

Trabalho Pratico 03

Compactador e descompactador de Huffman

Davi Sakamoto Lamounier - 2022035873

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

1. Introdução

O problema proposto foi implementar um programa que recebe como argumento dois arquivos de texto e uma flag “-c” ou “-d”. O programa então deve ser capaz de ler a armazenar o texto presente no primeiro arquivo, e baseado na flag de comando, no caso “-c” ele deve ser capaz de armazená-lo de forma compactada no segundo arquivo, ocupando menos espaço que o anterior, e no caso “-d”, deve ser capaz de descompactar o arquivo e reescrevê-lo em sua forma original no segundo arquivo. Para isso, o programa utiliza o método da Árvore de Huffman.

2. Método

2.1. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

A implementação do programa teve como base o uso da biblioteca “fstream”, que permite a manipulação de arquivos, aliado do uso de loops em que são usadas as funções “get()” e “getline()”, dependendo do contexto/função em que são necessitadas. A partir desse ponto, a implementação varia muito de acordo com a flag de entrada, por isso vamos tratá-los separadamente.

No caso de compactação, enquanto o arquivo é percorrido, os seus caracteres são contados separadamente, para determinar a frequência de cada um. Então são criados nós para cada tipo de caractere, que são ordenados pela sua frequência. Uma vez ordenados, a árvore é criada utilizando o método de huffman, para que todos os caracteres estejam em nós “folhas”, e os nós “pais” não representem caracteres, e sejam apenas um caminho para a raiz. Com isso, é criado um dicionário para os caracteres presentes no arquivo, onde o caminho da raiz até seu nó na árvore determina o seu código binário. Então o arquivo é percorrido novamente, e uma string de 0’s e 1’s é criada baseada na troca dos caracteres por seu código.

Agora que as preparações estão completas, começamos a escrever nosso novo arquivo. Em seu cabeçalho, é imprimido a quantidade de caracteres no arquivo original,

seguido pelo dicionário, que é escrito em duplas contendo o índice do caractere e seu código. Por fim, finalizamos a compactação com um loop que, aliado de manipulação de bits em caracteres, é utilizado para escrever a string de 0's e 1's em forma de caracteres, de maneira a ocupar 8x menos espaço.

No caso da descompactação, primeiramente o cabeçalho é lido e construímos uma tabela para o dicionário e a árvore correspondente. Então o resto do arquivo é percorrido, construindo uma string de 0's e 1's a partir da representação binária dos caracteres lidos. Por fim, basta percorrer a árvore utilizando o caminho na string para reescrever o arquivo de texto original.

2.1. Estrutura de Dados e Classes

Para modularizar a implementação do programa, foram criadas três classes. A primeira delas é a classe “Nó”, que é a mais simples, apesar de não ser um nó convencional. Diferente de outras mais comuns em árvores, essa classe tem, além de dois ponteiros para seus dois filhos, há também um terceiro ponteiro para um “próximo” nó. Isso porque para facilitar a implementação e diminuir a necessidade de criação de mais objetos e outra classe, os mesmos nós utilizados na árvore são primeiramente células de uma lista encadeada. Além disso, a classe possui dois atributos privados para representar o índice de seu caractere e sua frequência.

A segunda classe é uma lista encadeada de nós, que é utilizada para ordená-los. Essa classe já passa a ser um pouco mais complexa por conta de sua implementação encadeada, que utiliza de ponteiros para o primeiro e último nó, e depende dos ponteiros internos dos nós para relacioná-los. Seus métodos são em sua maior parte como os de uma lista encadeada comum, como construtor/destrutor, funções get/set e funções para inserir, remover e limpar a lista.

A terceira e última classe, que é a mais complexa do programa, é uma árvore de Huffman, fazer a maior parte das operações do programa. A princípio, ela funciona como uma árvore binária comum, porém possui diversos métodos, dentre eles vale destacar a sua construção a partir de um vetor de frequência de caracteres ou uma tabela dicionário, a geração dessa tabela e a descompactação de textos. Essas funções utilizam alguns métodos privados recursivos da árvore, como inserções, remoções e caminhamentos.

2.2. Métodos

Primeiramente falaremos sobre as funções utilizadas na compactação.

A função `constroiHuffman()` é responsável por construir a árvore de Huffman com base em uma tabela de frequência de caracteres. Primeiro, é criada uma lista de nós para armazenar os caracteres e suas frequências. Em um loop, os caracteres com frequência maior que zero, ou seja, que estão presentes no arquivo, são inseridos na lista. O algoritmo de Huffman é executado enquanto a lista de nós tiver mais de um elemento. A cada iteração, os dois nós de menor frequência são removidos da lista. Em seguida, é criado um novo nó com frequência igual à soma das frequências dos nós removidos, que passam a ser seus filhos. O novo nó é inserido novamente na lista. Após

o término do algoritmo, o último nó restante na lista é atribuído como raiz da árvore de Huffman. Por fim, a função libera a memória alocada para a lista de nós.

Vale destacar a função `insereNo()` da Lista que, além das operações comuns de uma função de inserção em lista encadeada, também confere se o caractere inserido é maior ou igual ao caractere a sua esquerda, o deslocando para o lado até que isso seja satisfeito. Dessa forma, a lista se mantém sempre ordenada.

A função `atualizaTabela()` é responsável por gerar/atualizar a tabela de mapeamento de caracteres para seus respectivos caminhos na árvore de Huffman. Ela chama a função recursiva privada `caminhoRecursivo()` com os parâmetros adequados.

A função `caminhoRecursivo()` é uma função auxiliar que percorre a árvore de Huffman de forma recursiva escrevendo o caminho percorrido e o enviando ao dicionário. No corpo da função, são realizadas chamadas recursivas para os nós filhos, concatenando 0 no caminho para a esquerda e 1 para a direita. É sempre verificado se o nó atual é um nó folha ou não. Se for, significa que representa um caractere, portanto o caminho atual é atribuído à posição correspondente na tabela de mapeamento. Dessa forma, a função `caminhoRecursivo()` percorre todos os nós da árvore de Huffman, construindo os caminhos binários para cada caractere e atualizando a tabela de mapeamento. Ao final da execução da função `atualizaTabela()`, a tabela estará completa.

Um método utilizado na função `main` que não é uma função separada, mas vale ser comentado, é a transformação da string de 0's e 1's para uma sequência de caracteres. Nela, durante um loop, os caracteres da string são lidos em grupos de 8, em que é utilizada aritmética binária para gerar um char com representação binária igual ao subconjunto analisado da string. Então esses caracteres são imprimidos no arquivo.

Agora serão analisados os métodos utilizados durante a descompactação.

A função `constroiTabela()` é responsável por construir a árvore de Huffman com base na tabela de mapeamento. Ela inicializa a raiz da árvore, e em seguida, itera sobre a tabela e, para cada elemento não vazio, chama a função `insereRecursivo()` passando os parâmetros necessários.

A função `insereRecursivo()` é uma função auxiliar que percorre a árvore de Huffman de forma recursiva para inserir os caracteres de acordo com os caminhos binários. Quando não existe nó filho para prosseguir esse caminho, ele é criado, e ao fim do caminho, o nó recebe os dados do seu caractere correspondente. Ao final da execução da função `constroiTabela()`, a árvore de Huffman estará construída corretamente com base na tabela de mapeamento, com caracteres nos locais apropriados e sempre em nós "folha".

A função `descompacta()` é responsável por descompactar uma string compactada codificado usando a codificação de Huffman. Dessa forma, a função percorre a string, seguindo o caminho na árvore de Huffman para a esquerda ou direita de acordo com os bits '0' e '1', respectivamente, presentes na string. Quando um caractere válido é encontrado, ele é escrito no arquivo de saída, e a iteração continua até que a string

termine ou o número de caracteres do arquivo original seja alcançado. Isso permite a descompactação dos dados originalmente comprimidos.

3. Análise de Complexidade

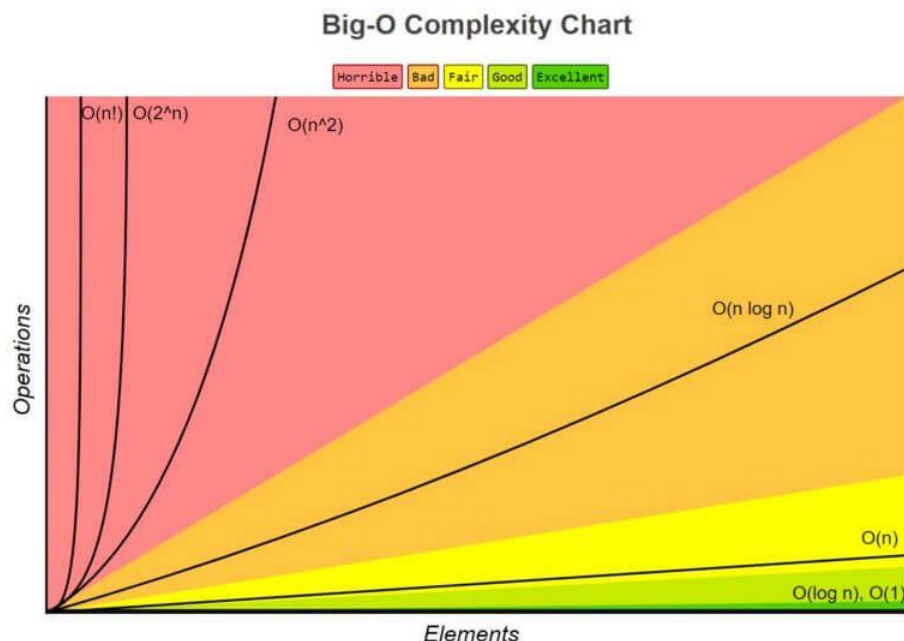
ConstroiHuffman(): A função começa com uma iteração sobre a tabela de frequência, etapa que realiza 256 iterações, ou seja, $O(256)$. Em seguida, os nós são agrupados em pais e filhos até que reste apenas um nó na lista. Essa etapa depende da entrada e das distribuições de frequência, além disso, a inserção de um elemento de forma ordenada na lista é de ordem $O(n)$. No fim, a função tem uma complexidade que varia de $O(255)$ a $O(n \log n)$, dependendo da distribuição das frequências dos caracteres.

CaminhaRecursivo(): A complexidade da função está diretamente relacionada ao número de nós presentes na árvore de Huffman e ao seu formato. Em uma árvore completamente balanceada, onde todos os nós possuem dois filhos, a complexidade seria $O(n)$, onde "n" é o número total de nós na árvore. Isso pois o caminhamento funciona de forma recursiva passando por todos os caminhos possíveis, porém sem a necessidade de voltar a raiz para cada folha. No entanto, a complexidade pode variar dependendo da estrutura da árvore, sendo reduzida em casos de árvores esparsas ou degeneradas.

CaminhaRecursivo(): A complexidade da função está diretamente relacionada ao tamanho do dicionário, ou seja, ao número de caracteres que possuem códigos binários atribuídos. Em um cenário ideal, onde todos os caracteres possuem códigos binários, a complexidade seria $O(n)$, onde "n" é o número de caracteres no dicionário. No entanto, se o dicionário for esparso e possuir apenas alguns caracteres com códigos binários atribuídos, a complexidade será reduzida proporcionalmente ao número de caracteres com códigos binários.

InserereRecursivo(): A complexidade da função depende do tamanho do código binário do caractere a ser inserido. Em um cenário ideal, a complexidade será $O(\log n)$, onde "n" é o número de bits do código binário. Isso pois a função utiliza do princípio de dividir para conquistar, que a partir do teorema mestre chega nesse resultado. No entanto, se o código binário for irregular, a complexidade pode variar e ser afetada pela estrutura da árvore.

Descompacta(): A complexidade da função pode ser considerada como $O(n)$, onde "n" é o número de bits da sequência compacta. Isso porque para cada bit lido, uma operação de caminhamento para nó filho esquerdo ou direito é realizada, e isso é repetido até que todos os bits da sequência sejam lidos ou o número de caracteres original seja alcançado.



4. Estratégias de Robustez

Para garantir mais robustez e a corretude do programa diante da entrada do usuário, vários critérios são verificados e testes são feitos, mantendo o funcionamento quando possível, e emitindo avisos de erros e parando o programa quando necessário.

Em primeiro lugar, há uma verificação da quantidade de argumentos da linha de comando, que deve ser feita com 4 argumentos, no formato de executável + nome do arquivo de entrada + nome de arquivo de saída + flag de comando ("-c" ou "-d"). Caso ela seja feita de forma incorreta, o usuário é alertado e instruído sobre a forma certa de executar o programa. Além disso, também é verificado se a flag de comando está no padrão correto. Em seguida, há uma verificação da abertura dos arquivos de entrada e saída, para evitar problemas em casos como arquivo inexistente, nome errado ou permissão negada. Então chegamos na análise da entrada.











Ao longo do programa, diversos critérios vão sendo analisados para evitar defeitos no funcionamento, com blocos try-catch para casos de entrada inválida. Verificações lógicas também são realizadas, como a condição de imprimir no máximo a quantidade original de caracteres do arquivo, para evitar a leitura de bits adicionais que podem ocasionalmente estarem na string para completarem bytes.

5. Análise experimental

Para evitar vazamentos de memória, utilizei o programa Valgrind ao longo do desenvolvimento para identificar possíveis erros. Em alguns momentos foi possível

encontrar blocos de memória que não estavam sendo liberados adequadamente, e várias vezes ocorreu de o programa tentar acessar blocos de memória indevidos, gerando segmentation fault, porém com uso do Valgrind aliado de técnicas de depuração foi possível resolver todos os problemas.

Em outros momentos em que o programa não se comportava da maneira esperada, foi muito utilizado o gdb para analisar com mais precisão o que estava acontecendo em cada função e descobrir a causa do erro. Além disso, vários arquivos de testes “entradaN.txt” foram criados na pasta “bin” para analisar como o programa responderia a diferentes tipos de entradas, com tamanhos diversos e diferentes distribuições de caracteres. Abaixo seguem as suas respectivas saídas com seus tamanhos.

 entrada.txt	04/07/2023 20:42	Documento de Te...	99 KB
 entrada2.txt	04/07/2023 20:44	Documento de Te...	5 KB
 entrada3.txt	04/07/2023 20:44	Documento de Te...	3 KB
 entrada4.txt	04/07/2023 20:44	Documento de Te...	20 KB
 entrada5.txt	04/07/2023 20:44	Documento de Te...	32 KB
 saida.txt	04/07/2023 20:42	Documento de Te...	54 KB
 saida2.txt	04/07/2023 20:44	Documento de Te...	2 KB
 saida3.txt	04/07/2023 20:44	Documento de Te...	2 KB
 saida4.txt	04/07/2023 20:44	Documento de Te...	3 KB
 saida5.txt	04/07/2023 20:44	Documento de Te...	20 KB

EXPLICAÇÃO DAS ENTRADAS COM TAXAS DE COMPRESSÃO:

Entrada1: Texto genérico muito longo

Entrada2: Repetição da mesma palavra separada por espaços

Entrada3: Texto curto com muitos caracteres diferentes

Entrada4: Repetição de apenas um tipo de caractere

Entrada5: Texto longo com distribuição balanceada de caracteres

Por fim, também foram realizados alguns testes de tempo através da biblioteca “chrono”, para analisar o desempenho de cada método e ver se eles se comportam como o esperado.

6. Conclusões

O trabalho lidou com problemas de leitura, armazenamento, ordenação, compressão e descompressão de diferentes textos, exigindo a criação de TAD's que representassem nós para armazenar os caracteres e a árvore de Huffman para compactá-los, junto à implementação de algoritmos de caminhamento e outras funções para concluir nosso objetivo.

Podemos concluir que o trabalho está consistente no que se diz respeito à complexidade em suas funções. Está funcional em respeito a realização das duas funções requisitadas e eficiente tanto em questão de tempo de execução quanto de compressão.

Houve dificuldade em casos com entradas muito extensas e com diversos caracteres, além do desenvolvimento de métodos para escrever os bits no arquivo e formas para o cabeçalho não afetar tanto a taxa de compressão.

Através do desenvolvimento das funções aliado da depuração, testes e análises de complexidade, tempo e espaço, promoveu-se a prática e um entendimento mais aprofundado de algoritmos de compactação e descompactação, e de aritmética binária, além da manipulação de objetos interdependentes e o seu uso em algoritmos complexos, reforçando a sua importância não só dentro da parte teórica, mas também como fator importante a ser levado em conta durante o desenvolvimento. Além disso, o exercício da implementação e do uso de estruturas de dados também foi praticado.

7. Bibliografia

- Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011
- Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle.
- Cormen, T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009

8. Instruções para compilação e execução

O programa requer os arquivos de entrada na pasta bin.

Basta escrever no terminal “make run” ou “make all” que será compilado o programa e serão gerados os arquivos “.o” na pasta “obj”, além do gmon.out no diretório raiz e o executável na pasta “bin”. Caso deseje testar entradas diferentes das que já estão lá, é necessário adicionar um arquivo de entrada na pasta bin e adicionar o comando de execução no Makefile no modelo:

```
$(EXE) $(BIN)/arquivodeentrada.txt $(BIN)/arquivodesaida.txt <flag>
```

Sendo que:

- 1- \$(EXE) é o executável
- 2- \$(BIN) é o endereço da pasta bin
- 3- “arquivo.txt” é o nome do arquivo de texto fornecido

4- <flag> é a opção de compactação (“-c”) ou descompactação (“-d”)

Para limpeza dos arquivos gerados durante a compilação, basta escrever “make clean” no terminal.