Trabalho Prático 3 - Estruturas de Dados

Matheus Aquino Motta[1]

[1]Bacharelado em Matemática Computacional
DCC - Universidade Federal de Minas Gerais

matheusmotta@dcc.ufmg.br

Abstract. *In the previous work we solved a optimization problem to find the best schedule for the Rick Sanchez intergalactic trips, which lead our beloved scientist to make a lot of new friends around the universe. Now our new problem consists in support the Rick Sanchez gregarious behavior, by optimizing his text messages reducing its size in function of the word repetition over the text. Hence, more generally our problem is: given a string made of several words under potential repetitions, we want to compress the storage memory of these words in function of those that more appear in our entry. Then, our first goal is to storage this words and count the frequency of each one of them, to do so we use a simple hash function with chaining. After that our problem was how to get these different words with higher a lower frequencies and compress them, to approach this problem we used a greedy algorithm called Huffman Coding, where the binary string associated with each word is compressed in order to fit our min heap sort, where the ones that appear more frequently receive a smallest binary string and the ones that not appear quite much receives the longest ones.*

Resumo. *No trabalho anterior resolvemos um problema de otimização que consistia em encontrar a melhor ordem para a lista de viagens intergaláticas de Rick Sanchez, o que levou nosso amado cientista a fazer vários novos amigos ao redor do universo. Agora nosso novo problema consiste em colaborar com Rick Sanchez e sua facilidade em fazer novos amigos, otimizando suas menssagens de texto reduzindo o seu tamanho de armazenamento em função da potencial repetição de palavras ao longo do texto. Desse modo, de uma forma mais geral nosso problema consiste em: dada uma sequencia de palavras com potenciais repetições, queremos comprimir a memoria usada para armazenamento dessas palavras em função da frequencia de cada palavra ao longo da nossa entrada. Assim, nosso primeiro objetivo é contar a frequência de cada uma delas, para fazer isso foi utilizada um simples função hash com encadeamento. Após isso, o problema consistia em como usar essas diferentes palavras e frequências para comprimi-las, para isso utilizamos um algoritmo guloso chamado Código de Huffman, onde a sequência binária é associada a cada palavra é comprimida em função de uma ordenação utilizando um minheap, onde aquelas que apareciam mais frequentemente recebiam a menor sequência de bits e aquelas que tinham uma frequência arbitrariamente inferior recebiam as maiores.*

1. Introdução

O problema abordado nesse trabalho consiste basicamente em: dado um texto de entrada queremos encontrar a frequência de cada palavra de modo a comprimir a suas respectivas strings de identificação e minimizar o espaço de armazenamento. Desse modo,

primeiramente utilizamos um Hash com encadeamento para contar as palavras e após isso, utilzando as informações adquiridas a partir da contagem construimos uma Árvore Huffman, que atribuia a cada uma das palavras uma string respectiva sua frequência no texto.

2. Implementação

A implementação do trabalho consistiu basicamente no uso do Hash com encadeamento e de um Min Heap, para a contagem de palavras e construção da árvore Huffman respectivamente.

2.1. Hash e Lista Encadeada

O Hash foi implementado utilizando o encadeamento nas entradas, onde cada entrada do Hash possuia uma lista a ela associada. Para isso uma solução simples foi o uso de 26 posições, de modo a armazenar em cada entrada no hash uma lista com as palavras que iniciavam com um determinado caracter de 'a' até 'z'.

Assim, podiamos inserir, procurar e retornar informações a respeito dos elementos existentes no Hash, por meio da funções associadas a estrutura, que funcionaram basicamente como uma interface para cada uma das 26 listas.

As listas utilizadas para construção do Hash tinham métodos especificamente implementados para o armazenamento das palavras em função de sua frequência, isto é, as funções padrões de uma lista encadeada de construção, inserção, tamanho e busca foram alterados de modo que para inserção verificassemos caso aquela palavra já estivesse ali, assim incrementavamos o contador associado a cada palavra, como também na busca das palavras que ocorriam em função do nome atribuido ao objeto Word.

2.1.1. Min Heap e a Árvore de Huffman

A partir das listas criadas pelo Hash para a contagem dos elementos palavra, criamos uma array com as diferentes palavras e suas respectivas frequências, que foi utilizada para a construção de um minHeap que funciona basicamente como um Heap Sort, onde as operações respectivas ao Heapify e outras comparações utilizadas para definir a hierarquia dos elementos foi dada em função do método $\mathrm{wgt}(\mathrm{MinHeapNode}\ a, \mathrm{MinHeapNode}\ \mathrm{b})$ no qual o peso de um elemento é dado primeiro em função de sua frequência, caso seja igual, em função do seu número de folhas (primeiramente todos possuem uma folha, mas de acordo com a construção da árvore de HuffMan são construídas subárvores, e suas folhas são somadas.) e caso ambas possuam o mesmo número de folhas a hierarquia é dada pela ordem lexicográfica das palavras avaliadas. A partir disso, a função Heapify irá elevar as palavras de menor peso por meio da função $\mathrm{swap}(\mathrm{MinHeapNode}\ a, \mathrm{MinHeapNode}\ b)$.

Após, a construção do MinHeap inicial ordenado, iremos construir a árvore de Huffman por meio do seguinte algoritmo: 1) Criamos um Nodo central do heap e dois auxiliares Left e Right. 2) Extraimos o menor elemento do nosso Heap e atribuimos seu endereço a variável Left, assim como o segundo menor e atribuimos a variável Right. 3) Definimos as variáveis Left e Right como os filhos a esquerda e a direita, respectivamente, do Nodo Central. 4) Definimos o nome do nodo central como a palavra respectiva

ao seu filho a esquerda, e suas folhas e frequência são definidas como a soma das folhas e frequências da subárvores enrraizadas pelos filhos as esquerda e a direita. 5) inserimos o nodo Central no Heap. Para cada extração de um nodo do Heap decrementamos o seu tamanho em "1" e para cada inserção incrementamos em "1". Assim, repetimos esse algoritmo até que o tamanho da nossa árvore MinHeap originalmente definido pelo número de palavras distintas, seja igual a 1, isto é, todas as sub-árvores já foram concatenadas em função de sua hierárquia.

Assim, resta percorrer a árvore até encontrarmos um cada um dos nós folha e definirmos o seu novo código binário, dado pela distribuição de arestas da raiz até a respectiva folha, onde uma aresta para um filho da direita soma +"1" a string binária resultante e para cada filho da esquerda soma +"0" a string binária resultante.

3. Análise de Complexidade

Destarta para realizar a análise de complexidade, iremos definir a complexidade de cada uma das funções respectivas ao Hash, List e MinHeap, de modo a facilitar a interpretação da função Main. Então, seja n o número de palavras, considerando o pior caso, onde todas as palavras são distintas.

1. Complexidade dos métodos de List no pior caso:
   (a) List::empty(): $\mathcal{O}(1)$
   (b) List::size(): $\mathcal{O}(1)$
   (c) List::push_back(): $\mathcal{O}(n)$
   (d) List::find_copy(): $\mathcal{O}(1)$
   (e) List::find(): $\mathcal{O}(n)$
2. Complexidade dos métodos de Hash no pior caso:
   (a) Hash::insert(): $\mathcal{O}(n^2)$
   (b) Hash::size(): $\mathcal{O}(1)$
   (c) Hash::get_element() $\mathcal{O}(n)$
   (d) Hash::wordFrequency(): $\mathcal{O}(n)$
3. Complexidade dos métodos de MinHeap no pior caso:
   (a) MinHeap::MinHeap(): $\mathcal{O}(\mathrm{n})$
   (b) MinHeap::wgt(): $\mathcal{O}(1)$
   (c) MinHeap::swap_nodes(): $\mathcal{O}(1)$
   (d) MinHeap::heapify(): $\mathcal{O}(\log n)$
   (e) MinHeap::extract_min(): $\mathcal{O}(1)$
   (f) MinHeap::insert(): $\mathcal{O}(1)$
   (g) MinHeap::set_codes(): $\mathcal{O}(\mathrm{n})$
   (h) MinHeap::get_size(): $\mathcal{O}(1)$
   (i) MinHeap::create_code() $\mathcal{O}(\log n)$

Todos os métodos de MinHeapNode e de Word possuem custo $\mathcal{O}(1)$ ou tiveram seu custo considerado na função em que são utilizadas.

Assim, primeiramente sabendo que temos um custo para criar o Hash igual a $\mathcal{O}(\mathrm{n}^2)$ no pior caso, assim como, para criar o vetor que será usado para construção da array base do nosso MinHeap. Em segundo lugar, para a construção da nossa Árvore de HuffMan, temos que a função extract_min() é chamada $2(n-1)$ vezes, que por sua vez possui o custo do método Heapify, logo possui um custo de $\mathcal{O}(\mathrm{nlogn})$. Além disso, o método

set_codes() possui um custo de $\mathcal{O}(\mathrm{n})$, que por sua vez chama o método create_code de custo $\mathcal{O}(\log n)$, resultando em um custo $\mathcal{O}(\mathrm{nlogn})$. Portanto:

$$\begin{aligned} f(n) &= \mathcal{O}(\mathrm{n}^2) + \mathcal{O}(\mathrm{nlogn}) + \mathcal{O}(\mathrm{nlogn}) + \\ &= \max(\mathcal{O}(\mathrm{n}^2),\ \mathcal{O}(\mathrm{nlogn})) \\ &= \mathcal{O}(\mathrm{n}^2) \end{aligned}$$

Portanto a complexidade assintótica do nosso algoritmo para o pior caso é da ordem de $\mathcal{O}(\mathrm{n}^2)$.

O que podemos visualizar a partir da análise experimental realizada a partir de diferentes amostras.

O primeiro gráfico mostra o tempo gasto pelo algoritmo em função da entrada de acordo com o número de palavras distintas, considerando uma probabilidade de repetição alta para 50 amostras. No entanto, minimizando a probabilidade de repetições de palavras podemos ver uma queda drástica no tempo de execução do algoritmo, que se aproxima de uma complexidade linear logarítmica.

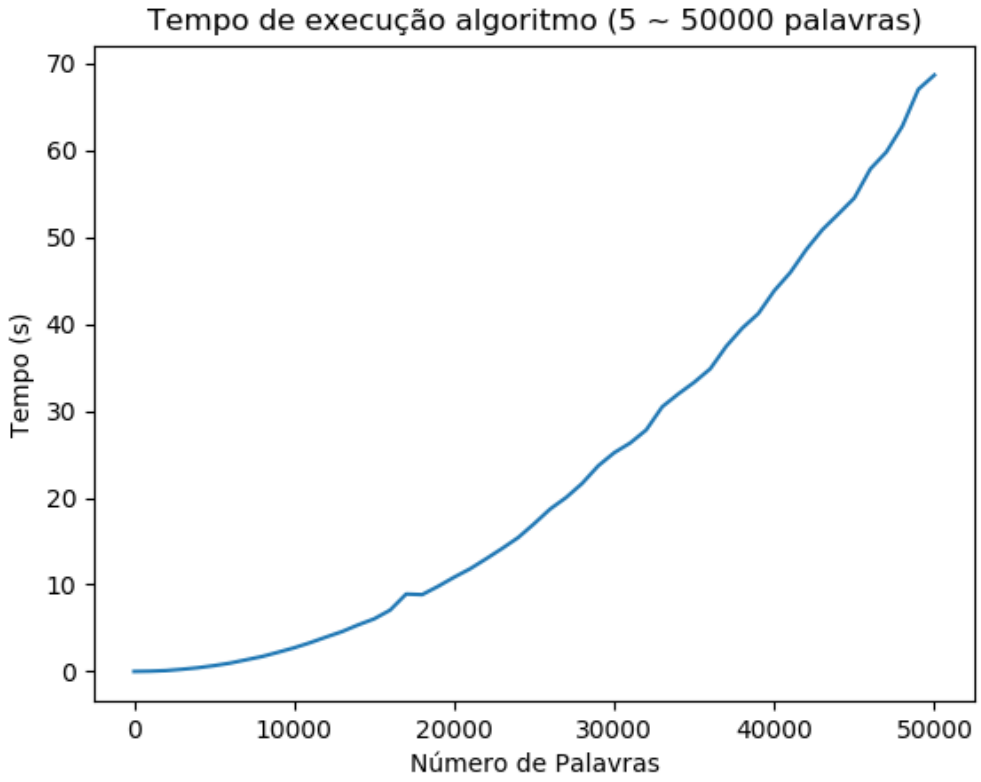

Figure 1. 50 amostras alta probabilidade de repetição.

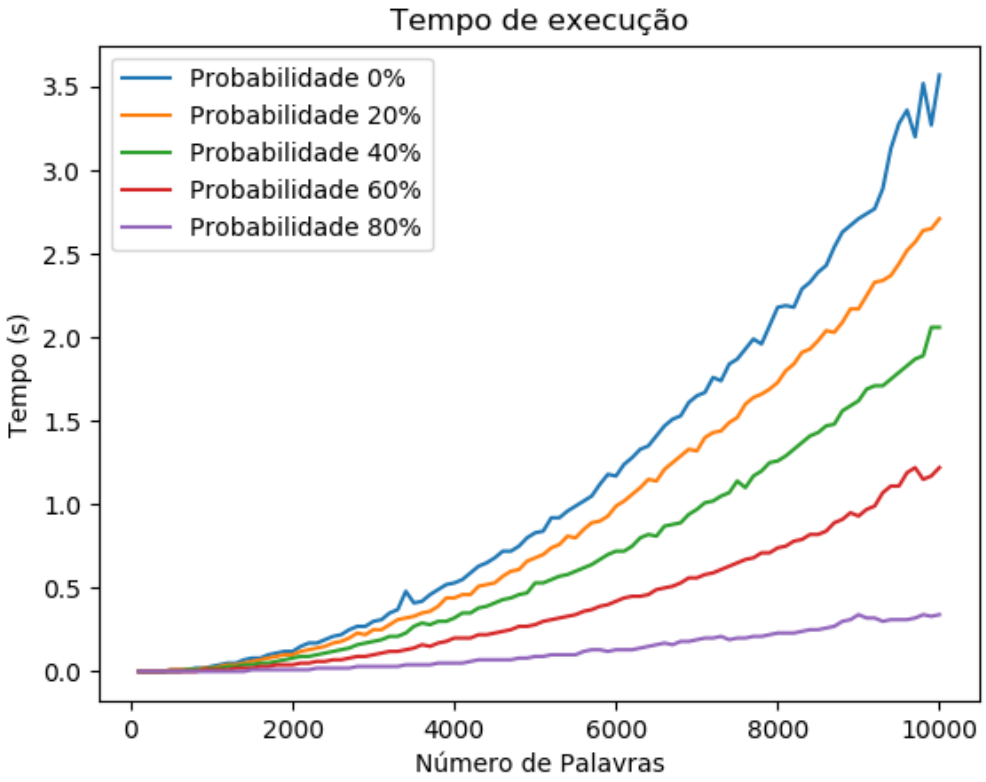

Figure 2. 100 amostras para diferentes probabilidades de repetição.

Então, podemos concluir que análise experimental possui resultados assimo como os esperados pela análise de complexidade teórica.

Os testes foram realizados em um Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz 8GB RAM.

4. Instruções de compilação e execução

Para a execução do programa basta a utilização do *Makefile* presente no arquivo e a execução do comando *make* no diretório, para limpar o diretório basta a execução do comando *make clean*.
O algoritmo foi implementado e testado em um ambiente linux Ubuntu

Distributor ID: Ubuntu
Description: Ubuntu 18.04.2 LTS
Release: 18.04
Codename: bionic

Utilizando o compilador *g++ -std=c++11*.

5. Conclusões

O trabalho possibilitou o entendimento e utilização de algumas Estruturas de Dados ensinadas na disciplina assim como um uso para a solução de um problema factível e de potencial real aplicação.

Além disso, podemos perceber que caso houvessemos utilizado um Hash mais eficiente nossa complexidade assintótica iria decair drásticamente, haja vista que a função principal do nosso algoritmo possui uma complexidade linear logarítmica $\mathcal{O}(n\log n)$.

.

See you later, Rick Sanchez!

References

https://www.geeksforgeeks.org/