Trabalho Prático 0: Anagramas

Gabriel Henrique Souto Pires (gabrielpires@dcc.ufmg.br)

September 21, 2015

1 Introdução

Um anagrama é uma transposição de letras de uma palavra ou frase que formam outra palavra ou frase diferente, ou seja, se uma palavra é a permutação das letras de outra, essa palavra é um anagrama. Neste trabalho o problema apresentado foi o de reconhecer anagramas em uma lista de palavras, agrupá-las e informar o tamanho de cada um desses grupos em ordem decrescente. Por exemplo, dada a seguinte frase:

- socorram me subi no onibus em marrocos

A saída do programa seria 2 2 1 1 1 já que as palavras se agrupam da seguinte forma:

- (socorram, marrocos), (me, em), (subi), (no), (onibus)

O programa processa várias listas de palavras em uma única execução até que a entrada indique que o programa deve parar.

Este tipo de problema acontece por exemplo em jogos onde o jogador deve formar várias palavras com as mesmas letras para ganhar pontos. O desafio seria identificar se uma palavra formada pelo jogador possui as mesmas letras da palavra original, e em caso positivo somar um ponto ao seu placar.

2 Modelagem do Problema

Cada lista de palavras pode ter até 10^6 palavras e cada palavra pode ter no máximo 50 caracteres, mas as listas de palavras nem sempre tem o mesmo número de palavras. Então foi utilizado um TAD Lista encadeada para armazenar as palavras, o que tornou o processo de inserção das palavras mais dinâmico e eficiente.

O TAD consiste de um conjunto de Nodos que apontam ao mesmo tempo para o seu antecessor e para o seu sucessor na lista. A vantagem de se utilizar esse tipo de TAD é que a cada vez que uma *string* for lida da entrada um nodo será alocado, fazendo com que a lista tenha exatamente o número de nodos de que necessita, fazendo com que apenas a

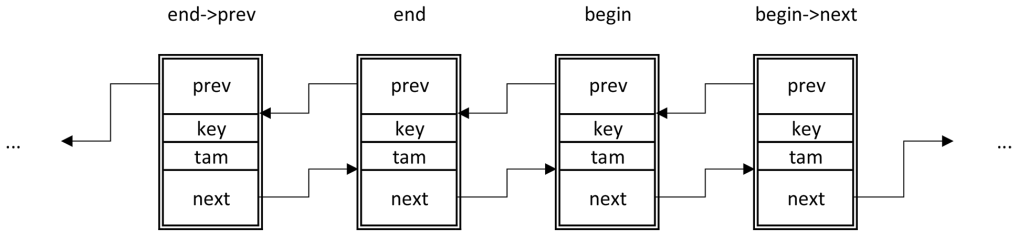

Figure 1: Representação da estrutura de dados utilizada

quantidade de memória necessária seja utilizada. A lista encadeada foi escolhida principalmente por ser um bom método de alocação dinâmica de memória e ser de fácil implementação.

Cada nodo da lista guarda a *string* recebida na entrada e uma variável *int tam* que inicialmente começa com o valor 1. Toda vez que uma palavra a ser inserida for encontrada na lista, ao invés dessa palavra ser inserida novamente apenas a variável *tam* do nodo é incrementada, dessa forma, ao acessar o valor dessa variável, é possível saber quantos anagramas foram inseridos em cada grupo.

Mas apenas inserir as palavras na lista não resolveria o problema de verificar se uma palavra é um anagrama de outra, pois apenas palavras idênticas seriam identificadas como anagramas. Para resolver esse problema foi preciso criar um método para comparar as strings inseridas de forma que fosse possível saber se duas palavras tem exatamente as mesmas letras.

Para resolver esse problema cada *string* tem seus caracteres ordenados em ordem alfabética. Por exemplo, se a palavra *batata* for inserida na lista seus caracteres são ordenados de forma que a *string aaabtt* seja inserida em seu lugar. Essa ordenação foi feita com a função *qsort()* nativa do C.

Algoritmo 1 Ordenação e inserção das strings (o loop roda enquanto o separador das strings for um espaço)

```
scanf("%d", &n);
while(n--){
    criaLista(&palavras);
    do{
        scanf("%s%c", minhaPalavra, &separador);
        qsort(minhaPalavra, strlen(minhaPalavra), sizeof(char), cmp);
        inserir(minhaPalavra, &palavras);
    }while(separador == ' ');
.
.
.
```

Após todas as palavras serem inseridas na lista, um vetor de inteiros é alocado com uma posição para cada grupo de anagramas e recebe a quantidade de palavras em cada grupo distinto, que no caso é a variável *tam* de cada nodo da lista de palavras. Este vetor é então ordenado com *qsort()* e seus valores são impressos na tela em ordem decrescente.

3 Análise Teórica do Custo Assintótico

Apesar de usar o TAD descrito acima o programa em si é muito simples, a *main* conta apenas com alguns loops para receber a entrada do programa e exibir a saída. A complexidade assintótica será discutida na subseção abaixo.

3.1 Análise Teórica do Custo Assintótico de Tempo

- A primeira parte do programa (**Algorithm 1**) recebe um número n que representa a quantidade de listas a serem processadas e roda um *while* n vezes, ou seja, *O(n)*. Dentro desse *while* a função *criaLista()* é chamada e aloca o nodo sentinela da lista em *O(1)*. Dentro desse *while* um loop *do-while* roda recebendo as strings de entrada e inserindo as palavras com seus caracteres ordenados na lista, esse loop roda uma

vez para cada palavra na lista, se considerarmos que o número de palavras = p então sua ordem de complexidade é *O(p)* e o algoritmo usado para ordenar os caracteres das palavras foi o QuickSort nativo do C (*qsort()* que roda em *O(c*logc)* com c = *número de caracteres da palavra.* A inserção na lista é feita em *O(p)*, pois a função *inserir()* percorre a lista até encontrar o local correto de inserção, que no pior caso é no final da lista.

$O(n) * (O(1) + O(p) * [O(1) + O(c * logc) + O(p)])$

$O(n) * (O(1) + O(p) + O(p * c * logc) + O(p^2))$

$O(n) + O(n * p) + O(n * p * c * logc) + O(n * p^2)$

Daí podemos concluir que a complexidade na primeira parte do programa é $O(n*p^2)$ uma vez que p é muito maior que as outras grandezas.

- A segunda parte do programa é ainda mais simples. Ainda dentro do *while* da primeira parte, um vetor de *int* chamado *grupos* é alocado em *O(1)* para armazenar a quantidade de anagramas em cada grupo. Depois disso, um loop *for* percorre a lista de palavras atribuindo o valor de cada grupo à uma posição do vetor *grupos*, essa operação executa em *O(p)*. Depois o vetor *grupos* é ordenado com o *qsort()* que roda em *O(p*logp)*. Depois de ordenado, o vetor *grupos* é exibido na tela em *O(p)* de trás para frente, já que a saída deve ser exibida em ordem decrescente. Com a saída exibida o vetor de grupos é liberado da memória com a função *free()*. Para liberar a lista de palavras, a função *liberaLista()* é usada e libera toda a memória alocada para a lista em *O(p)*.

  $O(n) * [O(1) + O(p) + O(p * logp) + O(p) + O(1) + O(p)]$

  $O(n) + O(n * p) + O(n * p * logp) + O(n * p) + O(n) + O(n * p)$

  Podemos então concluir que a complexidade da segunda parte é $O(n * p * logp)$. Então o custo total é a soma da primeira parte com a segunda, que é $O(n * p^2) + O(n * p * logp)$, ou seja, $O(n * p^2)$. Lembrando que foi considerado na análise de complexidade o pior caso da inserção, que é quando nenhuma das palavras inseridas é um anagrama, deste modo a lista terá exatamente a mesma quantidade de palavras da entrada, o que torna a complexidade do programa $O(n * p^2)$. No caso dos testes experimentais um número fixo de listas foi utilizado, então n se torna uma constante, o que faz com que o custo real do programa se torne na verdade $O(p^2)$.

3.2 Análise Teórica do Custo Assintótico de Espaço

Para analisar o custo de espaço do programa foi levado em consideração a quantidade de memória utilizada pelo TAD lista e o conteúdo armazenado nela e o tamanho do vetor de grupos alocado para exibir a saída do programa. Cada nodo da lista contém dois ponteiros para nodo, uma *string* e um *int* e a lista em si tem um *int* para guardar o tamanho da lista e um ponteiro para nodo sentinela. Considerando que um nodo é alocado para cada palavra e cada nodo tem 2 ponteiros para nodo de 8 bytes, um *int* de 4 bytes e uma *string* de ate 50 caracteres, então o custo em bytes da lista seria de *O(70*numPalavras)* ou *O(numPalavras)*. O vetor de grupos é alocado com n inteiros, sendo n o tamanho de palavras presentes na lista. Como cada inteiro ocupa 4 bytes então o custo do vetor de grupos é de *O(4*numPalavras)* ou *O(numPalavras)*, então o custo assintótico de espaço do programa é proporcional à soma do custo da lista e do vetor de grupos, ou seja, $O(numPalavras + numPalavras) = O(2 * numPalavras) = O(numPalavras)$. Pode-se

então concluir que a cada palavra lida da entrada, o programa ocupa cerca de 74 bytes na memória.

4 Análise Experimental do Custo Assintótico

Para fazer a análise experimental do programa, foi criado um outro programa que gera arquivos de entrada compatíveis de tamanhos variados de forma que a diferença do tempo de execução com entradas diferentes possa ser medido. Para não levar em consideração o tempo que se leva para digitar a entrada no tempo de execução do programa, a entrada foi lida diretamente dos arquivos. Para se medir o tempo de execução do programa foi usado o comando *time* que ao final da execução do programa grava na saída padrão (*stdout*) estatísticas de tempo sobre o programa executado. O comando completo usado na execução do programa e um exemplo de saída do comando *time* podem ser vistos abaixo:

```
time ./tp0 < input.txt

real    0m0.013s
user    0m0.000s
sys     0m0.000s
```

A saída do comando *time* pode ser interpretada da seguinte maneira:

- real: tempo real de execução. Leva em consideração todo tempo desde o início da execução até que o programa feche.
- user: o tempo de execução na CPU do comando dado pelo usuário.
- sys: o tempo em CPU gasto pelo sistema.

Para fazer a análise de tempo do programa foi levado em consideração a saída *user*.

Os testes foram realizados em uma máquina virtual rodando Xubuntu 15.04. O computador utilizado tem um processador AMD Phenom II X4 965 3.40GHz e 8GB de memória, porém, a máquina virtual utiliza apenas um núcleo do processador e 2GB de RAM.

Foram criados arquivos de teste com 10 listas de palavras, cada palavra com uma média de 40 caracteres e o tamanho das entradas variando de 1000 até 20 mil palavras. O número escolhido para o máximo de palavras não foi maior pois todos os testes foram rodados levando em consideração o pior caso possível, ou seja, o caso em que nenhuma das palavras inseridas sejam anagramas. Com estes arquivos gerados, o tempo de execução foi medido para cada execução. Como pode ser visto no gráfico abaixo, o programa de fato demonstra um comportamento assintótico de tempo quadrático como calculado anteriormente.

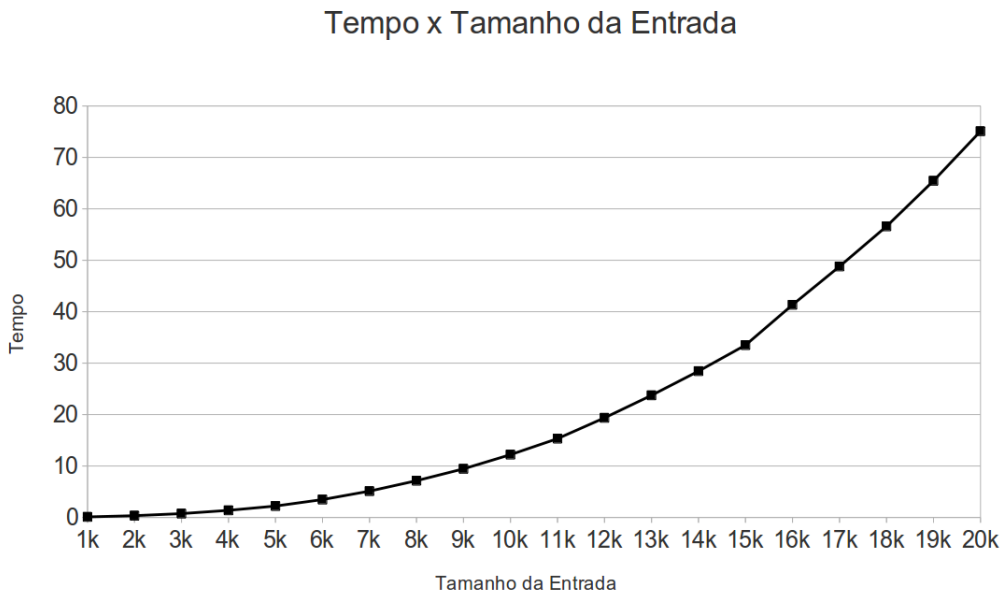

Figure 2: Gráfico da análise experimental de custo assintótico

5 Conclusão

Resolver o problema dos anagramas inicialmente parecia um pouco complexo, mas logo se mostrou simples o suficiente para que nenhum algoritmo conhecido precisasse ser implementado. O problema foi resolvido usando um tipo abstrato de dados chamado Lista Encadeada para alocar dinamicamente espaço para as palavras recebidas na entrada do programa. Antes de serem inseridas, as palavras tinham seus caracteres ordenados para que caso duas palavras tivessem as mesmas letras, ou seja, fossem um anagrama uma da outra, elas pudessem ser comparadas mais facilmente. Uma vez inseridas, as palavras representavam um grupo de anagramas. Cada vez que uma palavra igual tentasse ser inserida, o contador do tamanho do grupo era incrementado. Por fim, um vetor de inteiros armazena o tamanho de cada grupo, e após ser ordenado esse vetor é exibido de trás para frente, o que faz com que a saída fique em ordem decrescente, assim como pedido. Após ser calculada, a análise de complexidade de tempo foi provada com o auxilio de arquivos de entrada gerados por um programa à parte, que tornou possível criar arquivos com um número de palavras grande o suficiente para notar a diferença entre o tempo de execução do programa e com isso gerar um gráfico que de fato ilustrou um comportamento assintótico compatível com a análise feita.