ALGORITMOS E ESTRUTURAS DE DADOS III

Mírian Francielle da Silva - 2016/2

Trabalho Prático 0

Introdução

Existem inúmeros algoritmos e estruturas de dados, que podem ser aplicadas em diversas situações para resolver problemas computacionais. Uma dessas estruturas, é conhecida como **Trie**, ou Árvore Trie. Que consiste em indexar e buscar palavras (*strings)* em um determinado texto. Assim como em um dicionário manual, a busca de uma única palavra, é realizada a partir dos prefixos que à compõe. Um exemplo de aplicação da estrutura Trie, é a busca realizada no *web browser*. Assim que o usuário começa a inserir sua pesquisa, o navegador já se ao completa rapidamente, lhe dando diversas opções à partir do prefixo digitado.

A implementação a ser apresentada a seguir, é um algoritmo que utiliza a **Trie** para realizar buscas de palavras de um dicionário e verificar se estão presentes em um determinado texto, e retornar a frequência com que ela é encontrada. A ideia principal da estrutura consiste em armazenar cada caractere da palavra em um nó da árvore, e assim por diante, até que toda a palavra esteja indexada. A raiz da árvore, é uma palavra vazia, e seus nós filhos são as letras das respectivas palavras do texto.

O algoritmo Trie usará o largo texto inserido para checar cada *string* que faz um casamento exato com a correspondente no dicionário. Sendo assim, retornando quais palavras do dicionário estão contidas no texto. É possível que nem todas as *strings* do dicionário estejam no texto, e vice e versa. É possível que contenha somente prefixos das *strings* do texto no dicionário, mas que não correspondam a busca realizada, resultando em um não casamento da palavra completa no dicionário.

Modelagem e Solução Proposta

A estrutura **Trie** implementada, se resume em:

struct TrieNode
    boolean end;
    integer count;
    TrieNode *child[ALPHABET_SIZE];

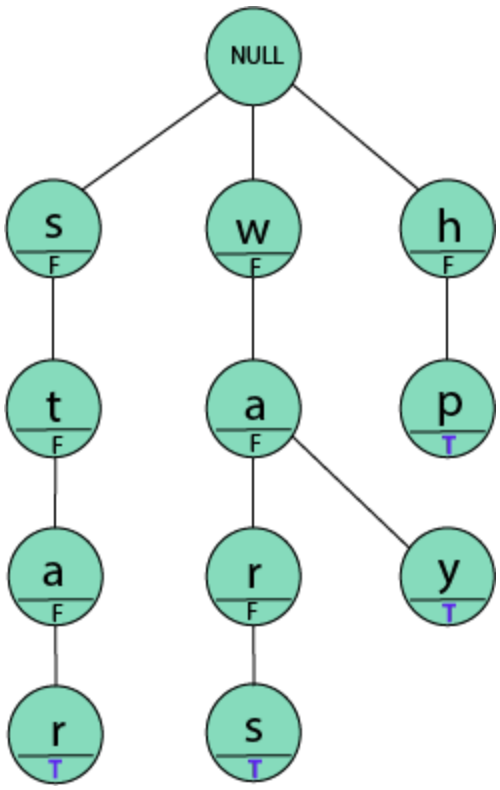
- Uma variável booleana que indica se a palavra foi encontrada ou não no texto. Ou seja, se todos os nós filho, são prefixos da busca e se chegaram ao fim da palavra buscada. Retorna se o final da palavra existe ou não no nó.

- Um contador - inteiro - de ocorrências da palavra em todo o texto. O contador será incrementado a cada vez que a variável booleana marcar que a palavra chegou até o fim.

- Um ponteiro para os nós filhos. Para que seja feito o acesso em O(1), o ponteiro é um tipo *char* do tamanho do alfabeto latino, do padrão ISO, que possui 26 letras.

No algoritmo principal, as palavras que compõe o dicionário e o texto, são lida e armazenadas em um vetor de *char,* em que cada posição do vetor, indica uma *string* do tamanho máximo definido, para depois serem chamadas nas funções de adição e busca na **trie**.

Representação de uma Trie com palavras adicionadas, e seus respectivos status, sendo eles *true* quando a palavra chega ao final, e *false* quando a palavra não chegou ao final:



As seguintes funções foram utilizadas para a manipulação da árvore **Trie :**

initializer: Função que cria um nó e inicializa o ponteiro para a raiz da árvore, e seus respectivos dados são alocados dinamicamente e setados em zero.

search_word: Função que realiza a busca de uma determinada *string* na árvore texto, e retorna o contador de ocorrências dela, mesmo que ela esteja presente no texto ou não.

add_word: Função que adiciona uma palavra na estrutura da árvore. A função consiste em verificar tais

parâmetros: Se a palavra a ser inserida não existe; se ela já foi inserida e já existe na árvore; ou se a palavra a ser inserida não existe, mas é sufixo de uma palavra já indexada.

free_trie: Função que libera a memória alocada dinamicamente para a criação da Trie.

Análise do Algoritmo

A **Complexidade** de busca e inserção na Trie é linear, no entanto ela é realizada em diversos níveis da árvore, sendo o nível, correspondente ao tamanho da palavra inserida.

A ***Complexidade Temporal*** do algoritmo implementado, é **O(n)**. Onde n é o tamanho da palavra a ser inserida. E a ***Complexidade Espacial***, é dada pelo espaço ocupado em memória ao executar o algoritmo, sendo ela o número de caracteres da palavra (*length n*), multiplicado por uma constante **c**, que determina as variações das letras do alfabeto aceito - 26 letras -, resultando em **O(n*c)**.

Tipos de Entrada e Saída:

O algoritmo ao ser executado, lê as seguintes informações:

- 1º linha: valor inteiro **n**, que determina a quantidade de palavras que irão compor o dicionário.
- 2º linha: **n** palavras dos dicionário;
- 3º linha: valor inteiro **m**, que determina a quantidade de palavras que irão compor o texto.
- 4º linha: **m** palavras do texto.

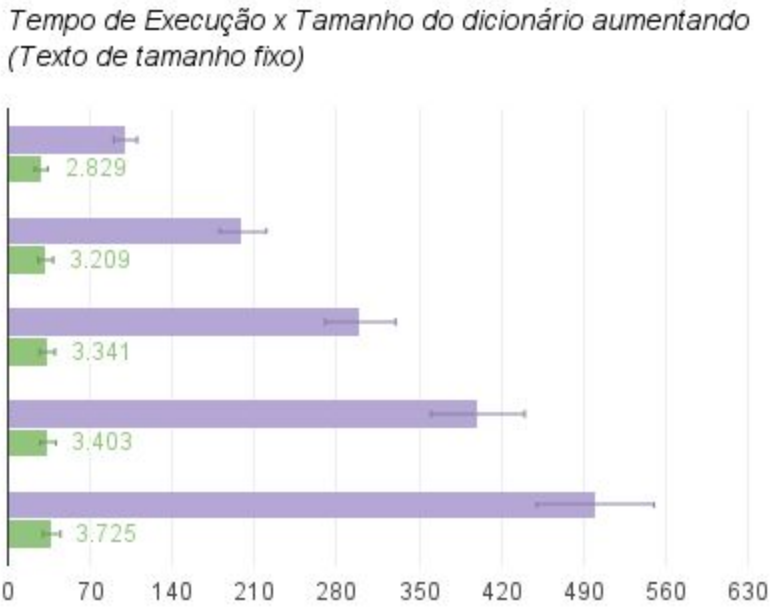
Todas os caracteres que irão ser lidos, são letras minúsculas, cada palavra será composta pelo tamanho máximo de 15 caracteres.

A ***saída*** é composta por inteiros, que correspondem aos valores de frequência de cada palavra do dicionário encontrada no texto. Podendo ela ser 0, se a busca não for encontrada no texto.

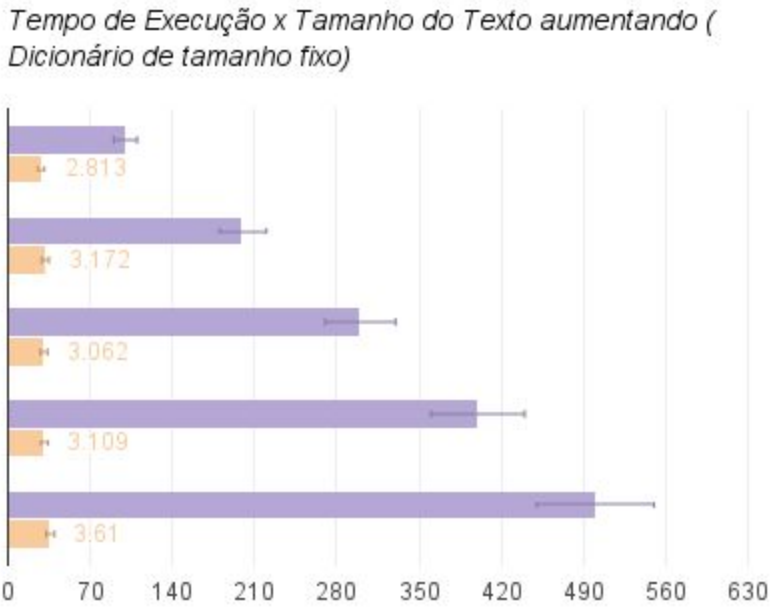
Análise e Experimentos:

A análise gráfica feita a seguir, apresenta o resultado dos experimentos, de acordo o número de palavras, tanto no dicionário, quanto no texto, aumentam. No primeiro gráfico **(1)**, o tamanho do texto está fixo, e as palavras do dicionário aumentam gradativamente. No segundo gráfico **(2)** a quantidade de palavras do dicionário é mantida fixa, e o texto aumenta gradativamente. E na terceira análise **(3)**, tanto o texto, quanto o dicionário aumentam de tamanho. As entradas foram geradas aleatoriamente. A análise, é coerente com a complexidade temporal da implementação.

A média do tempo de execução foi gerado através da função *time* [-p] via terminal linux. No qual é fornecido um sumário de tempo real consumido, de CPU e do sistema (em milissegundos). Nos gráficos o tempo está em micro segundos.

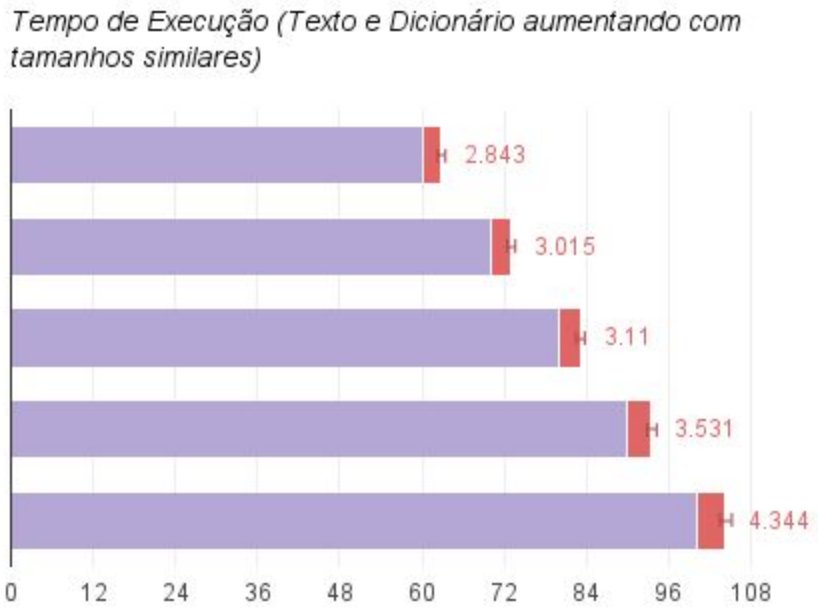


(1) Em cor verde, a média de tempo para diferentes casos testes, gerados aleatoriamente, sendo as palavras do dicionário pertencentes ou não ao texto nesses casos. E o tamanho do texto é fixo, alterando somente o tamanho do dicionário (em roxo).



(2) Em cor laranja, a média de tempo para diferentes casos testes, gerados aleatoriamente, sendo as palavras do dicionário pertencentes ou não ao texto nesses casos. E o tamanho do dicionário é fixo, alterando somente o tamanho do texto (em roxo).

(3) Em cor rosa, a média do tempo de execução, para diferentes casos testes. No entanto, o tamanho, tanto do texto, quanto do dicionário permanecem o mesmo a cada teste, e aumentam ao mesmo tempo. Ex: Tamanho 50 do dicionário = Tamanho 50 do texto; e assim por diante. É aleatorio a quantidade de palavras pertencentes ou não ao texto.



Referências:
[1] https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/
[2] https://www.youtube.com/watch?v=NKr6gWcXkIM

Anexos: [1]makefile, [2] main.c, [3]trie.h, [4] trie.c