Trabalho Prático Aeds III
Pedro Nascimento Costa
TP0

Introdução:

O trabalho apresenta como proposta trabalhar com a estrutura de uma árvore trie para realizar reconhecimento de palavras em um texto. Temos então um dicionário de palavras e um texto no qual deseja-se identificar a ocorrência de palavras do dicionário.

A árvore trie é um tipo de estrutura relativamente simples de se entender, é também conhecida como uma árvore de prefixos, tendo isso em mente, na implementação do trabalho foi utilizada uma aplicação simples de árvore trie na qual palavras de mesmo prefixo tem o mesmo caminho percorrido até o ponto em que se diferem. Para marcação de ocorrência de uma palavra, foi utilizada a última letra de cada palavra.

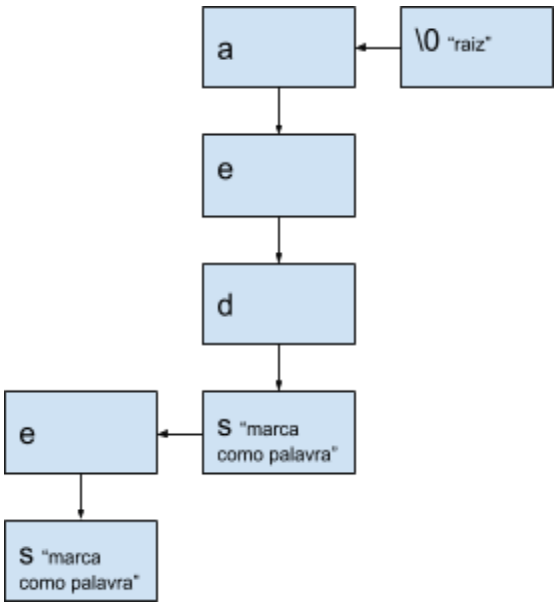


Figura I

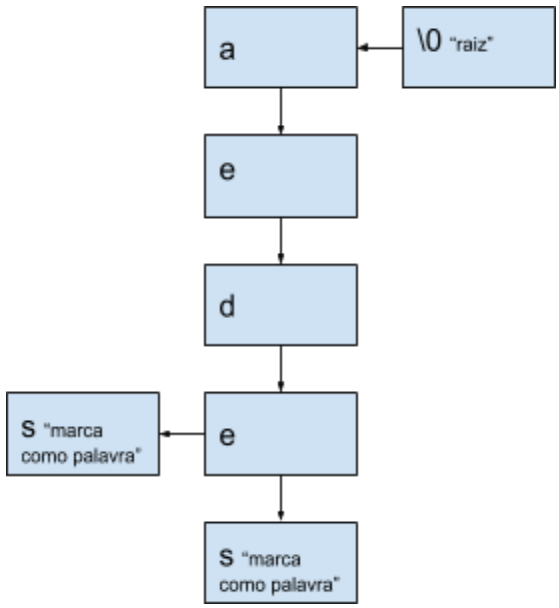


Figura II

A Figura I representa, de uma maneira simples o funcionamento da árvore trie que foi utilizada na implementação do trabalho, nela temos a inserção de duas palavras, são elas: “aeds” e “aedes”, que nessa mesma ordem, foram inseridas na árvore. Uma observação a se fazer, caso mudassem a ordem de inserção, a árvore se formaria de maneira diferente, no caso, a maneira representada na Figura II.

Desenvolvimento:

Para implementar o trabalho, foram escolhidas duas TADs a serem utilizadas em junção com a main, na implementação, a aplicação da árvore trie escolhida deve-se a simplicidade encontrada para entender e usá-la.

Foram criados três partes para implementação, diga-se três módulos, um para tratar da junção e execução de tudo, o módulo da 'main', um para tratar toda parte do programa relacionado com a árvore trie e um último que envolve manutenção da entrada.

As TADs estão separadas de acordo com um tema, uma trata de toda a parte funcional da árvore trie implementada, para ela, criou-se um estrutura de dado específica para representar os nós, nela temos indicadores para nós seguintes e marcadores indicando se temos uma palavra formada ao chegar naquele nó e um contador de ocorrências. A outra TAD faz a manipulação da memória a ser utilizada pelo dicionário e trata também de desalocar toda a memória utilizada pelas entradas do programa ao final da execução. O diagrama abaixo retrata a relação entre os Módulos, as TADs e todo o programa.

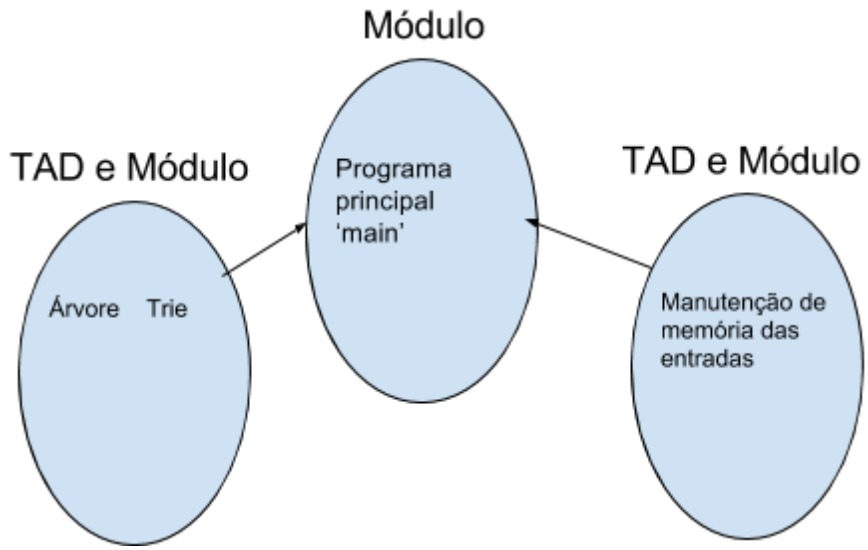


Figura III

Na Figura III, temos, de maneira simples, a representação de interação e dependência entre todos os componentes da implementação, as duas TADs são independentes entre si, no sentido de que uma não chama partes da outra diretamente, entretanto, uma utiliza dados organizados pela outra nas chamadas de seus componentes dentro do programa principal que, por sua vez, depende diretamente de ambas as TADs.

A entrada de dados do programa consiste em dois blocos de texto, um o dicionário a ser usado e o outro, o texto no qual busca-se as ocorrências de palavras do dicionário. São quatro linhas de entrada, a primeira nos apresenta o número de palavras do dicionário, a segunda contém todas as palavras do dicionário, a terceira, apresenta, assim como a primeira, um inteiro, dessa vez o número de palavras do texto e, por fim a última contém todo o texto. A saída é bem simples, ela contendo inteiros indicando, na mesma ordem de entrada das palavras do dicionário, a quantidade de ocorrências de cada palavra no texto. Ambas entradas e saídas são a padrão, stdin/stdout.

De maneira simples, são três os passos principais do programa, primeiro temos a criação da árvore através da inserção do dicionário nela, esse é nosso primeiro passo, o segundo consiste na pesquisa e marcação de ocorrência, caso exista, de palavras do dicionário no texto e, por fim, o terceiro passo é a pesquisa e impressão da quantidade das ocorrências de cada palavra, na mesma ordem em que foram inseridas na árvore.

Para se executar corretamente o programa, primeiro utilize o comando make, assim fazendo uso do makefile para compilar, em seguida, para executar, deve se digitar no terminal apenas: ./exec
em seguida entre com os dados de entrada, de acordo com o especificado acima.

Analisando a Complexidade:

Tempo:

São três as principais ações, três Loops, que influenciam no tempo de execução, eles podem ser identificados como os que contém as funções principais do programa, a de inserção que cria a árvore trie com o dicionário, a que pesquisa a ocorrência de palavras do dicionário no texto e a que busca os valores da ocorrência de cada palavra no texto. As demais operações envolvem simples atribuições, ou leituras, ou alocações.

No primeiro Loop, o de inserção, temos n execuções, sendo n o número de palavras do dicionário, ou seja, seu tamanho, em cada execução temos três operações de ordem O(1) para atribuição, leituras e alocação, além disso temos a função de inserção que realiza algo próximo de t operações, dividido entre comparações e inserções, sendo t o tamanho da atual palavra sendo inserida, podendo variar entre 1 e 15.

No segundo Loop, o de pesquisa de ocorrências no texto, temos m execuções, em cada temos uma operação O(1) de leitura e algo próximo de t operaçoes de comparação na função de pesquisa, sendo t o tamanho da palavra do texto, podendo variar entre 1 e 15, assim como as do dicionário no Loop 1.

No terceiro Loop, o que busca a quantidade de ocorrências, temos n execuções, sendo n o tamanho do dicionário, nele realizamos t comparações, sendo t o tamanho da palavra atual do dicionário sendo utilizada na execução, podendo variar de 1 a 15.

Juntando tudo, temos: $n + n + n + n*t + m + m*t + n*t = 3n+2(n*t) + m + m*t$, que nos leva a ordem de $O(n+m+t(2n+m))$ no pior caso, que é algo entre $O(n)$ e de $O(n^2)$, uma entre ordem linear e quadrática.

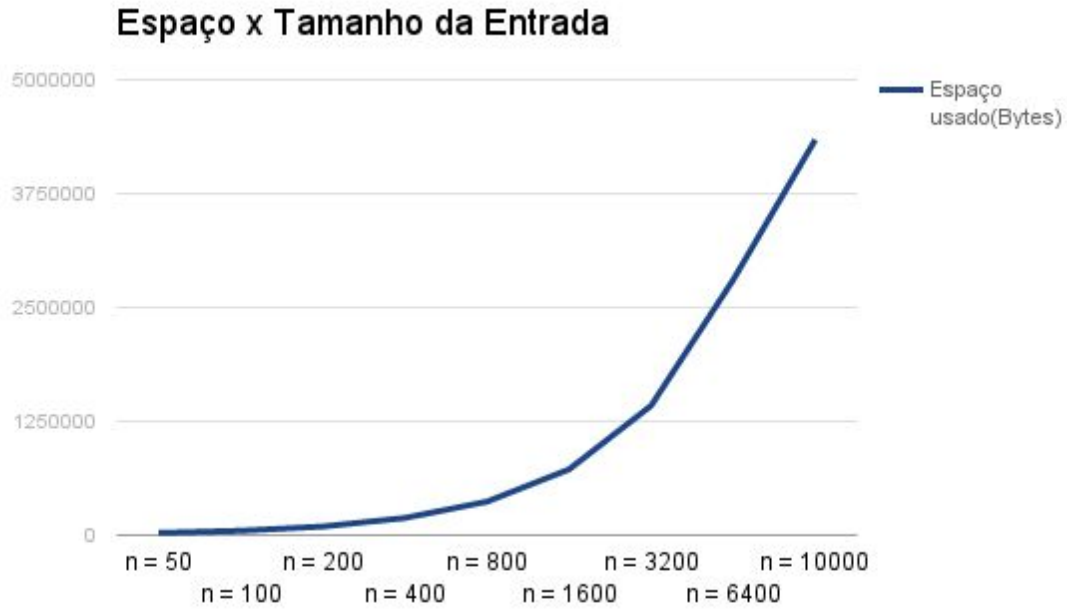
No Gráfico abaixo temos a representação do tempo de execução através de diversos testes realizados, no caso, em todos os mementos temos $t = 15$ e variados valores para m e n, percebe-se que ele aproxima-se de algo linear entre $O(n)$ e $O(n^2)$, aumentando com o tamanho de n e m, como analisado anteriormente. Para determinar os tempos apresentados no gráfico, foi utilizado a biblioteca “time.h”, juntamente com sua função “clock_t”.

Espaço:

Em termos de espaço, há dois principais gastos, um com a àrvore trie e outro com o dicionário no qual armazenamos várias palavras. Em ambos os casos, temos dependência de dois fatores n e t, sendo n o número de palavras do dicionário e t o tamanho de cada palavra, podendo variar de 1 a 15, além disso, temos também o espaço de uma palavra extra, que é usada como auxiliar para inserir e ler o texto da entrada.
Assim temos um espaço gasto de:

n*t + t, que nos da algo do tipo O(t(n+1)). Vemos então que o espaço gasto é de ordem relativamente linear, crescendo com o tamanho de n.

O Gráfico abaixo representa diversos testes realizados com valores crescentes de n, temos o aumento do espaço gasto com o aumento do tamanho de n, para tal estamos usando um t de tamanho constante igual a 15, ou seja todas as palavras tem tamanho máximo. Para cálculo de quantidade de espaço gasto, foi utilizado o valgrind.
Podemos observar, que ao dobrar o tamanho de n, o número de bytes cresce relativamente proporcionalmente, assim temos algo linear, lembrando que para o exemplo abaixo, foi utilizado o pior tamanho possível para t em todas as palavras.



Conclusão

A implementação ocorreu sem problemas, ao analisar os resultados, podemos concluir que o fator mais importante da entrada, em termos de espaço e tempo, pois afeta tanto o tempo de execução, quanto o espaço gasto, é o tamanho do dicionário, o n. Além disso, a escolha de árvore trie para implementar é bem interessante, visto que palavras de mesmo prefixo compartilham espaço na árvore, fazendo com que se gaste menos espaço na implementação, quando comparada a uma aplicação que cada palavra tem seu espaço.

Referências

http://stackoverflow.com/questions/5248915/execution-time-of-c-program
http://www.ufjf.br/jairo_souza/files/2009/12/6-Strings-Pesquisa-Digital.pdf
http://stackoverflow.com/questions/19068643/dynamic-memory-allocation-for-pointer-arrays
http://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/radixsearch.pdf