O problema da compressão de mensagens de Rick Sanchez

Felipe Matheus Guimarães dos Santos[1]

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

felipe.guimaraes@dcc.ufmg.br

1. Introdução

A situação problema do presente trabalho é a compressão de mensagens de Rick Sanchez. Rick constantemente faz viagens interplanetárias e nessas viagens acaba fazendo amigos. Para manter contato com esses amigos, a principal forma é por mensagens, porém, pacotes de dados interplanetários são muito caros. Por esse motivo, Rick quer um algoritmo para compactar as mensagens e fazê-lo gastar menos dados.

2. Implementação

Para resolver esse problema, serão usados os seguintes algoritmos: para a contagem de palavras, a tabela Hash; e para a compressão, o Algoritmo de Huffman.

O programa principal roda em loop infinito, recebendo um inteiro de entrada, o número de palavras na frase. Após, faz-se a leitura das palavras e as armazena na tabela hash. O retorno do programa depende da ação do usuário: pode retornar a quantidade de vezes que uma determinada palavra aparece na frase ou retornar o código da palavra comprimida. O loop do programa principal é interrompido por EOF (Ctrl + C no CMD do Windows ou Terminal do Linux).

2.1. Nó

Tipo abstrato de dados usado para armazenar os atributos de cada palavra: a própria palavra,.uma string com tamanho máximo 64 caracteres, um contador para a quantidade de repetições da palavra na frase, um apontador para o próximo Nó da lista e um atributo que armazena a codificação da palavra.

2.2. Lista Encadeada

A implementação da Lista Encadeada foi a mesma usada no TP1, porém, com algumas modificações no tipo de nó e adição de métodos para o Hash. A lista se baseia em uma estrutura com apontadores para o Nó inicial e para o Nó final, a estrutura interna é resolvida com ponteiros.

O TAD Lista Encadeada possui os seguintes métodos:

- **is_vazia:** Método booleano que retorna verdadeiro se o apontador final e inicial apontarem para o mesmo Nó. Caso contrário, retorna falso.
- **pesquisa:** Recebe uma string palavra por parâmetro. Procura nos Nós se a palavra é igual a alguma salva. Se a palavra estiver presente na Lista, retorna o número de vezes que ela está presente, caso contrário, retorna zero.
- **incluir_elemento:** Recebe uma string palavra por parâmetro. Primeiro, verifica se a palavra já está contida na lista com o método **pesquisa**. Se estiver, aumenta o contador em uma unidade. Se não estiver presente, adiciona a palavra no final da estrutura.

2.3. Hash

A implementação do algoritmo Hash foi baseada nos slides da disciplina. O Hash é um TAD que armazena um ponteiro para uma Lista Encadeada, isto é, um vetor de Listas Encadeadas.

Possui os seguintes métodos:

- **indice:** Recebe uma string palavra como atributo. O índice do Hash implementado neste trabalho é bem simples: apenas separa as palavras pela primeira letra. Para tal, seleciona a primeira letra da palavra recebida e subtrai dela o valor 'a' ou 97 na tabela ASCII, isto é, 'a' – 'a' = 0 e 'a' – 97 = 0. Dessa forma, a Tabela Hash precisa ter apenas 26 posições, uma para cada letra. Retorna o índice em forma numérica.
- **inserir_hash:** Recebe uma string palavra. A partir dessa string, calcula-se o **índice** e chama a função **incluir_elemento** da estrutura Lista Encadeada, no índice correto. Ou seja, se a palavra começa com a 6ª palavra do alfabeto, o hash fará a inclusão ou o incremento na lista de índice 5, pois começa-se do zero.

2.4. Main

Primeiro, recebe-se uma variável inteira n, quantidade de palavras na frase. Com esse valor, lê-se as n palavras pela entrada padrão. Cada palavra lida é passada para tabela Hash, por meio do método **inserir_hash**.

Após receber as n palavras, roda em loop infinito recebendo as opções de saída: calcular a quantidade de repetições, q, ou a codificação da palavra, c, e uma palavra string.

Para a opção q, calcula-se o índice da palavra e faz-se a pesquisa pela palavra na lista da tabela hash com o índice correto. Printa na tela a quantidade de repetições da palavra especificada.

Para a opção c, não foi possível conseguir implementá-la a tempo.

Interrompa a execução com EOF, Ctrl+C no CMD ou Terminal.

2.5. Makefile

Para compilar e rodar o programa foi gerado um Makefile. Este arquivo facilita a compilação, pois elimina a necessidade de dar os comandos do g++ no terminal, apenas digitamos *make test* para rodar os testes ou *make run* para rodar o programa.

3. Instruções de compilação e execução

Para compilar e executar o programa, use o Makefile. Para tal, abra o Terminal e entre no diretório onde se encontra o programa, i.e., *.../felipe_guimaraes/src*. Dentro do diretório, dê o comando *make test* para rodar os testes ou *make run* para rodar o programa. Após a execução, dê o comando *make clean* para apagar os arquivos temporários. *Há* um arquivo *README.txt* com as especificações e mais detalhes.

Os compiladores utilizados foram: g++ (MinGW.org GCC-8.2.0-3) 8.2.0*,* no Windows 10, e g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0, no Ubuntu.

4. Análise de complexidade

- **ListaEncadeada::is_vazia:** Complexidade constante, $\theta(1)$.
- **ListaEncadeada::pesquisa:** Melhor caso: a palavra buscada é a primeira da lista, $\Omega(1)$. Pior caso: a frase inteira é composta por palavras diferentes iniciadas pelas mesmas letras e a palavra buscada foi a última a ser inserida, logo, a

complexidade é linear, O(n).

- **ListaEncadeada::incluir_elemento:** A inserção de Nós sempre acontece na última posição de uma Lista de índice qualquer, logo, complexidade constante, θ(1).
- **Hash::indice:** Cálculo simples, apenas subtração, complexidade constante, θ(1).
- **Hash::inserir_hash:** Custo da chamada do método **indice** mais a chamada do método **incluir_elemento**. Ambos têm complexidade constante, logo, inserir_hash também, θ(1).
- **Main:** Somando as complexidades do Hash e da Lista Encadeada, percebe-se que a função dominante é o pior caso do **ListaEncadeada::pesquisa**. Portanto, a complexidade no pior caso do algoritmo como um todo é linear, O(n) e constante no melhor caso, θ(1).

5. Conclusão

Portanto, para concluir o presente trabalho foi usado um algortimo de busca e uma estrutura de dados: Tabela Hash e Lista Encadeada. A utilização de cada um desses TADs foi pensada de acordo com as especificações do trabalho e com o auxílio do monitor Allison Svaigen.

A maior dificuldade do trabalho foi a escolha da função hash, pois existem muitas e, na maioria das vezes, são confusas. Por fim, foi escolhida uma função não muito eficiente, porém, completamente simples de entender.

Além disso, o tempo disponibilizado para fazer o algoritmo de Huffman foi muito curto.

Logo, o trabalho cumpre o que era esperado: utilizar todo o conhecimento que foi acumulado desde o ingresso na Universidade e o conhecimento de todos os algoritmos de busca, isto é, saber quando usar determinado algoritmo, respeitando as especificações.

O código e a documentação deste trabalho estão disponíveis no GitHub: https://github.com/FelipeGuimaraes42/TP3-ED

6. Bibliografia

TEODOROWITSCH, Roland. Adaptação para OpenOffice.org 1.1 feita em 29 mar. 2005. Disponível em: http://www.sbc.org.br/documentos-da-sbc/summary/169-templates-para-artigos-e-capitulos-de-livros/878-modelosparapublicaodeartigos. Acesso em 26 set. 2019.