

UNIVERSIDADE FEDERAL DE OURO PRETO
PROGRAMA DE PÓS GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PROJETO E ANÁLISE DE ALGORITMOS – TRABALHO PRÁTICO I

Trabalho apresentado ao mestrado em Ciência da Computação do departamento de Computação, da Universidade Federal de Ouro Preto, como parte da avaliação do corrente semestre.

Professores: Haroldo Gambini Santos, Rodrigo Cesar Pedrosa Silva

Ítalo A. Aguiar

Ouro Preto
2020

Sumário

Introdução	3
Árvores Digitais – Tries	3
Definindo a estrutura da árvore	3
Carregando dados na árvore	4
Algoritmo de Busca	5
Ordenação	5
Comprimindo a estrutura da árvore	5
Desempenho	6
Conclusão	7
Referências	8

1. Introdução

Lidamos com sistemas interativos todos os dias. Em grande parte destes sistemas existe a necessidade de se pesquisar em cima de determinado texto, conteúdo, ou base de dados. Um dos usos mais comuns deste tipo de pesquisa está na sugestão e complemento baseado em um prefixo. Por exemplo, utilizamos o *Google*¹ todos os dias, e sempre que começamos a digitar em seu campo de pesquisa, o sistema nos apresenta uma série de sugestões de busca baseado no termo digitado.

Em ciência da computação também possuímos um recurso muito comum à maioria dos programadores, presente em grande parte dos Ambientes de Desenvolvimento Integrado, ou simplesmente, IDEs. No caso, este recurso é o autocompletar sentenças, onde o sistema permite um ganho de produtividade ao permitir escolher rapidamente o método, propriedade, evento, campo, etc. de forma simples baseado em poucos caracteres digitados.

Neste trabalho foi desenvolvido uma implementação de árvore digital de busca, capaz de armazenar a frequência em que determinada palavra se repete e também capaz de apresentar resultados de busca de forma ordenada. Além disso, este trabalho também apresenta algumas métricas de desempenho coletadas a partir da execução e instrumentação do código fonte do programa.

2. Árvores Digitais – Tries

Uma árvore digital é uma árvore de busca especialmente útil para auto sugerir termos e sentenças. Esta estrutura de dados foi inicialmente proposta por Edward Fredkin, e foi nomeada baseada no termo do inglês *retrieval*, que significa recuperação, no caso, recuperação de dados[1]. Na Figura 1 podemos ver um exemplo de Trie.

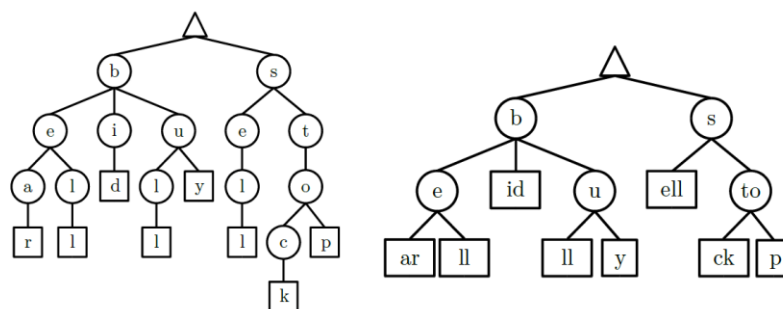


Figura 1 – Exemplo de árvore digital. Fonte: Roteiro do trabalho.

Em geral, esta estrutura apresenta alguns benefícios quando comparado às tradicionais árvores de busca binária. Entre estes benefícios, está a velocidade de busca, uma vez que o acesso dos nós da árvore são baseados em chaves. Além disso, as árvores digitais possuem menor profundidade quando comparadas com a estrutura correspondente de uma árvore binária simples.

3. Definindo a estrutura da árvore

¹ <http://www.google.com>

O primeiro passo para implementar o algoritmo da árvore é definir a forma com que os dados serão armazenados em memória principal (RAM) durante a execução do *software*. Em C, podemos definir estruturas de dados através da declaração da forma com que estes dados deverão ser armazenados, utilizando uma notação da própria linguagem de programação (Figura 2).

```
struct Node {  
    int frequency;  
    struct Node* childs[ALFABET_SIZE];  
};
```

Figura 2 – Estrutura de dados adotada para a árvore de busca

As inserções na árvore digital consistem em inserir os caracteres de determinada palavra em sua estrutura. Dessa forma, cada nó da árvore irá corresponder a um caractere do alfabeto que a árvore é capaz de trabalhar. Note que não é armazenado em nenhum momento o caractere dentro de cada nó da árvore. Para resolver essa questão, cada nó da árvore possui um conjunto (*array*) de nós com exato tamanho do alfabeto. Cada posição deste conjunto é específica, e pertence somente a determinado caractere. Desta forma, para se recuperar qualquer dado a partir de um nó da árvore, basta selecionar caractere por caractere do termo de busca e verificar se o índice correspondente nos filhos do nó é não nulo. Assim, é possível navegar entre cada um dos nós da árvore até que se encontre o nó final da ramificação.

Um dos requisitos deste trabalho consiste em armazenar o número de vezes em que determinado termo se repete no texto de entrada. Perceba que a princípio, a estrutura de dados vista na Figura 2 não depende de nenhum outro dado interno além de seus próprios filhos e um marcador de finalização de palavra. Porém, para se atender aos requisitos do trabalho, o campo denominado frequência, fica encarregado de armazenar a quantidade de repetições de determinado termo. Essa abordagem simplesmente substitui o marcador de finalização de palavra por este contador, assim, sempre que este contador é maior que zero, o nó em questão corresponde a uma palavra completa.

4. Carregando dados na árvore

Definir a estrutura que irá armazenar os dados é o ponto inicial deste trabalho, porém, uma vez que as estruturas estejam definidas, é preciso preenche-las com dados. A primeira parte desta tarefa consiste na leitura de um conjunto de textos de um arquivo. A estratégia aqui adotada foi utilizar um *Buffer* de leitura de disco. Um *Buffer* permite carregar um conjunto de dados ao invés de se carregar caractere por caractere, ou carregar todo o arquivo em memória de uma única vez.

A cada leitura em buffer, o conteúdo em texto era percorrido, separando-se cada um dos caracteres presentes. Alguns caracteres não são relevantes para uma árvore digital, como por exemplo, sinais de pontuação e caracteres especiais. Dessa forma, uma função auxiliar foi escrita para permitir selecionar somente os caracteres de interesse. Neste caso, os caracteres que nos interessam são as letras comuns, maiúsculas e minúsculas, e o hífen.

A estratégia de inserção consiste em partir da raiz da árvore e à medida em que novos caracteres são inseridos, novos níveis são criados na árvore. A função de inserção sempre recebe o nó de partida e retorna o nó em que parou, desta forma, é possível inserir todos os

caracteres da palavra. Quando um caractere não esperado passa pela função de inserção, isto indica que a palavra finalizou, logo o contador de frequência do nó é incrementado, e o retorno da função passa a ser a raiz para permitir novas inserções.

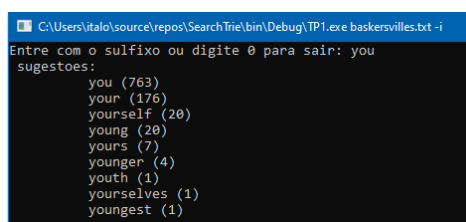
5. Algoritmo de Busca

O Algoritmo de busca é relativamente simples. Ele consiste em percorrer a árvore baseado no prefixo informado de forma iterativa. Dado um nó de partida, o algoritmo pega o primeiro caractere da entrada e verifica se possui um filho correspondente, e, caso afirmativo, o algoritmo passa para o próximo caractere e o procura neste filho, repetindo este passo até que toda a entrada tenha sido consumida. Quando a entrada é totalmente consumida, isso significa que há possíveis resultados na árvore. Sendo assim, uma outra função, desta vez recursiva, é responsável por imprimir todas as possibilidades combinadas com o prefixo informado.

6. Ordenação

Um dos requisitos deste trabalho foi definido como necessária a apresentação dos resultados ordenados por frequência de repetição das palavras. Então, para atender a este requisito, uma estrutura adicional de lista encadeada foi implementada de modo a armazenar a palavra e sua respectiva frequência.

O passo seguinte foi escolher um algoritmo que pudesse fazer a ordenação destes resultados. Inicialmente foi cogitado a possibilidade de se empregar algoritmos conhecidamente mais eficientes, como por exemplo, o *QuickSort*[2]. Porém, pensando-se em questões de tempo, simplicidade de implementação, depuração e testes, foi optado o *BubbleSort*[2]. Podemos ver na Figura 3 um exemplo de resposta ordenada da implementação.



```
C:\Users\italo\source\repos\SearchTrie\bin\Debug\TP1.exe baskersvilles.txt -i
Entre com o sufixo ou digite 0 para sair: you
sugestoes:
you (763)
your (176)
yourself (20)
young (20)
yours (7)
younger (4)
youth (1)
yourselves (1)
youngest (1)
```

Figura 3 – Exemplo de resposta ordenada

7. Comprimindo a estrutura da árvore

A estratégia inicial adotada foi implementar a árvore digital simples e a partir dela, modificá-la para obter uma árvore digital reduzida. Reduzir uma árvore digital basicamente consiste em copiar o conteúdo dos nós filhos para os nós pais quando os filhos não possuem ramificações, ou seja, são nós unitários. Esta estratégia permite percorrer a árvore com menos saltos de memória, já que em determinado ponto, a leitura dos dados será sequencial a partir de uma *string*. O primeiro passo desta modificação consiste em adicionar a string que irá armazenar os dados dos filhos na estrutura de dados que definimos anteriormente, conforme podemos ver na Figura 3:

```
struct Node {
    int frequency;
    char* value;
    struct Node* childs[ALFABET_SIZE];
};
```

Figura 3 – Estrutura de dados modificada

O algoritmo implementado utiliza uma função recursiva para percorrer cada um dos nós da árvore. Ao se chegar um nó intermediário, no qual possui um único filho unitário, este nó unitário deve ser removido da memória e seu conteúdo deve ser copiado para o nó pai. O Pseudocódigo 1 mostra a lógica adotada na simplificação da árvore:

```

para cada no da arvore
    para i = 0 até Tamanho do alfabeto
        se no->frequencia = 0 e no->filho(i).filhos = 0
            no->frequencia = no->filho(i)->frequencia
            no->valor = concatenar(no->filho(i)->valor, caractere(i))
            liberar da memoria no->filho(i)
        fim se
    fim para
fim para

```

Pseudocódigo 1 – Lógica de simplificação de árvore

8. Desempenho

Analisar o desempenho do algoritmo é um passo importante para saber se determinada implementação é passível de ser utilizada em um ambiente de produção onde as cargas poderão ser elevadas, podendo haver uma necessidade de resposta no menor tempo possível.

Para analisar o desempenho do algoritmo implementado, duas abordagens foram adotadas. A primeira abordagem consiste em medir o tempo gasto para carregar um conjunto de dados de tamanhos distintos na árvore. Para se medir esse tempo, foi feito o uso da função *clock()* da biblioteca *time.h*. A função *clock* basicamente retorna o número de “ticks” da CPU desde que o programa foi iniciado. Esses “ticks” representam o número de ciclos de processamento no qual o programa acumulou ao ser executado pela CPU. Para se obter o tempo de execução entre determinados métodos, basta obter uma leitura anterior a determinado método e uma leitura posterior. Pela diferença das leituras, basta se dividir o valor pelo tempo de duração do ciclo para se obter o tempo de execução.

Nesta primeira abordagem, utilizando-se o arquivo de processamento fornecido no roteiro do trabalho, foi gerado outros 9 arquivos contendo cópias adicionais do conteúdo original. No caso, o décimo arquivo, por exemplo, possui o conteúdo do primeiro arquivo repetido dez vezes. Para cada um destes arquivos foi feita a medição do tempo gasto pelo algoritmo para processar todos os dados. O Gráfico 1 mostra os resultados obtidos neste experimento:

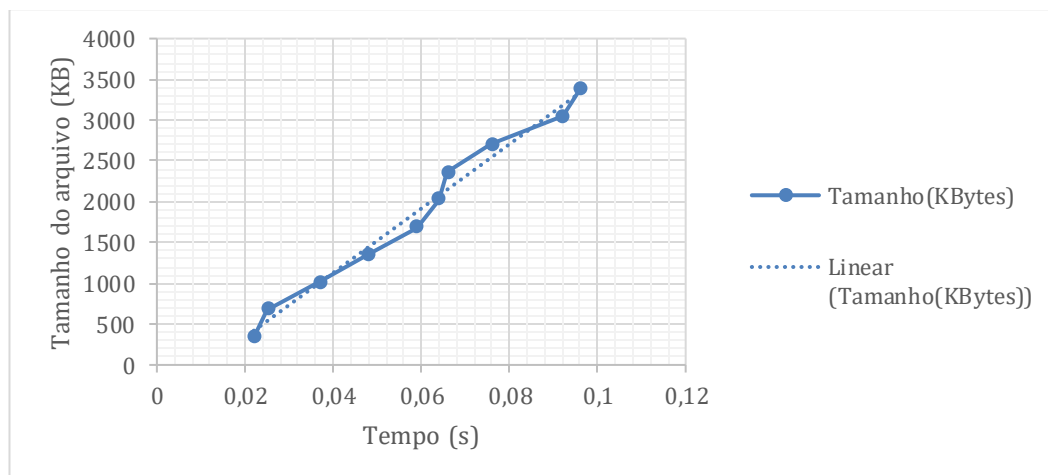


Gráfico 1 – Relação de tempo e tamanho do arquivo de entrada

A partir do Gráfico 1, podemos perceber que a relação entre o tamanho do arquivo e o tempo gasto no processamento segue uma forma linear. Neste gráfico não é possível perceber nenhuma degradação ou melhoria de desempenho ao se aumentar ou reduzir as cargas de entrada.

Na abordagem do experimento, o objetivo foi medir o tempo gasto na busca de uma árvore não reduzida. Para esse experimento, um conjunto de 40 palavras de tamanhos distintos, foi utilizado para se obter um tempo médio de busca, dada a entrada, também de tamanho médio. O Gráfico 2 apresenta os resultados coletados neste experimento:

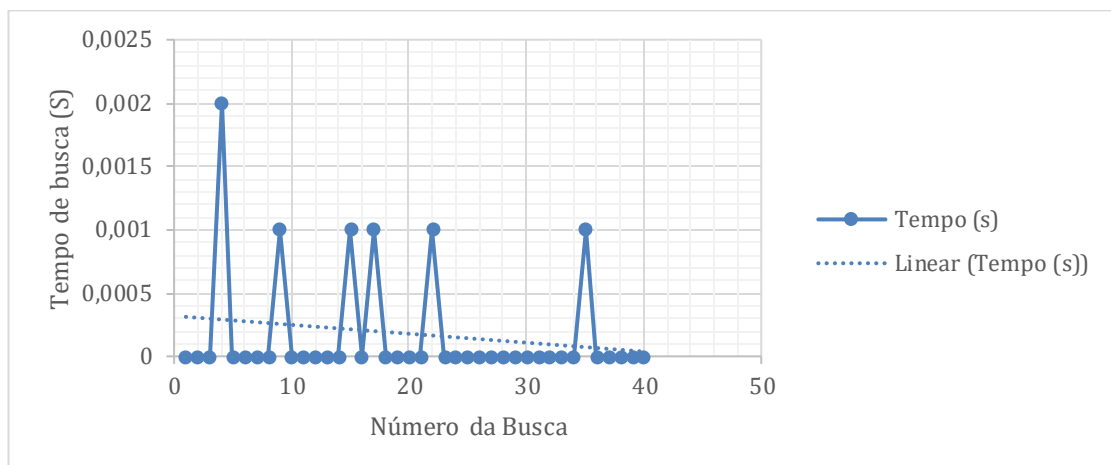


Gráfico 2 – Tempo de busca (Árvore não reduzida)

Observe que neste ponto, os resultados fornecidos pela função *clock()* não possuem uma grande precisão, ainda assim, fica fácil perceber que dada as entradas deste teste, o algoritmo obteve um desempenho a ser considerado.

Na última parte do experimento, o algoritmo de redução da árvore foi executado de modo a eliminar nós unitários da árvore. Seguindo as mesmas entradas do teste anterior, os resultados obtidos foram registrados no Gráfico 3:

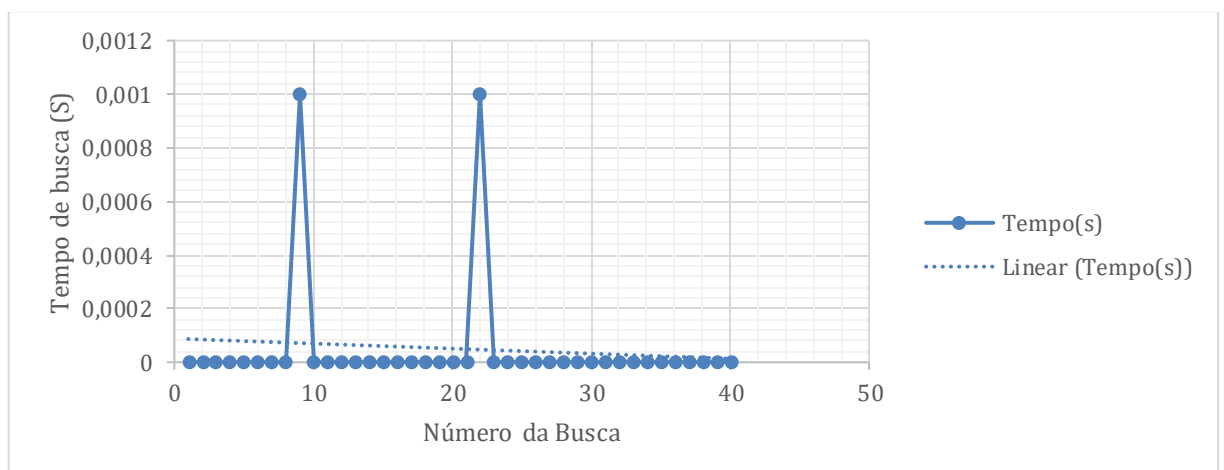


Gráfico 3 – Tempo de busca (Árvore reduzida)

Comparando-se os Gráficos 2 e 3, podemos perceber uma melhoria significativa no desempenho da busca. Essa melhoria em parte se deve ao fato da redução de apontamentos de memória, para regiões de memória contígua, no caso, *strings*, onde não são necessários tantos saltos de memória para se recuperar uma informação.

9. Conclusão

Neste trabalho foi feita a implementação de uma árvore de busca digital capaz de se auto reduzir e de apresentar seus resultados de forma ordenada. Como podemos perceber pelos resultados coletados, a estrutura possui um desempenho consideravelmente bom além de apresentar um comportamento linear à medida em que aumentamos o tamanho da entrada de dados.

Pela literatura, este tipo de estrutura apresenta uma eficiência consideravelmente mais elevada quando comparada a uma árvore binária tradicional. Desta forma, existe um grande ganho de desempenho computacional ao adotar tais estruturas[1].

Referências

- [1] FREDKIN, Edward. Trie memory. Communications of the ACM, v. 3, n. 9, p. 490-499, 1960.
- [2] ASTRACHAN, Owen. Bubble sort: an archaeological algorithmic analysis. ACM Sigcse Bulletin, v. 35, n. 1, p. 1-5, 2003.
- [3] AREF, Walid; BARBARA, Daniel. Trie structure based method and apparatus for indexing and searching handwritten databases with dynamic search sequencing. U.S. Patent n. 5,768,423, 16 jun. 1998..
- [4] Função Clock; Referência da linguagem C. Disponível em <http://www.cplusplus.com/reference/ctime/clock/> Acesso em 05/09-2020