é utilizado para armazenar algumas informações

importantes para o processo de descompactação, como a trie de Huffman utilizada para codificar cada caractere.

2.4 Descompressão

O processo de descompressão é de fato o mais importante. De nada adianta compactar se depois não será possível descompactar.

Nesse sentido, a primeira tarefa realizada é ler o cabeçalho do arquivo compactado. Durante esse processo a trie é reconstruída, além da verificação de se o arquivo que será descompactado foi de fato compactado pelo presente programa. Isso é feito, pois o processo de compactação produz um arquivo específico para cada implementação do algoritmo de Huffman, o que implica que nem toda implementação do algoritmo será capaz de descompactar todos os binários produzidos por qualquer um dessas implementações.

3 Análise de Complexidade

3.1 Complexidade de tempo

A primeira etapa da compressão é contar a frequência de cada caractere da string. Para isso, toda a string deve ser lida, o que ocasiona uma complexidade de tempo $O(n)$. A contagem em si é realizada em nosso map. Cada caractere é armazenado de forma que a chave seja o caractere e o valor a sua frequência. Como nosso map foi implementado em cima de uma árvore Red-Black, o custo de acesso ao valor de uma determinada chave é $O(\log n)$ [1]. Portanto, a complexidade da contagem é $O(n \log n)$, uma vez que cada caractere da string será acessado no map para incrementar o seu valor. Como será mostrado posteriormente, a etapa da contagem de frequência é a mais demorada dentre as outras etapas do processo de compactação.

Em seguida, é realizado a construção da trie. Essa etapa envolve a ordenação dos elementos do map em uma fila de prioridade, com o intuito de buscar os caracteres menos frequentes antes dos mais frequentes [2]. Isso é necessário, pois a construção da trie é um processo *bottom-up*, isto é, os nós mais baixos são construídos primeiro, de forma que a raiz seja o último nó a ser construído.

A inserção na fila de prioridade mínima implementada em cima de uma lista ligada tem a complexidade $O(n)$ no pior caso, o que ocorre quando um elemento maior do que todos os outros precisa ser inserido no fim da fila. Para o caso médio, espera-se que a inserção requeira a iteração pela metade da lista, resultando em uma complexidade de tempo de $O(\frac{n}{2})$, sendo simplificado para $O(n)$ [1]. Portanto, o pior caso de construção da fila teria complexidade (n^2), o que ocorreria em um caminhamento na árvore que pegasse sempre o caractere mais frequente até aquele momento. Por outro lado, a remoção de um elemento da fila de prioridade mínima para a construção de um nó da trie tem complexidade $O(1)$. Logo, a construção completa da trie a partir da fila de prioridade mínima tem complexidade $O(n)$, no caso médio.

Por fim, ocorre a compressão do arquivo propriamente dita. Essa etapa envolve a leitura da string original, seguida da codificação de cada caractere utilizando a trie, e a gravação no arquivo binário. Com o intuito de diminuir a quantidade de acessos ao disco, um buffer de leitura e um buffer de escrita de 16kB são utilizados. Dessa maneira, o acesso para leitura e escrita são realizados de forma mais eficiente. Notavelmente, antes da implementação dos buffers, a leitura e escrita ocorria de byte em byte. Em média, a compactação de um arquivo de 2.1 GB demorava 20min, enquanto a descompressão do binário gerada durava 8min. Após a implementação dos buffers o mesmo arquivo foi comprimido em 8min e a sua descompactação durou 2min.

3.2 Complexidade de espaço

Carregar arquivos grandes na memória principal talvez não seja a melhor ideia, pois pode ocasionar a operação de swaps na memória externa [1]. Por outro lado, um grande número de acessos ao disco para gravação pode ser bastante dispendioso no tocante ao tempo de execução do algoritmo. Portanto, o compromisso assumido para preservar tanto a memória principal, quanto para otimizar o tempo de execução do programa é a utilização de buffers de 16kB, como já referido.

Já com relação às estruturas de dados, na trie cada nó corresponde a um caractere UTF-8. Dessa maneira, a complexidade de espaço necessário para armazenar a trie, como também a fila de prioridade mínima, seria $O(n)$, onde n é o número de caracteres diferentes em uma determinada string. Atualmente, a codificação UTF-8 pode representar 1.112.064 caracteres diferentes [3], sendo este o máximo número de nós em que a trie do nosso programa estaria sujeita.

4 Estratégias de Robustez

A estratégia de robustez empregada foi o controle de exceções, além dos testes implementados utilizando a biblioteca DOCTEST[1].

O controle de exceções foi implementado em múltiplas partes do programa. Para cada uma das estruturas de dados utilizadas, existe um conjunto de exceções que são lançadas e tratadas, conforme o necessário. Dessa maneira, erros malquisto são devidamente tratados, de forma que a experiência do usuário não seja desagradavelmente afetada. Além do mais, a compressão e a descompressão só são realizadas com arquivos válidos, isto é, a compressão só é realizada em arquivos codificados como UTF-8 e a descompressão só é realizada em binários com a assinatura do nosso programa.

Os testes foram implementados considerando o mais diversos tipos de cenários. Da mesma maneira que as exceções, para cada estrutura de dados implementada, há também um conjunto de testes realizados no intuito de confirmar a resiliência de tais estruturas.

5 Análise Experimental

Com o objetivo de testar a eficiência do programa, tanto com relação ao tempo de execução, como também com a taxa de compressão alcançada, uma bateria de testes foi realizada. Para automatizar esta etapa, dois scripts foram escritos.

O primeiro gera um arquivo com o tamanho solicitado em bytes. Com o intuito simular um texto, conjuntos com tamanhos variados de letras são gerados de forma pseuda-aleatória.

Já o segundo script é responsável por executar o programa em cada arquivo gerado pelo primeiro script, calculando os tempos de compressão e descompressão, e também a taxa de compressão alcançada. Os resultados são expostos abaixo.

5.1 Tempo para compressão

O tempo necessário para compactar um arquivo aumenta com uma tendência linear conforme o tamanho do arquivo aumenta. Essa característica condiz com a análise de complexidade de tempo realizada anteriormente, uma vez que os tempos médios estavam entre $O(n)$ e $O(\log n)$. Vale citar

[1]DOCTEST pode ser encontrado no github: `https://github.com/doctest/doctest`

que dentre as tarefas realizadas durante a compressão, a contagem de caracteres é a que apresentou maior custo com relação ao tempo para ser finalizada.

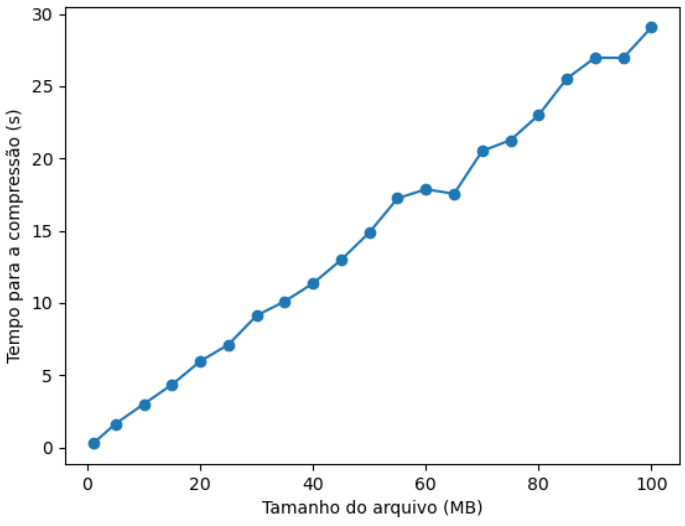

Figura 3: Tempo de compressão

5.2 Tempo para descompressão

Consoante ao gráfico do tempo de compressão, o gráfico de tempo de descompressão apresenta a mesma tendência linear. Isso pode ser explicado pelo fato de descompressão ser o processo inverso. A árvore é reconstruída e o arquivo binário é lido e descompactado. Um ponto interessante é que o tempo necessário para descompactar um arquivo é menor que o tempo necessário para compactá-lo. Nesse sentido, essa redução é o efeito de que a contagem dos caracteres não é necessária durante está etapa, o que ocasiona nessa queda significativa do tempo de descompressão.

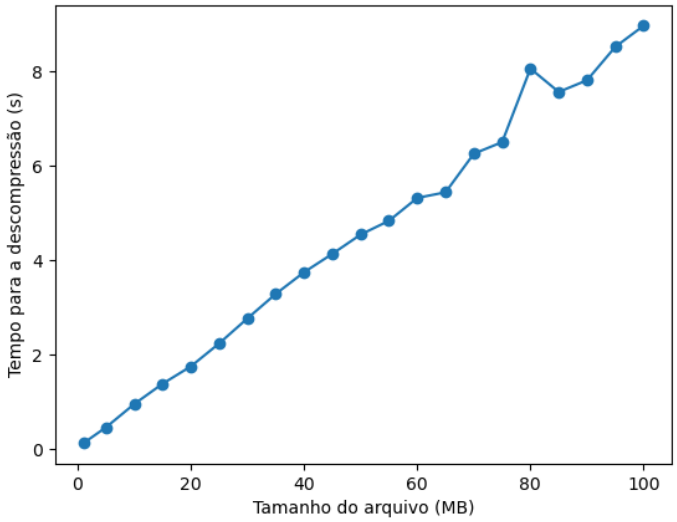

Figura 4: Tempo de descompressão

5.3 Taxa de compressão

O último teste realizado foi a verificação da taxa de compressão. A taxa de compressão de um arquivo é dada por $\frac{C(B)}{B}$, onde $C(B)$ é o número de bits do arquivo produzido pelo compressor (arquivo compactado) e B é o número de bits do arquivo original [2]. Nesse sentido, dado um texto com distribuição uniforme de caracteres, a taxa média de compressão fica entre 25% e 30%, o que é o esperado para algoritmo de Huffman [1].

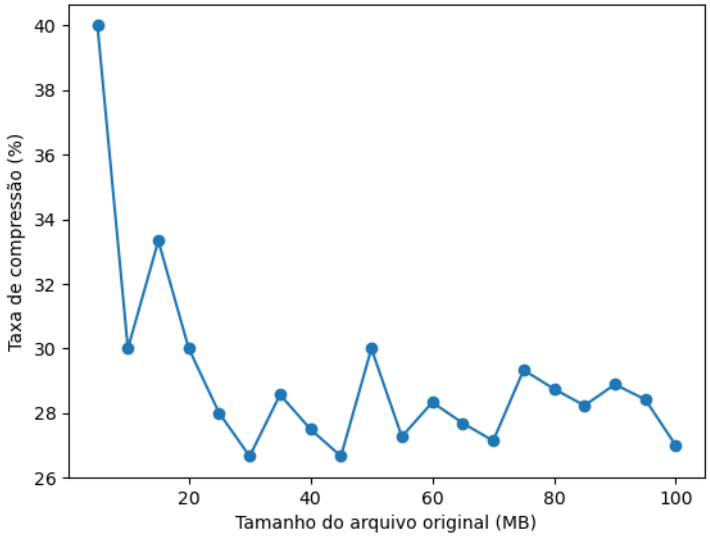

Figura 5: Taxa de compressão

6 Conclusões

Nesse trabalho foi proposto um programa que recebe um arquivo codificado como UTF-8 e o compacta, além de possibilitar sua posterior descompactação. Tal programa utiliza-se das estruturas de dados vector, map, fila de prioridade mínima e árvore binária para realizar as tarefas a qual foi proposto, além de dispor-se de um ferramental completo no tocante à tratativa de possíveis erros.

Por fim, a construção do referido programa possibilitou um amplo aprendizado sobre a manipulação das estruturas de dados citadas, além de reforçar os conceitos básicos de programação orientada a objetos. Além disso, a presente implementação exigiu o estudo do algoritmo de Huffman, além de estudos a parte para manipular a escrita e leitura de dados no disco de forma eficiente.

Referências

[1] Thomas H. Cormen et al. *Algoritmos - Teoria e Prática*. Elsevier Brasil, 2017. ISBN: 978-85-3527-179-9.

[2] Robert Sedgewick. *Algorithms, Fourth Edition: Book and 24-Part Lecture Series*. Addison-Wesley Professional, 2015. ISBN: 978-01-3438-468-9.

[3] *UTF-8 Encoding*. URL: `https://en.wikipedia.org/wiki/utf-8` (acesso em 27/06/2023).

Apêndice

Instruções para compilação e execução

OBS.: O programa foi desenvolvido utilizando o g++12. Gentileza executá-lo nesta versão do compilador. Se você estiver em um ambiente Ubuntu, o Makefile tentará compilar o programa com a referida versão do compilador. Caso ela não esteja disponível, você pode instalá-la utilizando o comando `sudo apt install g++-12`.

Uma vez no diretório raiz do programa, a compilação poderá ser realizada por meio do comando `make build`.

A compactação de um determinado arquivo pode ser feita utilizando a flag `-c` e a descompactação pode ser realizado utilizando a flag `-d`. Você pode usar o comando `make run` e passar as flags junto com o nome dos arquivos de entrada e saída, respectivamente.

Ademais, os testes e o valgrind podem ser executados via os comandos `make tests` e `make valgrind`, respectivamente.