Trabalho Prática 3

Compactação de Arquivos de Texto

Eduardo Silveira Cezar Fernandes – 2021019424

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

Belo Horizonte – MG – Brazil

eduardoscf1210@gmail.com

1. Introdução

Esta documentação lida com o problema de compactar um arquivo texto. Para concretizar a funcionalidade proposta, o programa é utilizado com 2 funcionalidades principais, são elas: a compactação para diminuir o tamanho do arquivo original e a descompactação para verificar se o arquivo se manteve igual.

2. Método

2.1. Configurações da máquina

Sistema operacional: Ubuntu 20.04 LTS

Linguagem de programação: C++

Compilador: G++ / GNU Compiler Collection

Processador: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz 2.40 GHz

Memória RAM: 12,0 GB

2.2. Estruturas de dados

As estruturas de dados utilizadas nesse trabalho foram a lista ordenada e a árvore binária. O uso dessas estruturam específicas foram úteis primeiro para armazenar todos os caracteres e gerar a tabela frequência que cada um ocorre, e a árvore serviu para dizer a ordem que elas aparecem.

2.3. Classes

Na implementação desse trabalho, foram utilizadas 2 classes, a No e a lista.

A classe No é onde cada caractere é armazenado, e possui a frequência com que ele se repete, um ponteiro para o caractere a esquerda e outro a direita e o próximo.

Já a classe lista possui um ponteiro para o seu início, um tamanho, e métodos relacionados a própria lista ordenada como inserirOrdenado e preencherLista.

Observação, a árvore é montada dos nós porém esses métodos ficaram foram da definição da própria classe pois se trata da árvore.

2.4. Funções

O código possui 2 funções principais, a compactar() e a descompactar(). Porém para elas funcionarem há diversas funções ao redor.

Para compactar, é necessário usar a descobrirTamanho() primeiro que verificar qual é o tamanho do arquivo texto original para conseguir alocar a memória corretamente futuramente. Em seguida, usamos a inicializaTabelaComZero() e a preencheTabelaFrequencia(), como o nome já indica, criando a tabela de frequência dos caracteres, depois usamos o preencherLista que irá criar a lista ordenada conforme a quantidade de vezes que cada caractere apareceu na tabela de frequência, após isso, criamos a árvore com a montarArvore() e o dicionário com a gerarDicionario, codificamos esse texto e o descodificamos para finalmente podermos utilizar a compactar() com todas as informações que foram fornecidas até agora.

Já para descompactar, recriamos todos os passos da função compactar com o auxílio de um arquivo de apoio que salvou as informações e assim usamos a descompactar().

Observação, quanto mais vezes um caractere apareceu no texto original, maior será a sua altura na árvore binária.

3. Análise de complexidade

descobrirTamanho() – Complexidade de tempo: Possui complexidade O(n), sendo n o número de caracteres do arquivo original pois possui um loop que é percorrido enquanto o arquivo não termina.

lerTexto() – Complexidade de tempo: Assim como a função anterior, essa também percorre o arquivo original inteiro, com isso, sua complexidade de tempo é O(n).

inicializaTabelaComZero() – Complexidade de tempo: Também sera O(n) pois possui um loop com o tamanho da tabela ASC2 extendida, ou seja, n=256.

preencheTabelaFrequencia() – Complexidade de tempo: Já nessa função, ela deve percorrer o texto que foi lido inteiro, então sua complexidade será O(n) porém n é o número de caracteres do texto.

inserirOrdenado() – Complexidade de tempo: No pior caso, em que o elemento a ser inserido tem frequência menor que todos os elementos da lista, a função percorre a lista até o final, o que resulta em uma complexidade de tempo O(n), onde n é o número de elementos na lista.

No melhor caso, em que o elemento a ser inserido tem frequência maior que o primeiro elemento da lista, a função simplesmente insere o elemento no início da lista, resultando em uma complexidade de tempo O(1).

No caso médio, a função percorre a lista até encontrar o local correto de inserção, resultando em uma complexidade de tempo média O(n/2), onde n é o número de elementos na lista.

preencherLista() – Complexidade de tempo: Ela possui um loop que percorre a tabela ASC2 e ainda dentro desse loop possui a função inserirOrdenado, com isso, sua complexidade será O(n) com n = 256.

montarArvore() – Complexidade de espaço: A função utiliza três ponteiros: primeiro, segundo e novo, para armazenar os nós da lista que estão sendo combinados. Além disso, a função cria um novo nó novo a cada iteração do loop.

Portanto, a complexidade de espaço da função montarArvore é proporcional ao número de nós criados durante o processo de montagem da árvore. Se a lista contém n nós, a função criará n-1 nós adicionais para a construção da árvore.

Assumindo que a lista inicial contém n nós, a complexidade de espaço é O(n).

alocaDicionario() – Complexidade de espaço: O(TAM * colunas), onde TAM é o tamanho do array dicionario e colunas é o número de colunas alocadas para cada linha.

A função aloca um array bidimensional dicionario de tamanho TAM por colunas. Primeiro, é alocado espaço para o array de ponteiros dicionario com o tamanho TAM usando malloc. Em seguida, é alocado espaço para cada linha do array bidimensional usando calloc, o que resulta em uma alocação de colunas bytes para cada linha.

Portanto, a complexidade de espaço é proporcional ao produto de TAM e colunas, ou seja, O(TAM * colunas).

geraDicionario() – Complexidade de espaço: O(colunas), onde colunas é o número de colunas do array bidimensional dicionario.

A função gerarDicionario utiliza duas variáveis locais, esquerda e direita, que são arrays de caracteres com tamanho colunas. Essas variáveis são usadas para construir os caminhos do dicionário, adicionando os caracteres "0" e "1" respectivamente.

Portanto, a complexidade de espaço da função é proporcional ao tamanho desses arrays locais, ou seja, O(colunas). O restante das variáveis e parâmetros da função não afetam a complexidade de espaço, pois são ponteiros ou variáveis de tamanho fixo.

codificar() – Complexidade de espaço: O(tam), onde tam é o tamanho da string resultante da codificação do texto.

A função começa alocando espaço para a string codigo usando calloc com tamanho tam. O tamanho tam é calculado pela função calculaTamanhoString, que determina o tamanho necessário para armazenar o texto codificado.

Em seguida, a função entra em um loop while para percorrer cada caractere do texto e concatenar a codificação correspondente a esse caractere na string codigo usando strcat. A concatenação dos códigos de cada caractere é realizada de forma incremental.

Portanto, a complexidade de espaço da função é proporcional ao tamanho da string codigo, ou seja, O(tam).

Decodificar() – Complexidade de espaço: O(n), onde n é o tamanho do texto decodificado.

A função começa alocando espaço para a string decodificado usando calloc com tamanho strlen((const char*)texto). Isso permite que a string decodificado tenha o mesmo tamanho do texto decodificado.

Em seguida, a função entra em um loop while para percorrer cada caractere do texto. Dentro do loop, são feitas operações de navegação na árvore de decodificação, alterando o nó auxiliar aux com base nos caracteres do texto. Quando um nó folha é alcançado, o caractere correspondente é adicionado à string decodificado usando strcat.

A quantidade de espaço utilizado pela função é determinada pelo tamanho da string decodificado, que será igual ao tamanho do texto decodificado. Portanto, a complexidade de espaço da função é O(n).

compactar() – Complexidade de espaço: O(1), ou seja, constante.

A função utiliza algumas variáveis locais, como arquivo, mascara, byte, i e j, mas essas variáveis têm um tamanho fixo e não dependem do tamanho da entrada.

A única operação que poderia consumir espaço adicional é a abertura do arquivo arquivo usando fopen, mas isso não é considerado na complexidade de espaço da função, pois é uma operação fora do escopo da função em si.

Portanto, a complexidade de espaço da função compactar é constante, O(1).

descompactar() – Complexidade de espaço: Depende do tamanho do arquivo descompactado, mas podemos analisar a complexidade de espaço em relação ao processamento da função.

A função utiliza algumas variáveis locais, como arquivoCompactado, arquivoDestino, aux, byte e i, mas essas variáveis têm um tamanho fixo e não dependem do tamanho da entrada.

A função também utiliza a função auxiliar ehBitUm, que calcula se um bit específico de um byte é 1. Essa função utiliza uma variável local mascara, mas novamente, ela tem um tamanho fixo e não depende do tamanho da entrada.

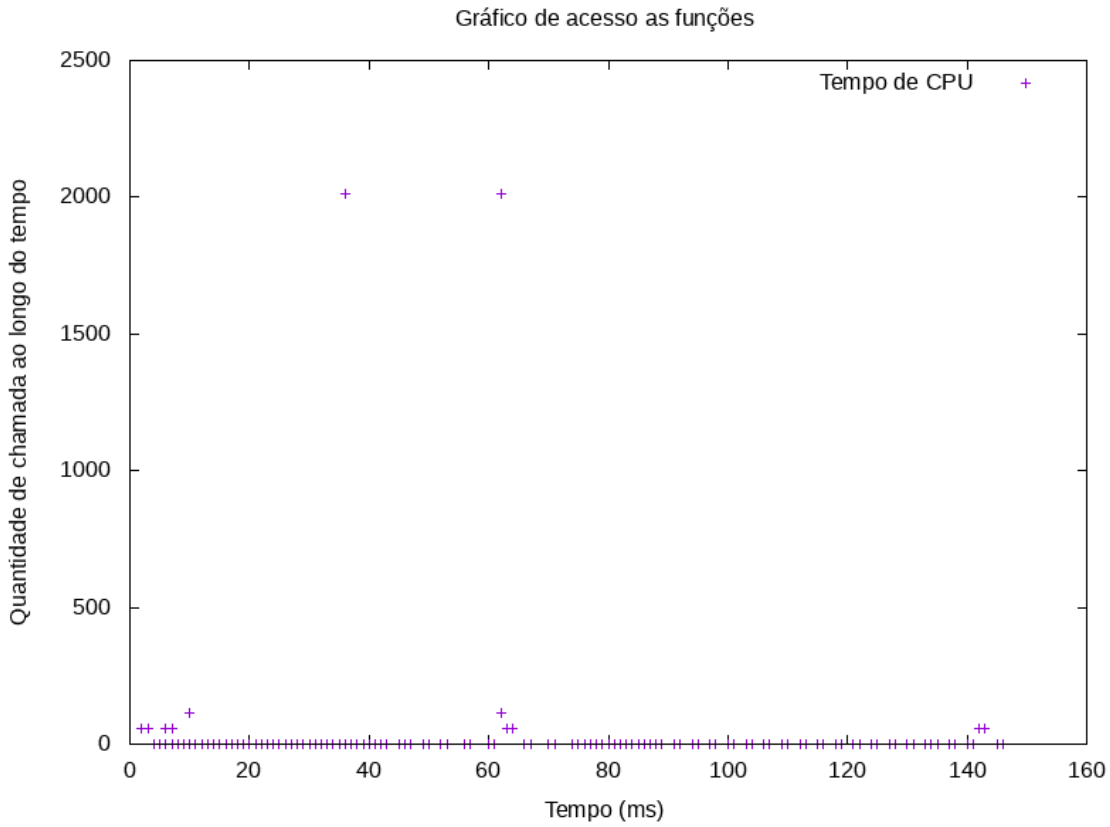
A única operação que pode consumir espaço adicional é a abertura dos arquivos arquivoCompactado e arquivoDestino usando fopen, mas isso não é considerado na complexidade de espaço da função, pois é uma operação fora do escopo da função em si. Portanto, a complexidade de espaço da função descompactar é constante, O(1).

4. Análise de robustez

Para padronizar o código, todas as variáveis foram nomeadas de acordo com a sua função, além da indentar corretamente o código para ajudar na leitura. Também foram adicionados comentários esclarecedores por todo o programa.

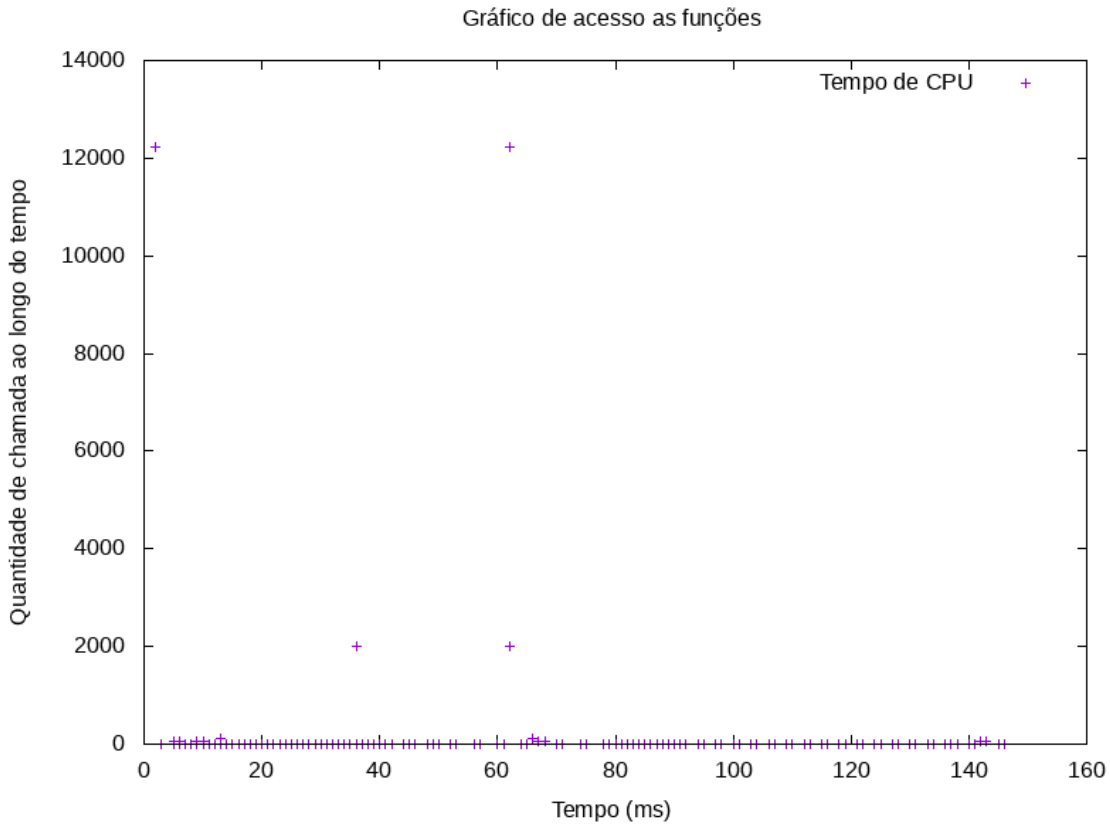
Para tratar erros, o programa emite um aviso caso não seja possível abrir o arquivo de leitura corretamente e outro caso o formato do comando da linha no terminal não estiver correto. Já na hora de debugar o código, foi utilizado o GDB para entender passo a passo o que estava acontecendo com o programa. Também houve o uso do Valgrind para garantir que não estava acontecendo vazamento de memória.

5. Análise experimental



Esse primeiro gráfico nos mostra que durante duas etapas do processo de compactação, há 2 momentos em que a quantidade de chamadas acontecendo

ao mesmo tempo é maior, nesta execução, um valor de 2000. Isso ocorre pois na montagem da árvore binária de huffman, há o uso de recursividade, o que normalmente eleva o valor citado.



Já nesse gráfico, ele mostra a descompactação, desta vez o número foi mais elevado pois como será melhor elaborado na conclusão, a minha descompactação gera novamente a lista ordenada, a árvore e o dicionário, isso pode até ser observado pois temos novamente aqueles pontos próximos ao 2000 na quantidade de chamadas, e ainda serão executadas todas as funções para descompactar.

Observação: Este exemplo foi realizado com um texto de tamanho mediano (50kb) e funcionou tudo perfeitamente. Porém, quando testei um arquivo maior (4MB), ele demorou aproximadamente 30 minutos para funcionar, justamente por esse número enorme de chamadas recursivas.

6. Conclusão

Dessa forma, o programa criado fornece corretamente o arquivo compactado e também é possível descompactá-lo sem perder o conteúdo do texto original.

Durante sua modelagem e execução, a maior dificuldade foi para conseguir descompactar o arquivo gerado anteriormente numa execução diferente, pois preciso da tabela frequência para poder descompactar corretamente. Com isso, a minha solução foi escrevê-la em binário no arquivo compactado antes da compactação do texto, e então gerar a lista ordenada, a árvore e o dicionário novamente, e na hora de descompactar, quando chega no caractere “ ` “, eu sei que começou o texto pois esse é o meu divisor entre tabela de frequência e texto.

7. Bibliografia

Márcio Costa Santos e Wagner Meira Jr. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Algoritmo de Huffman em C, disponível em:

<https://www.youtube.com/playlist?list=PLqJK4Oyr5WShtxF1Ch3Vq4b1Dzzb-WxbP>

Acessado dia 22 jun. 2023

8. Instruções de compilação e execução

Diretamente do diretório TP3 utilize o comando “make”. Em seguida, utilize “./bin/main nomeDoArquivoFonte nomeDoArquivoDestino (-c ou -d)”, sendo que nomeDoArquivoFonte deve ser substitúido pelo nome do arquivo que será lido ou descompactado e o nomeDoArquivoDestino pelo nome do arquivo onde a informação será armazenada, seja para o texto compactado ou o descompactado. Já o -c serve para compactar e -d para descompactar. Observação: O texto que será compactado também deve estar na pasta raiz.

Um exemplo de linha de comando para o uso do programa e compactar seria:

make

./bin/main texto.txt arquivoCompactado.wg -c

E para descompactar:

./bin/main arquivoCompactado.wg textoDescompactado.txt -d