Trabalho Prático 3: O problema de compressão de mensagens de Rick Sanchez

Paulo Henrique Maciel Fraga 2018054451
Universidade Federal de Minas Gerais (UFMG)

Resumo. *Documentação do trabalho prático aplicado pela Professora Olga N. Goussevskaia para a turma de Estrutura de Dados 2019/2.*

1. Introdução

O problema proposto no trabalho é, recebida uma mensagem com 'n' palavras, gerar uma codificação para cada palavra presente, visando comprimir o texto recebido. Para isso, palavras que ocorrem mais vezes na mensagem precisar ter um código menor em relação a palavras que apareceram menos vezes. Também é necessário a implementação de estruturas de busca eficientes e os códigos devem ser gerados por meio de uma árvore de Huffman. Além disso, as mensagens somente contém palavras de até 63 caracteres e caracteres de 'a' até 'z'. Por fim, o programa recebe, após a entrada da mensagem, instruções que podem pedir a quantidade de vezes que uma palavra aparece na mensagem ou o código de uma palavra comprimida.

2. Implementação

O compilador utilizado foi G++ 5.4.0 e a pasta src contém os arquivos principais do programa: main.cc, hashing.h, hashing.cc, lista.h, lista.cc, huffman.h e huffman.cc. O funcionamento do algoritmo será explicado abaixo, assim como suas principais funções.

Hashing

Para a realização da tarefa, primeiro foi implementado um hashing que trata as colisões em listas encadeadas. A estrutura foi implementada por meio de um struct, em hashing.h/cc, e possui três atributos: o seu tamanho, sua chave para decodificação e ponteiros para listas encadeadas. Foi escolhido um hashing com 26 listas para o programa e a chave de decodificação é o inteiro 97, que equivale ao carácter 'a', pela tabela ASCII, assim as palavras são encaminhadas a cada posição da estrutura baseado na divisão de sua primeira letra por 97 ('a'). Assim, palavras começadas com 'a' ficam na lista encadeada na posição '0', palavras começadas em 'b' na lista encadeada na posição '1', e assim por diante até 'z' na posição '25'. Na main.cc do programa o ponteiro para hashing é alocado e em seguida cada lista encadeada desse hashing é inicializada vazia. Cada nódulo de uma lista encadeada contém dois ponteiros para vetor de caracteres que contém a palavra e código da palavra, um contador de quantas vezes a palavra apareceu e, naturalmente, um ponteiro para o próximo nódulo da lista.

Assim que o hashing é inicializado, na main, recebemos um número inteiro que representa o número de palavras que a mensagem terá é pega-se char por char do usuário

salvando-os num vetor de caracteres de tamanho 64 (limite especificado nas instruções) até que se encontre um espaço ou uma quebra de linha. Quando isso ocorre, coloca-se um '\0' na posição final da palavra, para que posamos tratá-la como string mais tarde. A chave de inserção no hashing é determinada pela primeira letra da palavra, como explicado no paragrafo anterior, e a função de inserção da lista encadeada é chamada, recebendo como parâmetro esse vetor de caracteres gerado. A função de inserção (insert), que pode ser encontrada em lista.h/cc, primeiro chama uma função auxiliar de pesquisa, que percorre a lista procurando a palavra e retorna um ponteiro para o nódulo onde a palavra se encontra, ou um ponteiro para cabeça da lista caso a palavra não exista ainda. Caso a palavra seja encontrada em algum nódulo da lista, o nódulo tem seu contador atualizado. Porém, caso a palavra não seja encontrada na lista, primeiro, determina-se o tamanho real da palavra e aloca-se dinamicamente memória para esse tamanho de palavra. Por exemplo, a palavra 'palavra\0', antes estava em um vetor de 64 caracteres e agora se encontrará em um ponteiro para um vetor de caracteres de tamanho '8'. E, por fim, aloca-se um novo nódulo da lista para a palavra, inicializando seu código como um ponteiro vazio e seu contador como 1, e adicionando o novo nódulo a primeira posição da lista. Esse processo se repete até que o número de palavras recebidas seja igual ao número de palavras que especificado pelo usuário.

Huffman – carregar dados

Assim que as entradas são recebidas, na main, a estrutura de dados Huffman é inicializada. Essa estrutura contém um ponteiro para um vetor de nódulos de árvore, o tamanho desse vetor e um nódulo de árvore raiz. Esse vetor de nódulos árvore é o que será usado para a criação da árvore de Huffman. Assim, cada elemento de cada lista encadeada presente do hashing é removido de sua lista e em seguida adaptado de um nódulo de lista para um nódulo de árvore, e inserido no vetor de nódulos presente na estrutura Huffman utilizando a função inserirElemento da estrutura Huffman. Os nódulos retirados são reinseridos em sua lista original logo em seguida, utilizando a função da lista: inserirNodulo, ela é diferente da função 'insert' vista antes e apenas insere um nódulo no final da lista.

A função inserirElemento do Huffman cita-da acima, insere o nódulo árvore em uma determinada posição do vetor de maneira a manter sempre o vetor ordenado primeiramente por contador, em segundo lugar pelo número de folhas e, por fim, por ordem lexical das palavras, esse algoritmo de inserção será utilizado novamente e é crucial para que a árvore de Huffman seja gerada corretamente. Para isso, essa função, aloca um novo ponteiro, com o tamanho atualizado, para um vetor de nódulos árvore e copia o vetor antigo até que encontre a posição do novo nódulo, quando encontrada a posição, o novo nódulo é inserido e o resto do vetor antigo e copiado no resto do vetor. Por fim, o vetor antigo é deletado e o novo é salvo na estrutura.

Huffman – gerar árvore

Agora que temos o vetor que será usado para gerar a árvore propriamente carregado, chamamos a função gerarÁrvore que é responsável por gerar a árvore de Huffman. Essa função segue exatamente os procedimentos descritos nas especificações desse trabalho.

Ela remove os dois primeiros nódulos do vetor ordenado de nódulos árvore presente na estrutura Huffman e gera um novo nódulo árvore pai cujo ponteiro filho da esquerda aponta para o primeiro nódulo e o ponteiro filho da direita aponta para o segundo nódulo. Esses nódulos representam os menores nódulos seguindo a ordenação descrita no paragrafo anterior. Além disso, o nódulo pai terá o contador e o número de folhas correspondente a soma desses componente dos filhos e terá como palavra a menor palavra presente nos filhos, que é determinada por uma função auxiliar. Assim, os dois primeiros nódulos do vetor gerador são removidos e o vetor gerador da estrutura é atualizado. Em seguida, o novo nódulo árvore que foi gerado é inserido usando a função inserirElemento, explicada no paragrafo de cima, mantendo sempre o vetor ordenado. Assim, esse procedimento se repete enquanto o vetor gerador tiver mais do que um nódulo. Quando o vetor gerador só tiver um elemento ele é salvo na raiz da estrutura e a nossa árvore de Huffman está gerada.

Huffman – gerar códigos

Agora que temos a árvore de Huffman gerada e o hashing funcional, precisamos percorrer a árvore para gerar os códigos de cada palavra. Para isso, foi utilizada a função recursiva gerarCodigos que recebe, inicialmente, a raiz da árvore de Huffman, a tabela hashing onde os códigos também ficarão salvos e um vetor de caracteres ‘code’, inicialmente de tamanho 1 e com ‘\0’ em sua primeira posição. A função recursiva tem como condição de parada um nó vazio ou o encontro de uma folha da árvore de Huffman. A cada chamada da função, caso o nódulo não seja uma folha, dois códigos novos são gerados, adicionando um ‘0’ ou ‘1’ no final do atual string code, a função é então chamada recursivamente duas vezes com o filho da direita e com o filho da esquerda como raizes, com os respectivos códigos atualizados e com a tabela Hashing. Caso ela encontre uma folha o código atual é salvo na árvore de Huffman e no devido elemento da lista encadeada presente do Hashing.

Por fim, temos os códigos gerados e salvos no Hashing, a parte final da main recebe um char e uma string e utiliza a função pesquisa já explicada para procurar uma palavra no hashing em sua respectiva lista encadeada e imprime ou seu código ou o número de vezes em que ela apareceu na mensagem, de acordo com o carácter digitado pelo usuário.

Ao fim do programa, os destrutores são chamados e toda a memoria alocada para execução é desalocada.

3. Instruções de compilação e execução

O trabalho pode ser compilado no linux utilizando-se o com o comando "make" enquanto dentro do domínio da pasta (paulo_fraga). O comando make gera o executável "tp3". Para rodar o executável basta clicar no .exec criado pelo comando make ou digitar "./tp3" no terminal enquanto no domínio que o executável se encontra.

4. Analise de complexidade

<u>Hashing</u>: A complexidade para encontrar qualquer lista encadeada do hashing é sempre O(1)

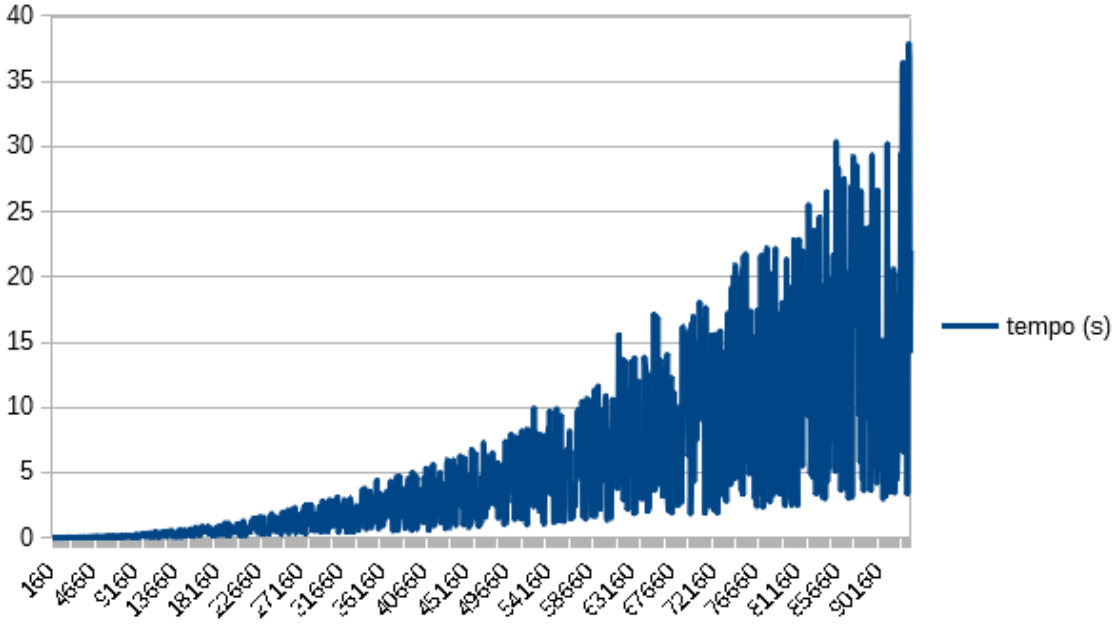
<u>Lista</u>:

- ***pesquisa***: a função tem complexidade (n * 63 ‘percorrer todas as palavras cuja última letra apenas é diferente’) no **pior caso**, ou seja, **O(n)** e **O(1)** no **melhor caso**
- ***insert***: a função chama pesquisa e tem que percorre o string recebido duas vezes, ou seja tem pior caso: O(n) ‘pesquisa’ + 2x64 ‘percorrer string’ + O(1) ‘inserir’, ou seja tem **pior caso O(n)**. e melhor caso O(1) ‘pesquisa’ + 2x1 ‘percorrer string’ + O(1) ‘inserir’, ou seja tem **melhor caso O(1)**
- ***pop***: tem complexidade **O(1)** em todos os casos
- ***inserirNodulo***: tem complexidade **O(n)** em todos os casos

<u>Huffman</u>:

- ***inserirElemento***: complexidade **O(n)** em todos os casos, pois sempre copia o vetor inteiro
- ***gerarArvore***: tem complexidade (n + n-1 + n-2 + … + 1)*O(n)‘inserirElemento’, pois sempre copia o vetor inteiro – 1 e chama a função inserirElemento, ou seja, tem complexidade **O(n^3)**
- ***gerarCodigos****:* A função recursiva tem complexidade que pode ser dada pela expressão: T(n) = 2T(n/2) + O(1), que representa o primeiro caso do Teorema Mestre e, por isso, tem complexidade **O(n)**, tanto no pior, quanto no melhor caso.

Programa como todo:



5. Conclusão

Com o trabalho que desenvolvi é possível se resolver o problema propostos nas especificações. As implementações são autorais e não foram consultados códigos de terceiros. Por fim, todos os cuidados foram tomados com a manipulação das memórias dinamicamente alocadas, que foram muitas e todas são desalocadas. Por fim, acredito que o ruido no tempo de execução do programa se de pela extensa comparação entre strings randômicos de tamanho 1 a 64.

6. Bibliografia

Não foram consultados materiais adicionais para a confecção desse trabalho, apenas o que foi apresentado em sala de aula.