Trabalho Prático 0: Trie

Francis Carlos dos Santos

Matrícula: 2012022167

1 Introdução

Pesquisa e recuperação de palavras é uma tarefa normalmente necesssária em Ciência da Computação, principalmente nos ultimos anos com o aumento do numero de dados que podem ser acessadoos por meio da internet. Para um melhor proveito de toda a informação disponivel na internet varias estruturas de dados foram criadas afim de facilitar essa tarefa.

Nesse trabalho o problema a ser resolvido é, dado um dicionário, descobrir com qual frequência aparecem as palavras presentes no dicionario. Será utilizada a estrutura Trie, nome que ajuda bastante a entender seu proposito pois vem do inglês *reTRIEval*, que significa recuperação. A ideia principal dessa estrutura e evitar pesquisar a palavra inteira, ou seja implementando uma estrutura onde as palavras podem ser alcançadas de acordo com seu prefixo, o que faz desse tipo de estrtura potencialmente mais eficiente que estruturas como o *Hash* e árvores binarias [1].

2 Solução do Problema

A figura (1) mostra a estrutura de dados utilizada para armazenar as palavras.

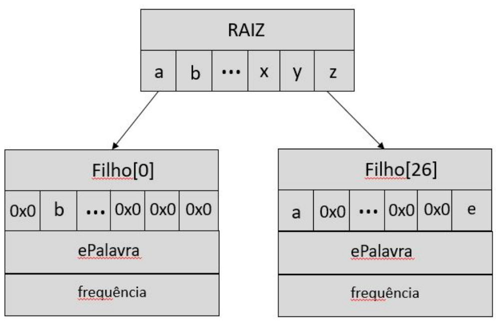

Figura 1: Estrutura de dados utilizada para armazenar o dicionário.

A figura acima ilustra como funciona a estrutura de dados proposta para resolver o problema. Cada célula receberá uma única letra. A estrutura contém um vetor apontador *filho* para uma estrutura do próprio tipo, este vetor contém vinte e sete posições. Uma variável booleana *ePalavra* que é verdadeira quando a letra inserida nesse nó representa

o fim de uma palara do dicíonario e um inteiro *freq* que é incrementada apenas quando $ePalavra$ = ***true*** para contar quantas vezes a palavra aparece no texto.
Na prática cada letra do alfabeto é representada por um index, obtido através da equação mostrada abaixo.
Para auxiliar na solução do problema, foi criada também uma estrutura **Dict**, que basicamente tem a função de armazenar as palavras do dicionário e no final receber da função **encontraFreq** a frequência com que cada palavra aparece no texto. O algoritmo abaixo mostra a implementação das estruturas utilizadas.

Algoritmo 1: Estruturas utilizadas

```
typedef struct trie
  bool ePalavra
  int freq
  struct trie *filho[TAM_ALFABETO]
TrieNo;

typedef struct
  char string[MAX_TAM]
  short int freq
Dict;
```

$$\mathit{index} = \mathit{letra} - \text{"a"} \tag{1}$$

Equação para obtencão do index de cada letra.

Como a letra 'a' é representada pelo numero 97 na tabela ASCII, com essa equação é possível representar todas as letras do alfabeto como numeros de 0-27, logo cada indice do vetor apontador *filho* se torna uma letra quando inicializado, isso elimina a necessidade de utilizar comparação entre caracteres.
Após ler o dicionaririo do arquivo de entrada, todas as palavras são inseridas na árvore. No inicio todas as posições do vetor apontador *filho* são inicializadas com **NULL**, caso o prefixo não exista na árvore um novo nó é criado, e o ponteiro passa a apontar para o próximo filho caso contrario é feita a verificação se a letra representa uma folha e aponta para o proximo nó.

A busca é realizada sequencialmente, para cada palavra lida no texto é verificado se o caractere existe na árvore, caso exista é verificado se a letra em questão representa uma folha e o ponteiro é incrementado, caso contrario a busca termina. Os algoritmos 2 e 3 representam as duas operções descritas anteriormente.

Algoritmo 2: inserePalavra(*TrieRaiz,palavra)

```
// Insere letra por letra a palavra na arvore
// Caso o prefixo ja exista, caminha na arvore até encontrar a posição final
// toda letra final de palavra é marcada por ePalavra =true
    ptr* = TrieRaiz
for i = 0 to k do
    index = palavra[i] - 'a'
   if ptr->filho[index] == NULL then // cria e inicializa celula
   ptr->filho[index] = criaNo();
   ptr = ptr->filho[index];
   ptr->freq = 0;
    if i == tamanhoPalavra-1 then Verifica se a letra e uma folha
           ptr->ePalavra = true
   else//caso ja exista a letra
      if i == tamanhoPalavra-1 then
         ptr->ePalavra = true
         ptr = ptr->filho[index];
```

Por ultimo é necessário pesquisar a frequência com que cada palavra do dicionário apareceu no texto e depois imprimir. A função que faz essa busca é bem similar a anteriormente apresentada, basicamente a unica diferença é que a ultima utiliza o próprio dicionário para fazer a busca. A impressão é feita percorrendo a estrtura dicionario e imprimindo a frequêcia de cada palavra da estrutura, como pode ser visto no algoritmo 4.

Algoritmo 3: BuscaPalavra(*TrieRaiz,palavra)

```
//Pesquisa palavra do texto na árvore
//Caso exista a palavra incrementa a frequencia
//Caso a palavra nao exista na arvore a busca é
//interrompida após o primeiro erro.
for i = 0 to k do
    index = palavra[i] - 'a'
if ptr->filho[index] == NULL then // se a letra nao existe sai do loop
      break
      else
      if ptr->filho[index]->ePalavra == true AND i == (tamanho-1) //Verifica se letra é folha
        ptr->filho[index]->freq = ptr->filho[index]->freq + 1
        ptr = ptr->filho[index]
     else
         ptr = ptr->filho[index];//Caso não seja sufixo, aponta para o proximo
```

Algoritmo 4: imprimeFreq(*dicionario,n)

```
// Imprime todas as frequencias
for i = 0 to m do
    print dicionario[i].freq
```

3 Análise de Complexidade

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço para as principais funções do algoritmo.

3.1 Análise Teórica do Custo Assintótico de Tempo

Sejam m e n e $k = 16$ o número de palavras no dicionário, no texto e o numero máximo de caractéres em cada palavra respectivamente. Para esse problema o pior caso ocorre quando todas as palavras do dicionario e do texto não possuem os prefixos iguais e as palavras tem o número máximo de caracteres permitido, o que nos da o caso em que teriamos que percorrer toda palavra para buscar ou inserir. Porém com a implementação por indexação temos o acesso ao ramo onde se encontra o prefixo da palavra com $O(1)$ para busca ou inserção, logo o custo maximo para localizar ou inserir uma palavra seria constante. Podemos analisar o custo assintótico do programa avaliando cada um dos algortmos apresentados juntamente com entrada e função de *freeTrie*.

- O custo assintótico do algoritmo 2 é O(m), para cada palavra inserida o algoritmo faz no k iterações , portanto a inserção de cada palavra tem custo constante, para m palavras inseridas temos a complexidade assintótica O(k*m) logo, O(m).
- O algoritmo 3 para cada palavra no texto temos uma chamada dessa função para o numero maximo de iterações igual ao tamanho da palavra. Portanto assintóticamente a função seria O(k*n), como K tem valor máximo igual a dezesseis, essa função tem complexidade assintótica igual a O(n)
- O algoritmo 4 é O(m), sendo que sempre será impresso m frequências na saída.
- A função leitura tem é feita em duas partes, primeiramente se lê o dicionario e em seguida o texto. A complexidade nesse caso será O(n), pois o texto sempre será maior que o dicionário.
- Para desaslocar a Trie a função *freeTrie* percorre todos os nós da raiz e para cada nó diferente de *null* a função percorre também esse nó, em chamadas recursivas na qual a entrada é o filho do nó anterior. Nesse caso a complexidade será dada por Om*TAM_ALFABETO pois a função percorre todos as posiçoes de todos os nós com filhos. Como TAM_ALFABETO é uma constante a complexidade assintótica dessa função é O(m).

Conclui-se assim que a complexidade assintótica de tempo do algoritmo é $max(O(\mathrm{m}),O(\mathrm{n}))$.

3.2 Análise Teórica do Custo Assintótico de Espaço

Temos duas estruturas de dados principais *TrieNo* e *Dict* com custos de 116 e 18 bytes respectivamente. As duas estruturas são dependentes do numero de palavras do dicionário *m*, logo, a complexidade de espaço de cada uma será O(m*116) e O(m*18), a complexidade assintótica de espaço do algoritmo será dada então por, O(116*m) ou$O$(m).

4 Análise Experimental

A análise experimental da implementação é mostrada pelas figuras (2) e (3). Para realizar os experimentos foi utilizado um gerador de casos teste além dos casos teste disponibilizados via moodle. O tempo de execução foi obtido pelo terminal ao fim da execução do progama. O espaço utilizado em cada execução foi obtido por meio da utilização do Valgrind. Os testes foram realizados em uma máquina com processdor Core i7-4720HQ 2.6GHz e 8GB de memória ram.

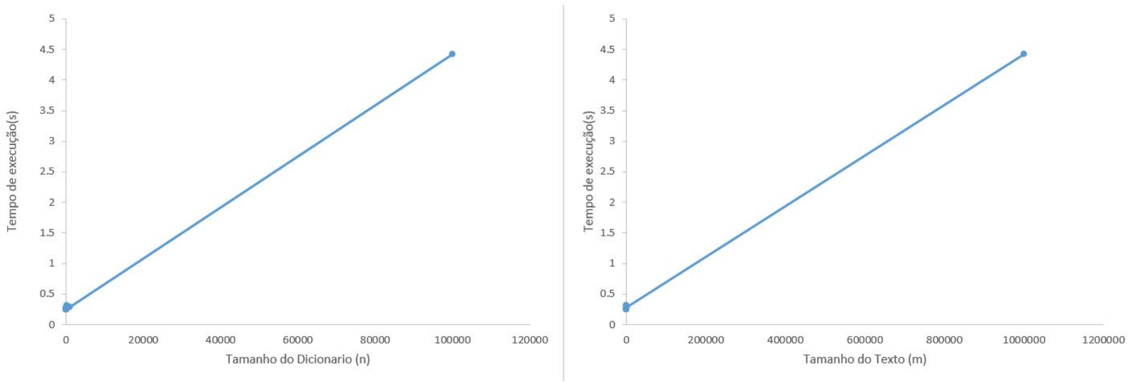

Figura 2: Grafico mostrando como varia o tempo de execução de acordo com o tamanho da entrada

A figura (2) fornece uma visão do tempo de execução do algoritmo para variações no tamanho do dicionario m e do texto n. Como esperado os grafico se comportam de forma linear conforme previsto na análise de complexidade assintótica de tempo. Ainda é possível observar que o tempo de execução para entradas grandes, realmente responde à maior entrada, m ou n.

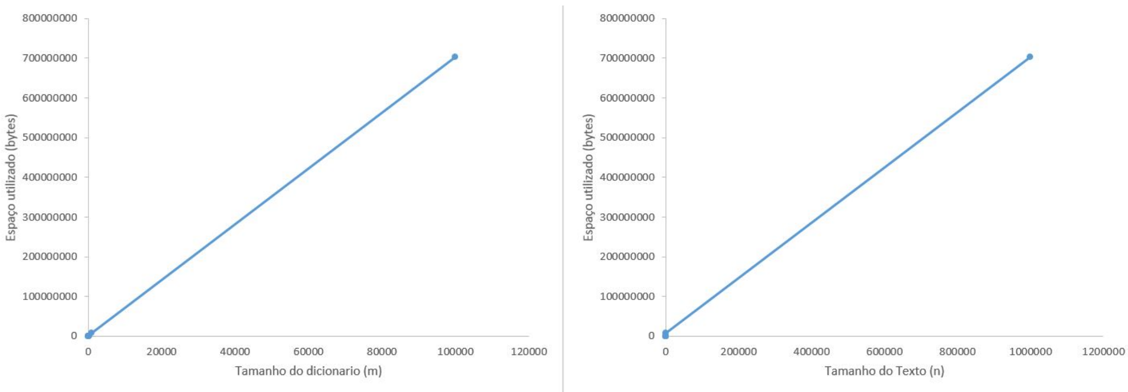

Figura 3: Estrutura de dados utilizada para armazenar o dicionário.

A figura (3) mostra a quantidade de memória, bytes, utilizada pelo programa a cada execução. É possível preceber que como previsto na análise assintótica do espaço utilizado pelo programa, os graficos só respondem á variações no tamanho do dicionário m. O que faz muito sentido já que só as palavras do dicionario são armazenadas.

5 Conclusões

Nesse trabalho, foi implementada a estrutura Trie,uma estrutura de dados mais eficiente que a tabela hash e arvores binarias, por não precisar percorrer toda a palavra. O problema a ser solucionado foi, dado um dicionário e um texto, calcular quantas vezes as palavras do dicionario aparem no texto sendo que a frequência deveria estar presente no nó da árvore. O problema foi solucionado de forma a minimizar o tempo de processamento e o uso de memória. A análise de complexidade teórica de tempo e espaço foi comprovada. Através da análise experimental podemos comprovar que de fato o algoritmo desenvolvido tem complexidade de tempo e espaço $max(O(\text{m}), O(\text{n}))$ e $O(\text{m})$ respectivamente.

Referências Bibliográficas

[1] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.