Trabalho Prático 0: Frequência de Palavras

Algoritmos e Estruturas de Dados III – 2016/2

Bruno Varella Peixoto

1 Introdução

Com o objetivo de trabalhar estruturas de dados vistas em AEDS2 para relembrar alguns conceitos, este trabalho consiste basicamente na implementação de uma árvore trie. Devido a boa eficiência desta estrutura em pesquisar e inserir entradas, a trie é amplamente utilizada em programas de predição de texto, autocomplete, checagem de ortografia, entre outros.

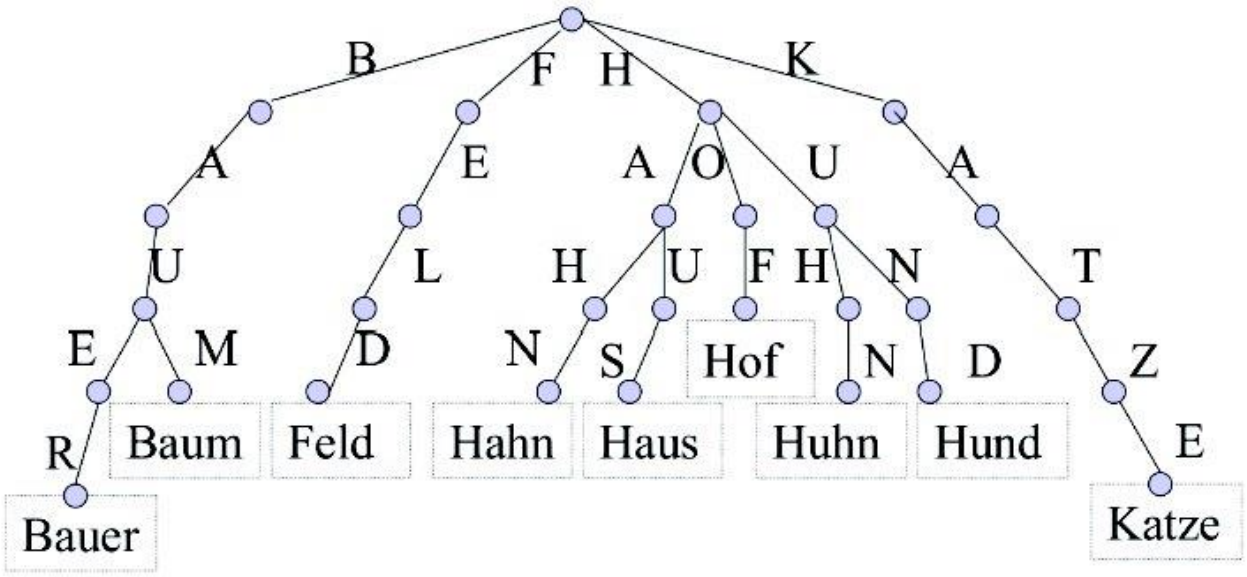
Neste trabalho, especificamente, são fornecidas palavras que compõe um dicionário, implementado por uma trie. Em seguida é passado um texto e deve-se retornar o número de vezes que cada palavra do dicionário aparece neste texto. Um detalhe especial é que a frequência de cada palavra deve ser armazenada dentro da estrutura da trie.

2 Solução do Problema

Inicialmente, todas as palavras do dicionário são lidas e armazenadas em uma lista, em que cada nodo guarda uma palavra. A lista é simplesmente encadeada (só tem ponteiro para uma direção). A inserção é sempre no fim, de modo a preservar a ordem das palavras para facilitar a impressão da frequência de cada uma no final. Com essa lista completa, é feita uma passagem nodo a nodo, armazenando cada palavra em uma árvore trie. Essa árvore tem algumas propriedades: cada vértice representa uma letra do alfabeto e possui uma flag para saber se esta letra representa ou não o fim de uma palavra. Outra informação que também é armazenada é o número de vezes que essa palavra existe no texto. Cada vértice também tem um vetor de 26 ponteiros para outros nodos, um para cada letra do alfabeto. Inicialmente, todos os ponteiros são inicializados como NULL e só passam a apontar de fato para uma nova letra caso seja necessário (caso exista uma palavra no dicionário com tal sequência de letras).

```
struct  trie_node{
struct trie_node* next_char[ALPHABET];
int is_word;
int num_visited;
```

Representação de um vértice da trie.



Exemplo de uma árvore trie com as palavras: Bauer, Baum, Feld, Hahn, Haus, Hof, Huhn, Hund, Katze

Há exemplos acima da implementação de um vértice em C e uma figura de exemplo de uma árvore trie completa para melhor entendimento.

Em seguida, todo o texto é lido, palavra a palavra e é feita a pesquisa na trie. A cada match, o contador num_visited é incrementado. Agora basta fazer outra pesquisa na trie, utilizando a lista do dicionário para imprimir a frequência de cada palavra, na ordem da entrada.

3 Análise de Complexidade de Tempo

As principais funções do trabalho são:

- (1) read_dictionary_words_from_stdin();
- (2) insert_dictionary_words_to_trie();
- (3) write_word_occurrencies_to_dictionary();
- (4) print_words_frequency();

A função (1) lê palavra por palavra do stdin e salva no fim de uma lista encadeada. A inserção na lista é O(m) em que “m” é a quantidade de palavras na lista (número de nodos). A leitura do stdin se repete “n” vezes, em que “n” é o número de palavras. Esta função como um todo é, então, da ordem de O(m*n).

Já a função (2) percorre toda a lista encadeada e salva cada palavra em uma árvore trie. Percorrer a lista é O(m), sendo “m” a quantidade de palavras nela. A inserção na trie tem um custo muito baixo, da ordem de O(k), tendo “k” como o tamanho da palavra inserida. Como

no trabalho o máximo de caracteres de uma palavra é 15, podemos tomar “k” como uma constante. Assim, essa função é da ordem de O(m).

A função (3) lê o texto da entrada, de palavra a palavra e faz uma pesquisa na trie, contando o número de ocorrências de cada uma. A leitura do texto é de complexidade linear, da ordem de O(m), tendo “m” como o número de palavras do texto. A pesquisa na trie tem a mesma complexidade da inserção, é O(k), sendo “k” a quantidade de caracteres da palavra a ser inserida. Também considerando k uma constante, temos essa função da ordem de O(m).

Essa última função (4) é a responsável por imprimir a saída. Ela recebe a lista do dicionário e a árvore trie. Cada palavra é pesquisada e é retornada sua frequência no texto. Considerando a complexidade de pesquisa na trie como constante novamente, essa função tem complexidade O(m) em que m é o número de palavras da lista.

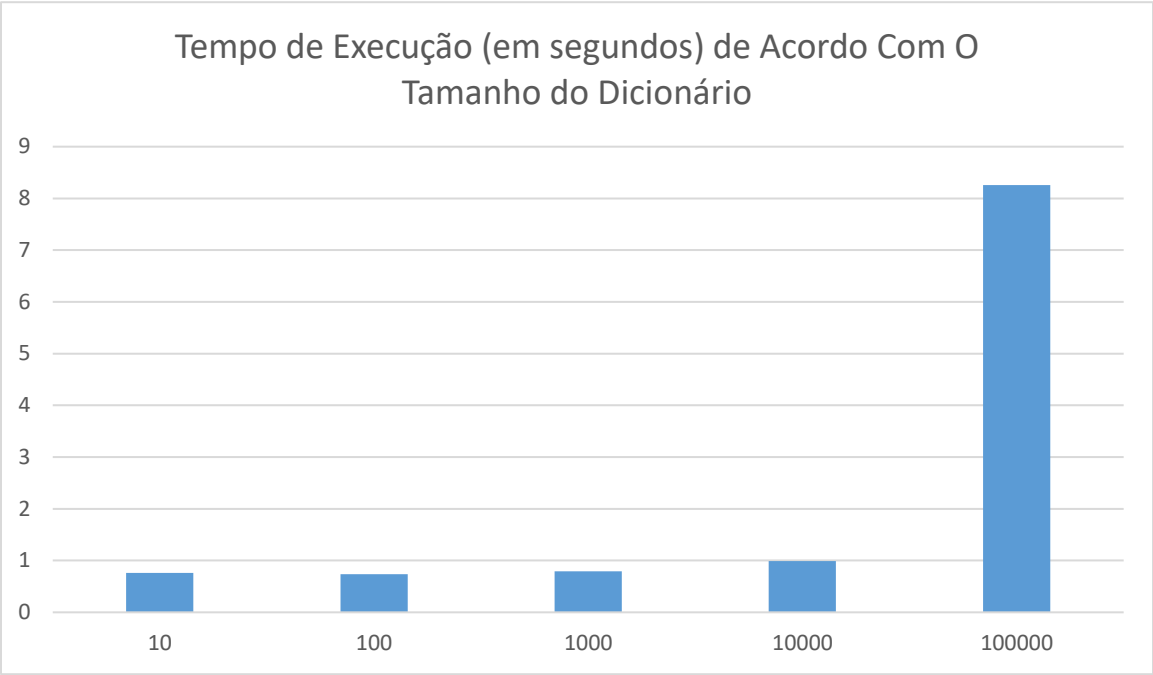
4 Análise de Complexidade de Espaço

Para este trabalho, a análise de complexidade de espaço deve ser observada nas duas estruturas que são criadas. A primeira estrutura é a lista encadeada que armazena as palavras do dicionário na ordem da entrada. Em questão de espaço, essa estrutura tem complexidade linear na ordem de O(n), em que “n” é o número de palavras passadas na entrada.

A segunda estrutura é a árvore trie. Essa já tem um gasto maior de memória. Sua complexidade espacial é da ordem de O(k*a) para cada palavra na trie. Consideramos “k” como a quantidade de caracteres de uma palavra e “a” como o tamanho do alfabeto. Esse grande consumo de espaço se deve ao fato de que cada letra de uma palavra aponta para outras 26 letras (tamanho do alfabeto). Adicionando “n” na ordem de complexidade como o número de palavras na trie, temos que essa estrutura é da ordem espacial de O(k*a*n).

5 Avaliação Experimental

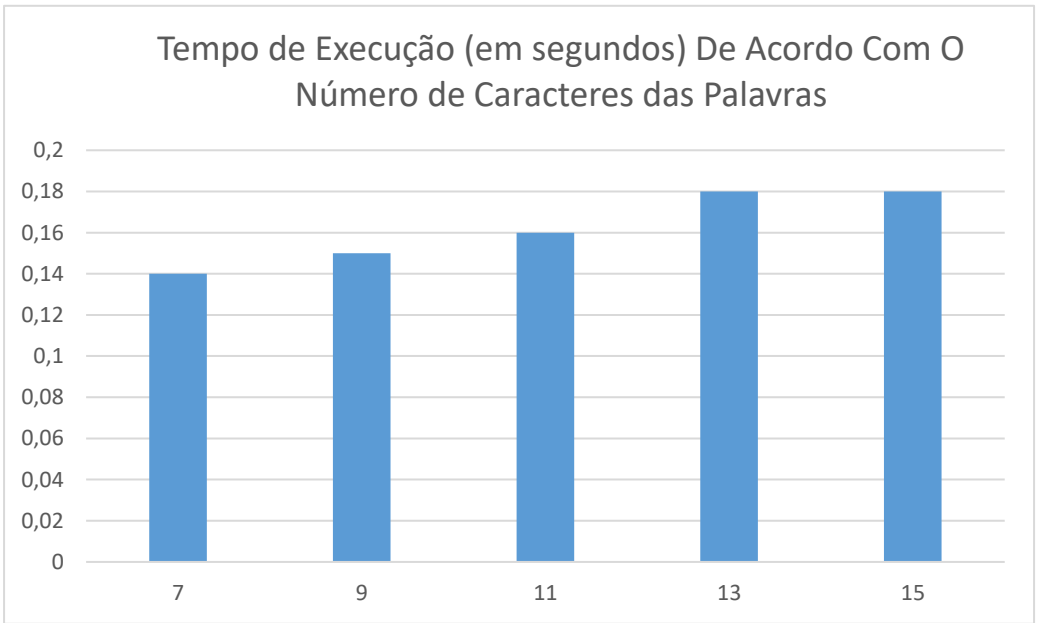
Os testes a seguir foram feitos em um computador com processador core i7 4770 4Ghz com 16GB de memória RAM. O sistema operacional utilizado foi o Fedora 24 e o compilador gcc. Os testes realizados foram os seguintes: no primeiro, o número de caracteres de cada palavra foi fixado em 15 e o texto tinha um tamanho fixo de 10^7 palavras. O que variou foi o número de palavras no dicionário, como mostra o gráfico abaixo:



O consumo de memória neste teste pode ser verificado na tabela abaixo:

WORD_SIZE 15 / TEXT_SIZE 10000000	Bytes allocated
DIC_SIZE 10	37776 bytes
DIC_SIZE 100	314040 bytes
DIC_SIZE 1000	2952264 bytes
DIC_SIZE 10000	27939576 bytes
DIC_SIZE 100000	263271360 bytes

O segundo teste foi feito de maneira um pouco diferente. O dicionário foi fixado em 10^5 palavras e o texto em 10^6 palavras. O que variou foi o tamanho de cada palavra, como pode-se ver no gráfico abaixo:



Neste segundo teste, o consumo de memória se comportou da seguinte forma:

DIC_SIZE 10000 / TEXT_SIZE 1000000	Bytes allocated
WORD_SIZE 7	10663464 bytes
WORD_SIZE 9	14986488 bytes
WORD_SIZE 11	19291368 bytes
WORD_SIZE 13	23628216 bytes
WORD_SIZE 15	27934824 bytes

É possível perceber que houve um salto no tempo gasto para a execução do último caso do primeiro teste. Isso provavelmente ocorreu pela lentidão da função read_dictionary_words_from_stdin(); que teve que percorrer uma lista muito grande. Todos os outros testes correram como esperado.

6 Conclusão

Neste trabalho foi implementada uma árvore trie que funcionava como um dicionário. Foi muito interessante perceber o quanto essa estrutura de dados é eficiente na inserção de uma nova palavra e na busca. Entretanto, o custo de memória é alto. Caso o limite máximo de palavras no dicionário fosse maior, talvez fosse mais vantajoso modelar o problema de outra forma, sem utilizar um vetor de 26 posições para cada nodo, por exemplo. É importante notar que essa estrutura de dados é amplamente utilizada no mundo real, então estudar sua implementação é muito importante.