Trabalho Prático 3
Compactação de Arquivos de Texto

Lucas Albano Olive Cruz – 2022036209

Departamento de Ciência da Computação (DCC) - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG – Brasil

lucasalbano@dcc.ufmg.br

1. Introdução

O problema proposto consiste em ajudar uma empresa de jornais a reduzir gastos com armazenamento de edições passadas do jornal. Assim, propõe-se utilizar o **algoritmo de Huffman** para compactar arquivos de texto. Dessa forma, o programa deve receber dois arquivos: um .txt com o conteúdo a ser compactado e um arquivo que após os passos do algoritmo possuirá o texto compactado.

A sequência de passos do programa proposto começa percorrendo todo o arquivo e registrando a frequência de cada caractere. Em seguida, criamos uma fila de prioridade mínima chaveada pela frequência de cada caractere. Com a fila de prioridade, montamos uma árvore que contêm em suas folhas os caracteres e suas frequências e a partir desta conseguimos construir uma tabela de codificação e então efetivamente começar a compactação do arquivo. Assim, percorremos novamente o arquivo e para cada caractere consultamos a tabela e registramos no arquivo de saída a sequência de bits correspondente. Armazenamos no cabeçalho do arquivo compactado a árvore utilizada para na etapa de descompactação remontá-la e partir do caminhamento na árvore recuperar o texto original.

2. Método

Alguns TADs foram necessários para a resolução do problema, assim como o uso de algoritmos como o algoritmo de Huffman. Nesta seção irei descrever as estruturas de dados utilizadas assim como os algoritmos e procedimentos necessários para a compactação e descompactação do arquivo de texto.

2.1. Estruturas de Dados

Ao todo precisamos utilizar três estruturas de dados auxiliares para realizar os passos do programa. Como veremos a seguir, somente com um tipo de nó conseguimos utilizar as três estruturas, sendo estas: **Árvore, Fila de Prioridade e Pilha**.

2.1.1. *Node*

Dessa maneira, nosso nó é a estrutura elementar do nosso programa, a partir dele construímos todas as demais estruturas. A classe *node* possui cinco atributos: um tipo *char* que armazena o caractere, um tipo *int* que armazena a frequência do caractere e três ponteiros para nós *left, right* e *next.* Os dois primeiros ponteiros são utilizados pela árvore e o último pela fila e pilha.

As funções são apenas *getters* e *setters* para estes atributos. Possui três construtores, um padrão quando nenhum parâmetro é fornecido que inicia todos os ponteiros como *nullptr* a frequência como zero e o caractere como nulo (‘\0’). Um construtor com parâmetros para o caractere e a frequência o qual inicia estes atributos conforme fornecidos e os ponteiros como nulo. Por fim, um construtor com dois outros nós como parâmetro que inicia o caractere como nulo, a frequência como a soma das frequências dos dois nós, o ponteiro para esquerda como o primeiro e o ponteiro para a direita como o segundo nó fornecido. Este último construtor irá nos auxiliar ao executar o algoritmo de Huffman descrito na Seção 2.2.1. O ponteiro *next* é inicializado como *nullptr* em todos os casos.

2.1.2. *Tree*

A classe árvore segue a estrutura usual de uma árvore binária, porém esta não é balanceada. Possui apenas um ponteiro para a raiz da árvore e funções simples *getters* e *setters*, construtor, destrutor, uma função para verificar se está vazia, limpar a árvore e medir sua altura.

A parte mais importante da classe está na função *build* que constrói a árvore a partir de uma fila de prioridade ou de uma pilha. A fila de prioridade é utilizada na compactação do arquivo e a construção da árvore a partir desta será explicada na seção 2.2.1. A pilha é utilizada quando queremos descompactar o arquivo e similarmente será explicada na seção 2.2.4.

2.1.3. *Queue*

Nossa classe fila é basicamente uma lista encadeada simples que é ordenada em ordem crescente pelo atributo frequência de cada nó. Assim, a classe possui um ponteiro para o primeiro elemento que chamamos de *head* e um atributo *size* que representa o tamanho atual da fila de prioridade mínima.

Além dos tradicionais construtores, destrutores, *getters* e *setters* as funções mais significantes da classe são *push* responsável por adicionar um elemento na fila e *pop* responsável por remover um elemento.

A função de inserção possui três casos. O primeiro é de que a fila está vazia, neste caso o elemento é inserido como o *head.* No caso de a fila não estar vazia, porém o elemento a ser inserido possuir frequência menor que o *head* ele passa a apontar para o *head* e se torna o novo *head* da fila. O último caso acontece quando não estamos em nenhum dos casos anteriores então buscamos o local da fila que o elemento deve ser inserido. Ao encontramos a posição o elemento imediatamente anterior passa a apontar ao elemento que está sendo inserido e este aponta para o elemento posterior. Em qualquer um dos três casos, após a inserção adicionamos um ao tamanho da fila.

Como a inserção é feita de forma ordenada, a função *pop* somente retorna o *head* que já será o elemento com a menor frequência da fila e assim torna o elemento imediatamente posterior a este o novo *head* da fila. Após a remoção o tamanho da fila decresce em um.

2.1.4. *Stack*

A classe pilha será usada para remontar a árvore na fase de descompactação. Esta segue as funcionalidades já conhecidas de uma pilha, assim possui somente dois atributos similares a fila: um elemento que representa o topo da pilha identificado por *top* e um atributo *size* para representar o tamanho da pilha.

Novamente, funções *getters, setters,* construtor e destrutor. A função *push* somente adiciona um novo nó ao topo da pilha e a função *pop* remove o topo da pilha, ambos atualizam o atributo *size*.

2.2. Algoritmos

Os algoritmos utilizados no programa manipulam as estruturas de dados anteriormente descritas com a finalidade de transformar um arquivo de entrada em um arquivo compactado e vice-versa. Assim, nesta seção irei descrever a utilização do algoritmo de Huffman e como se segue o passo-a-passo para compactar ou descompactar um arquivo de texto.

2.2.1. Algoritmo de Huffman

Para compactarmos um arquivo precisamos reduzir o tamanho da representação de cada caractere, portanto, um caractere que é representado por um byte passaria a ser representado por exemplo por somente dois ou três bits. O algoritmo de Huffman funciona definindo uma nova representação para um caractere de tamanho em bits variável a depender de sua frequência em determinado texto.

A nova codificação do caractere é definida pelo caminho necessário para chegar até este em uma árvore. Dessa forma, os caracteres sempre estarão nas folhas da árvore e a quantidade de passos para a esquerda ou para a direita, representados por 0 e 1 respectivamente, tomados na árvore definem a sequência de bits que representa o caractere.

O algoritmo constrói a árvore recebendo como entrada uma fila de prioridade mínima ordenada pela frequência do caractere, descrita na seção 2.1.3. Assim, removemos os dois primeiros elementos da fila que possuirão as duas menores frequências e formamos um novo nó com frequência igual a soma das frequências dos dois nós removidos e o inserimos novamente na fila. Esse processo continua até possuirmos somente um nó na fila que será então a raiz de nossa árvore. Um exemplo da execução do algoritmo pode ser visto na Figura 1.

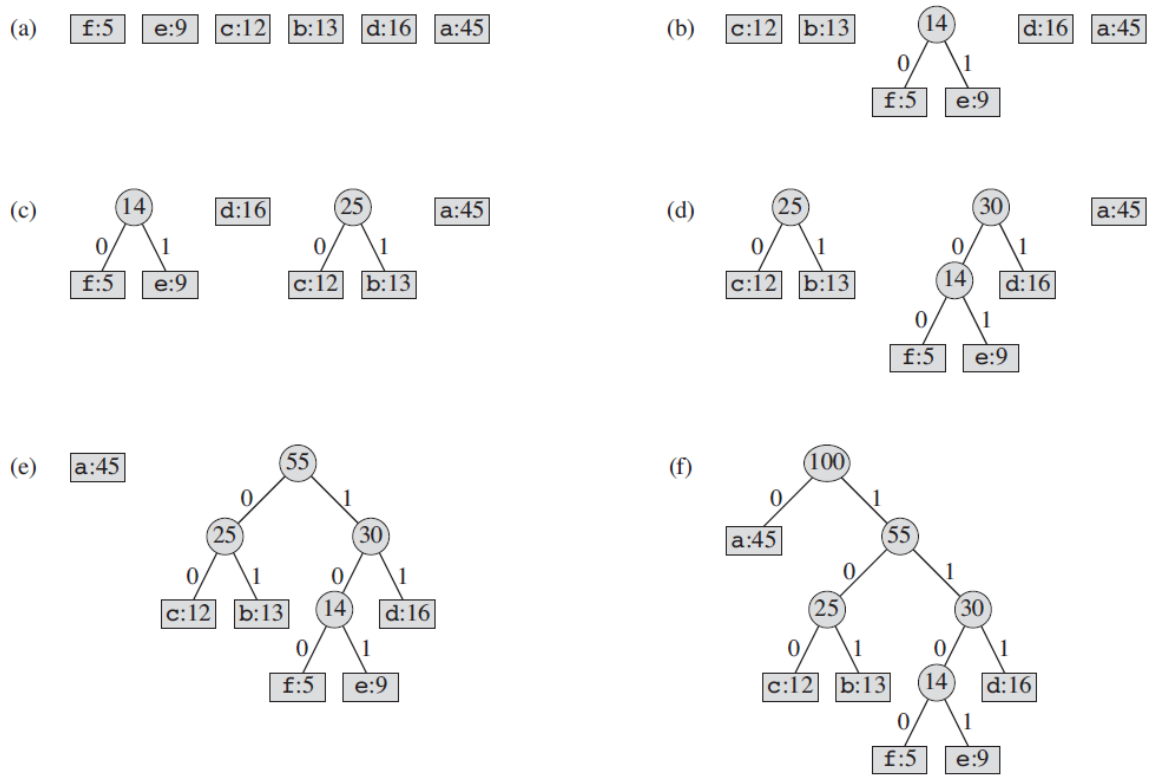


Figura 1 – Passos do algoritmo de Huffman. Retirado de *Introduction to Algorithms.*

2.2.2. Tabela de Codificação

Para compactarmos nosso arquivo precisamos encontrar a nova representação de um caractere sempre que o identificarmos no arquivo. Tentar toda vez encontrar essa representação caminhando pela árvore é bastante custoso, então utilizamos uma tabela de codificação para armazenar a sequência de bits que representa o caractere.

Esta tabela é uma matriz em que cada linha é a codificação de um caractere. Ela possui 256 linhas que são a quantidade de *chars* em C++. O tamanho de cada linha, ou o número de colunas, é a altura da árvore + 1 pois este número é a maior nova codificação de um caractere mais uma coluna para representar o fim da codificação.

Dessa forma, a tabela é construída ao percorrer toda a árvore e enquanto percorremos registramos 0 para cada passo a esquerda e 1 para cada passo a direita. Ao chegarmos em uma folha registramos na tabela o caminho obtido na posição correspondente ao caractere guardado na folha.

2.2.3. Compactação

A função de compactação recebe dois arquivos, o que deve ser compactado (*input*) e o que recebera o resultado da compactação (*output*). O primeiro passo da compactação é percorrer todo o *input* e contar a frequência de cada caractere, para isso utilizamos um vetor de tamanho 256 e para cada caractere usamos o índice ao convertê-lo para inteiro como sua

posição no vetor. Então, em cada posição do vetor com valor maior que zero significa que temos um caractere correspondente com frequência igual a encontrada na posição.

Assim, criamos um nó para esse caractere e o adicionamos à fila de prioridade. Após possuirmos a fila de prioridade construímos a árvore utilizando o algoritmo de Huffman. Iremos precisar desta árvore para descompactar o arquivo então a registramos no cabeçalho do *output* através de um caminhamento pós-ordem. Cada nó interno da árvore, e também as quebras de linha, são gravados como caracteres não utilizados da tabela ASCII para facilitar na reconstrução da árvore.

Com a árvore devidamente construída criamos então a tabela de codificação descrita na Seção 2.2.2. Conseguimos assim, finalmente começar efetivamente a compactação do arquivo. Percorremos todo o *input* e utilizando uma *string* como *buffer* para cada novo caractere gravamos no *buffer* o padrão encontrado na posição do caractere na tabela de codificação. Com o *buffer* reduzimos os *chars* dos 0's e 1's para *bits* e gravamos então no *output* o resultado. Ao fim, teremos no *output* o resultado da compactação.

2.2.4. Descompactação

A função de descompactação também recebe dois arquivos, o primeiro é um arquivo anteriormente compactado pelo nosso programa e o segundo um .txt que receberá o resultado da descompactação.

O primeiro passo é recuperar a árvore percorrendo o cabeçalho do arquivo. Fazemos isso utilizando uma pilha auxiliar. Assim, temos dois casos: Se o elemento for um caractere o empilhamos, se for um nó interno (representado por um caractere não utilizado da tabela ASCII) desempilhamos dois nós, tornamos estes os filhos a esquerda e à direita de um novo nó e empilhamos novamente. Ao chegar no fim do cabeçalho teremos apenas um nó na pilha que será a raiz da árvore.

Com a árvore construída conseguimos efetuar então a descompactação. Percorrendo nosso arquivo caminhamos na árvore para a direita ou para a esquerda ao ler respectivamente um 1 ou 0. Ao chegar em uma folha gravamos o caractere encontrado no arquivo de saída e voltamos a raiz da árvore. Ao terminarmos de percorrer todo o arquivo compactado teremos no arquivo de saída o nosso texto original.

3. Análise de Complexidade

Neste tópico iremos analisar a complexidade de tempo e de memória dos passos do programa proposto.

3.1. Compactação

Como descrito na Seção 2.2.3., o primeiro passo do programa é percorrer todo o arquivo e montar a tabela de frequência. Portanto, esse passo possui complexidade de tempo igual à $O(n)$ e como o vetor possui tamanho fixo possui complexidade de espaço igual à $O(256)$.

No próximo passo, montamos a fila de prioridade. A inserção possui um pior caso de $O(n)$ quando precisamos inserir um nó ao fim da fila. A remoção possui complexidade $O(1)$ pois sempre retiramos o primeiro elemento da fila. A complexidade de espaço é $O(n)$ sendo n a quantidade de diferentes caracteres encontrados no texto.

Ao montar a árvore o algoritmo de Huffman sempre executa $n - 1$ passos e para cada passo reinserimos um nó na fila de prioridade. Então a complexidade para montar a árvore será igual à $O(n^2)$ para o pior caso quando sempre que removermos os dois primeiros nós da fila e os combinarmos o novo nó seja inserido ao fim da fila. Como a fila usada é a mesma do passo anterior temos a mesma complexidade de espaço já que a fila nunca irá aumentar.

Para montar a tabela de codificação percorremos a árvore e sempre que encontramos uma folha inserimos o caminho feito à matriz. Portanto, a complexidade de espaço para percorrer toda a árvore é $O(n)$. A complexidade de espaço será igual à $O(256 \cdot h)$ sendo 256 as linhas da matriz e h a altura da árvore + 1.

Desta forma, o pior caso do programa possui complexidade de tempo $O(n^2)$ e de espaço $O(n)$ o que sempre irá variar dependendo das características do texto a ser compactado como tamanho e quantidade de caracteres diferentes.

3.2. Descompactação

Para descompactarmos o arquivo, primeiro remontamos a árvore. Assim, percorremos o cabeçalho e executamos o algoritmo descrito na Seção 2.2.4. Este algoritmo possui complexidade de tempo $O(n)$ que é o necessário para percorrer todo o cabeçalho já que as operações de combinação dos caracteres e nós, desempilhar e empilhar possuem complexidade $O(1)$. A complexidade de espaço será $O(n)$ já que a pilha sempre possuirá tamanho menor ou igual ao cabeçalho.

A descompactação efetivamente percorre todo o arquivo compactado e para cada bit caminha na árvore, ao encontrar uma folha grava o caractere no arquivo de saída e retorna ao início da árvore. Portanto, complexidade de tempo $O(n)$ e não precisamos de nenhum espaço adicional além da árvore.

4. Estratégias de Robustez

O programa possui somente três *flags* válidas (-c, -d e –h), qualquer opção além destas retorna um erro de opção inválida.

Caso o usuário não forneça arquivos suficientes é retornado um erro de quantidade de argumentos inválida.

Caso não seja possível abrir algum arquivo é retornado um erro informando que não foi possível abrir o arquivo.

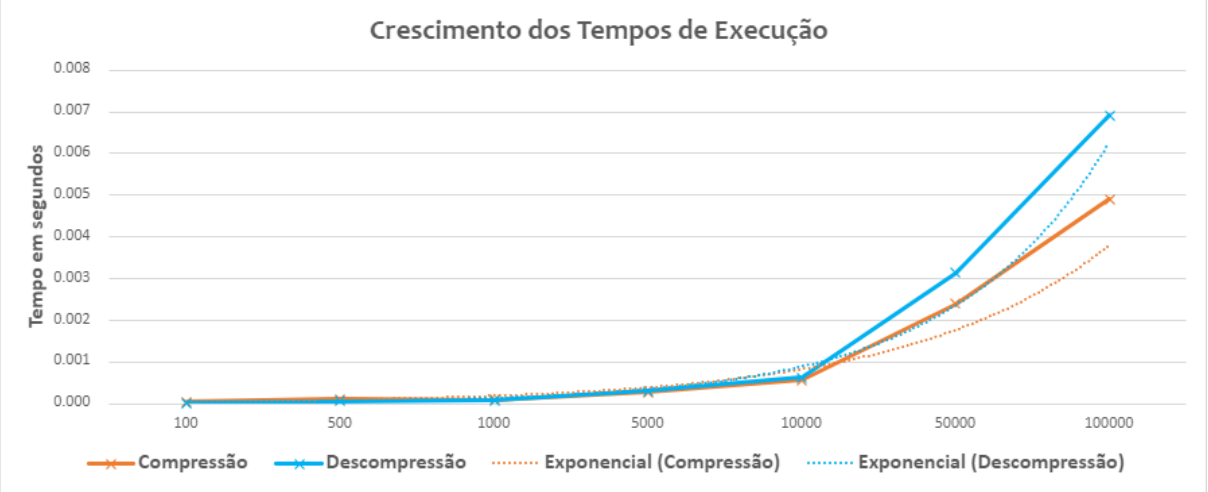
Seria interessante verificar se o arquivo a ser descompactado foi um arquivo anteriormente compactado pelo programa, por exemplo verificando a extensão. Não implementei a funcionalidade pois o arquivo de saída deve ser fornecido pelo usuário e não

tenho como controlar a sua extensão, então estou considerando que sempre será um .txt. No entanto, isso abre uma brecha para o usuário informar um arquivo qualquer para ser descompactado e quebrar o programa.

5. Análise Experimental

Para a análise experimental escolhi analisar o tempo gasto para a compactação e descompactação variando o tamanho do arquivo de texto e a taxa de compressão média do programa.

Assim, foi testado a compressão e descompressão de arquivos de texto de tamanho em bytes de 100, 500, 1.000, 5.000, 10.000, 50.000 e 100.000. Dessa forma, obtemos os seguintes resultados em termos de tempo de execução:



Podemos ver com o gráfico um crescimento exponencial de ambas as operações do programa. Para a compressão isso era o esperado devido a sua complexidade assintótica, no entanto a descompressão me surpreendeu devido a sua complexidade linear. Possivelmente o resultado inesperado se deve a manipulação dos binários a qual não tenho total compreensão, talvez com alguma correção nesta etapa a descompressão atinja crescimento linear.

A compressão média observada no programa foi de 43.33%. A taxa de compressão depende das características do arquivo de texto e pode variar de 20% a 90% [Cormen, 2022]. É interessante destacar que devido a necessidade de armazenar a árvore no cabeçalho arquivos pequenos podem acabar ficando com tamanho maior após passar pela compressão. Outra maneira de armazenar a árvore deveria ser estudada para contornar este problema.

6. Conclusões

O programa proposto consegue com sucesso compactar e descompactar arquivos de texto. A divisão dos passos do programa em etapas auxiliou na abstração do problema e melhor compreensão de como chegar ao resultado desejado.

Além disso, o programa pode obter um desempenho ainda melhor ao aprimorar a implementação da fila de prioridade, reduzindo a complexidade da inserção de novos elementos. Ainda, outra forma de armazenar a árvore no cabeçalho do arquivo pode melhorar significativamente a taxa de compressão, especialmente para arquivos de texto pequenos.

Por fim, as taxas de compressão do programa variam de 20% a 90% a depender das características do arquivo de texto, como tamanho e quantidade de diferentes caracteres.

7. Bibliografia

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. ISBN 978-0-262-04630-5. Pages 431–439 of section 15.3: Huffman codes.

PROGRAME SEU FUTURO. Algoritmo de Huffman em C. YouTube, 28 de julho de 2021. Disponível em: < https://www.youtube.com/playlist?list=PLqJK4Oyr5WShtxF1Ch3Vq4b1Dzzb-WxbP >. Acesso em: 30 de junho de 2023.

Apêndice A - Instruções para Compilação e Execução

Para compilar o arquivo primeiro descompacte o arquivo .zip e acesse a pasta raiz do projeto. Na pasta raiz execute o comando `make` para compilar. O executável será armazenado na pasta `bin` com o nome `programa`.

Para executar o programa acesse diretamente o programa com `./bin/programa` ou use o comando `make run`.

O comando `make run` executa o programa com a flag -h que exibe as instruções do programa. São estas:

- -c <arquivo a ser compactado> <arquivo compactado>
    - Compacta o primeiro arquivo e grava o resultado no segundo arquivo.
- -d <arquivo compactado> <arquivo descompactado>
    - Descompacta o primeiro arquivo e grava o resultado no segundo arquivo.
- -h
    - Exibe as instruções do programa.

Exemplo de execução:

```
./bin/programa -c test/input.txt test/binario.txt
./bin/programa -d test/binario.txt test/output.txt
```

Instruções adicionais do utilitário makefile:

- `make clean` - Deleta os arquivos .o e o executável.
- `make teste` - Deleta os arquivos .o e o executável, em seguida, compila o programa novamente e o executa primeiro compilando o arquivo `test/input.txt` e depois descompactando o `test/binario.txt` em `test/output.txt`.

A pasta `/test` contém três arquivos que podem ser utilizados para testar o programa através do comando `make teste`. O arquivo `input.txt` contém o texto a ser comprimido e pode ser alterado livremente. O arquivo `binario.txt` conterá o resultado da compactação e `output.txt` conterá o arquivo descompactado que será idêntico a `input.txt`.