Documentação - Trabalho Prático 0

Esthefanie Pessoa Lanza - esthefanielanza@gmail.com

Setembro, 2016

1 Introdução

Atualmente, devido a constantes mudanças de documentos escritos para suas versões digitais, encontrar palavras ou trechos em um texto é uma função extremamente utilizada em uma vasta gama de programas. Dessa forma, é necessário estudar a implementação de algoritmos eficientes, com baixo custo, para realizar pesquisas digitais. Uma das principais estruturas utilizadas para esse tipo de busca é a árvore Trie, que será discutida e implementada no decorrer deste trabalho.

Diferente da árvore binária, os nós de uma Trie não possuem chaves associadas a elas, ou seja, estas são indicadas a partir da posição do nó na árvore. Por exemplo, em um alfabeto de n caracteres cada letra será associada a um índice correspondente a um dos n nós filhos.

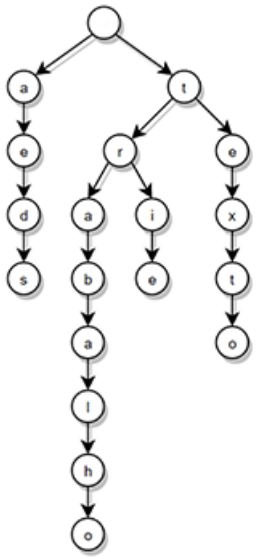


Figure 1: Exemplo de árvore Trie

Como representado na figura 1, a raiz é associada a uma cadeia vazia e cada nó possuí n apontadores, onde n é o número de caracteres presentes no alfabeto. Sua formação também contribuí para que todos os nós descendentes possuam um prefixo comum com a cadeia associada anteriormente.

O objetivo deste trabalho é a partir de um dicionário construir uma árvore Trie e dado um outro conjunto de palavras, identificar se elas estão presentes ou não na estrutura e o número de vezes que aparecem.

2 Solução do Problema

2.1 Modelagem

A árvore foi implementada a partir de estruturas de dados chamadas no código de TNode, ou em outras palavras, tipo nó. Assim como representado na figura 2, cada nó possui 26 apontadores para outros nós, correspondentes ao alfabeto minúsculo(a-z), que inicialmente são iniciados como nulos. Além disso, essa estrutura possui um flag para identificar o fim de uma palavra e um contador de frequência que, de acordo com a especificação, deveria ser implementado na mesma estrutura da árvore.

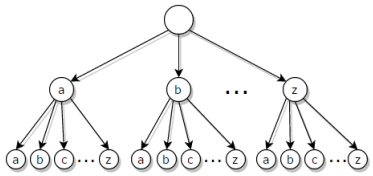


Figure 2: Exemplo estrutura TNode

2.2 Implementação

O fluxo do programa é inciado com a criação de um nó vazio que representará a raiz. Após isso é realizado a leitura do número de palavras presentes no dicionário, que através de um loop são inseridas uma a uma na Trie. A função Insere Palavra recebe como argumentos o nó raiz da Trie e a palavra em questão. Para inserir uma palavra, identificamos primeiramente o tamanho da mesma. Em seguida, a cada iteração de um loop for, que vai até o tamanho da palavra, criamos um nó, quando não existente, referente a cada caractere presente. Ao término da inserção, indicamos no nó do ultimo caractere que ele, em conjunto com seus antecessores, formam uma palavra. No loop externo à função cada palavra do dicionário é armazenada em um vetor de strings para ser usado posteriormente.

Algorithm 1: Insere palavra(TNode node, String palavra)

```
// Calcula o tamanho da palavra lida
tamanhoPalavra = calculaTamanho(palavra)
// Enquanto não lermos todos os caracteres da palavra
for i=0 até tamanhoPalavra do
    // Calculamos o índice representado por aquele caractere
    indice = calculaIndice(caractere)
    // Se ainda não existe nó para o índice atual
    if !node → filhos[i] then
        // Criamos o nó
        node → filhos[índice] = CriaNó()
    end
    // Caminha para o próximo nó
    node = node → filhos[índice]
end
// Ao terminar de ler a palavra indicamos o seu final
node → final = true
```

É importante ressaltar que, para a criação do índice, é realizada uma conversão de caractere para números inteiros que vão de 0 a 26. Para realizar essa transformação usamos o valor correspondente do caractere na tabela ASCII menos o valor correspondente a primeira letra do alfabeto, no caso 'a'. A equação é descrita como a fórmula abaixo:

$$indiceCaractere = ValorCaractereASCII - 97 \tag{1}$$

Em seguida, devem ser lidas as palavras pertencentes ao texto, usando um for, estas são verificadas uma por vez sem ser armazenadas. O algoritmo usado para procurar as palavras recebe como parâmetro apenas o nó raiz da Trie já que as palavras serão lidas durante sua execução. Através da função, o algoritmo, assim como na inserção, calcula o tamanho da palavra e faz as conversões necessárias para transformar o valor do caractere em ASCII em um número inteiro. Caso encontremos um nó nulo, retornamos ao programa principal, pois a palavra não está presente no dicionário. Por outro lado, se conseguimos sair do loop, verificamos se é uma palavra valida e incrementamos sua frequência.

Algorithm 2: Procura palavra(TNode node)

```
// Leitura da palavra
palavra = lePalavra() // Calcula o tamanho da palavra lida
tamanhoPalavra = calculaTamanho(palavra)
// Enquanto não lermos todos os caracteres da palavra
for i=0 até tamanhoPalavra do
    // Calculamos o indice representado por aquele caractere
    índice = calculaIndice(caractere)
    // Se não existe um nó
    if !node → filhos[indice] then
        // Retornamos pois a palavra não foi encontrada
        return
    end
    // Caminha para o próximo nó
    node = node → filhos[índice]
end
// Incrementamos a frequência caso seja um nó valido
if node → final then
    node → frequência + +
end
```

Após todas palavras do texto serem visitadas o algoritmo usa as palavras do dicionário que anteriormente foram armazenas em um vetor para procura-las novamente e imprimir suas frequências na ordem correta. O algoritmo para realizar a impressão funciona como o de busca, porém ao invés de incrementar a frequência utiliza-se uma função para imprimir seu valor no console.

3 Análise de Complexidade

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço.

3.1 Análise Teórica do Custo Assintótico de Tempo

Podemos analisar o custo assintótico avaliando as funções de inserção, busca e impressão da frequência.

- O algoritmo de inserção, a partir do nó raiz e da palavra lida, insere cada caractere na Trie com custo $O(1)$, pois podemos encontrar a posição de dada letra realizando o calculo do índice descrito na equação 1. Como essa operação é a realizada c vezes, onde c é o número de caracteres da palavra o custo da função é $O(c)$. Porém, temos que o maior tamanho de palavra aceito para uma entrada, de acordo com a especificação, é 15. Dessa forma, o custo assintótico é constante, ou seja, $O(1)$.

- O algoritmo de busca, assim como na função de inserção, também usa os índices dos caracteres para encontra-los com custo $O(1)$. A operação é

realizada no pior caso com custo $O(15)$, referente ao tamanho máximo da palavra. Assim, como custo assintótico é constante temos que a complexidade é equivalente a $O(1)$.

- O algoritmo de impressão da frequência também tem seu funcionamento similar aos de inserção e busca. Porém este tem como parâmetro o número de palavras do dicionário. A função, em um loop for, imprime a frequência das n palavras presentes no dicionário. Dessa forma, temos que a complexidade é equivalente a $O(n)$.

- No programa principal, o algoritmos de busca é chamado n vezes, sendo n o número de palavras presente no dicionário. Por outro lado, a função de busca é chamada m vezes, sendo m o número de palavras presentes no texto. Assim temos que, o custo assintótico de tempo do programa será equivalente ao maior valor entre n e m. De maneira geral, $O(max(n, m))$.

3.2 Análise Teórica do Custo Assintótico de Espaço

Podemos analisar o custo assintótico avaliando as funções de inserção, busca e o seus loop externos.

- A função de inserção armazena cada caractere das palavras lidas em um nó da árvore. Como cada palavra pode ter um tamanho máximo constante, o espaço gasto para a alocação da árvore depende apenas do número de palavras inseridas no dicionário, ou seja, $O(n)$.

- O loop que envolve a função de inserção armazena todas as palavras do dicionário em um vetor de palavras, ou seja, o custo de espaço será equivalente a o tamanho máximo da palavra vezes o número de palavras adicionadas. Porém como o tamanho máximo de uma palavra é constante a complexidade, de maneira geral, é $O(n)$.

- Na função de busca as palavras são armazenadas temporariamente em um vetor de tamanho 15, ou seja, a complexidade de espaço é constante independente do número de palavras no texto. Dessa forma, temos que o custo é $O(1)$.

- No programa principal, temos que a ordem de complexidade em relação ao espaço dependerá somente do número de palavras do dicionário, ou seja, $O(n)$.

4 Avaliação Experiental

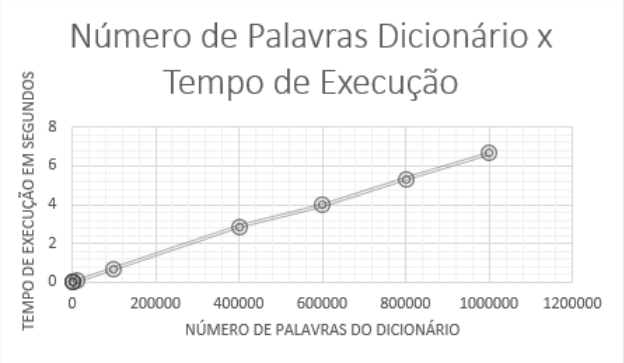
4.1 Metodologia

Os testes consistem em palavras de 15 caracteres gerados aleatoriamente. Cada teste foi realizado 30 vezes e foi feita uma média dos valores encontrados. O principio básico é manter uma das entradas constante em 100 enquanto variamos

a outra. A implementação da solução proposta foi compilada utilizando gcc 4.8.4. Os experimentos foram executados em um sistema com 4 GB de memória e um processador Intel Core i3 2310K @ 2.1 GHz.

4.2 Análise experimental em relação ao tempo

Para analisar o quanto o número de palavras inseridas no dicionário influencia no tempo de execução foram criados vários casos de teste variando o número de palavras do dicionário de 10 até 10^5 enquanto o número de palavras do texto permanecia constante em 100 palavras. O gráfico ilustrado na figura 3 demonstra o comportamento do programa.

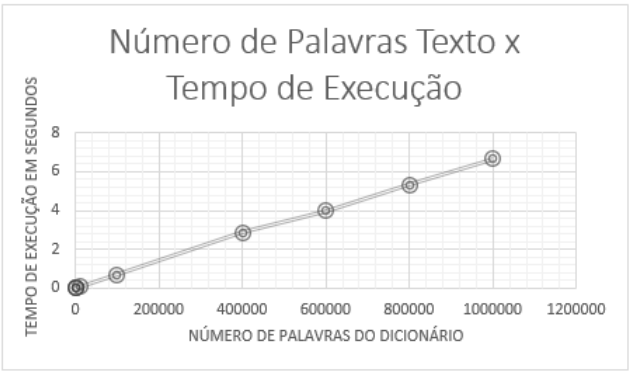


Número de Palavras	Tempo de Execução
10	0,0020
100	0,0040
1000	0,0180
10000	0,1000
100000	0,7085
400000	2,8640
600000	4,0065
800000	5,3235
1000000	6,6740

Figure 3: Análise experimental de tempo variando o número de palavras inseridas no dicionário

Como esperado, o tempo de execução cresce linearmente enquanto aumentamos o número de palavras. O pior caso, quando temos 10^5 palavras de 15 caracteres cada o, leva em torno de 6,67 segundos para ser executado.

Em seguida foi realizado o mesmo teste, mas foi mantido o número de palavras inseridas no dicionário constante em 100 palavras enquanto aumentamos o número de palavras do texto. Da mesma forma, foram testadas entradas que variam de 10 até 10^7 palavras.



Número de Palavras	Tempo de Execução
1000	0,005
10000	0,009
100000	0,043
1000000	0,227
10000000	1,811
40000000	7,067
60000000	10,584
80000000	14,094
100000000	17,583

Figure 4: Análise experimental de tempo variando o número de palavras inseridas no texto

Entretanto, devido a falta de recursos computacionais, não foi possível gerar entradas maiores que 10^7, como solicitado no enunciado. O sistema onde o programa foi testado ficou extremamente lento ao manipular arquivos na casa dos Gigas. Porém, através do cálculo da inclinação da reta descrito na equação 2, foi estimado o tempo gasto para a execução de entradas maiores que as suportadas pela máquina.

$$Inclinação = \frac{\Delta Tempo}{\Delta nPalavras} = 1,8945x10^{-7} \tag{2}$$

Assim temos que uma entrada de tamanho 10^{16} seria executado em 31,575 milhões de minutos, em outras palavras, a execução demoraia o equivalente a 60 anos para ser executada.

Foi possível perceber também, comparando as inclinações das retas, que o programa é menos sensível ao número de palavras inseridas no texto ao comparalas com inserções no dicionário. Isso ocorre pois as operações realizadas para adicionar uma palavra no dicionário e imprimi-las na ordem correta posteriormente são mais custosas do que apenas verificar se elas existem ou não na Trie. Por exemplo, uma entrada com 10^{16} palavras no dicionário demoraria cerca de 213 anos para ser executada enquanto o mesmo número de palavras inseridas no texto demoraria apenas 55 anos.

4.3 Análise experimental em relação ao espaço

A memória alocada foi testada com a ferramenta Valgrind e também cresce linearmente em relação ao número de palavras inseridas no dicionário. O programa agiu como o esperado e teve de alocar o equivalente a 2482 Mbytes para rodar o teste onde inserimos 10^5 palavras no dicionário.

Número de Palavras	Memória Alocada
10	0,0322
100	0,3091
1000	2,9456
10000	27,9281
100000	263,2284
400000	1020,0443
600000	1512,8140
800000	1999,5156
1000000	2482,4057

Figure 5: Análise experimental de espaço variando o número de palavras inseridas no dicionário

Por outro lado, variar o número de palavras do texto não interferiu no espaço alocado pelo programa. Dessa forma o gráfico gerado para diferentes entradas permaneceu constante como demonstrado na figura 6.

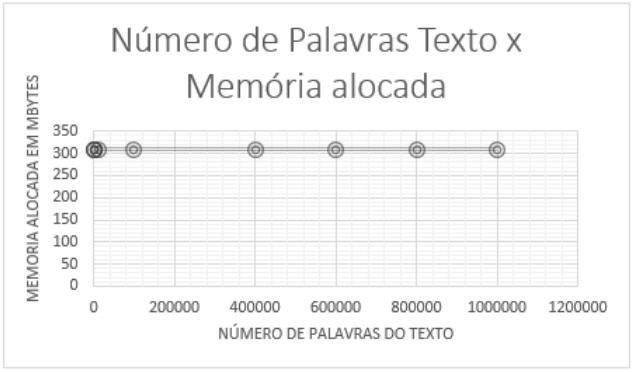


Figure 6: Análise experimental de espaço variando o número de palavras inseridas no texto

5 Conclusão

O trabalho permitiu implementar a árvore Trie e pôr à prova sua eficiência ao procurar palavras em um texto. Sua complexidade de ordem linear permite computar textos de ordem 10^7 em tempos razoáveis. Entretanto, ao adicionar muitas palavras ao dicionário é alocado um espaço considerável na ordem de Gigabytes.

A implementação não foi complicada, porém foi utilizada uma estratégia de armazenar as palavras do dicionário em um vetor alheio à árvore para imprimi-

las em ordem posteriormente. Utilizar outros meios poderia diminuir os tempos de execução e a memória alocada.

Gerar os casos testes e observar sua execução exigiu bastante tempo e poder de processamento da máquina utilizada para os testes, porém os resultados obtidos foram capazes de confirmar o que havia sido previsto na análise de complexidade.