Trabalho Prático I - Algoritmos I

Gabriel Victor Carvalho Rocha

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

gabrielcarvalho@dcc.ufmg.br - 2018054907

1. Introdução

O problema abordado foi a implementação de um grafo direcionado não ponderado, possuindo V vértices e A arestas. A modelação consiste em que cada vértice é um membro e cada aresta (X, Y) corresponde a uma relação de comando: X comanda Y e necessariamente não pode haver a aresta (Y, X): Y comanda X, ou seja, a relação é assimétrica. Após a modelação, foi realizada a implementação das três principais instruções:

- Swap: Comando que troca a direção de uma aresta, caso ela exista e não forme um ciclo.
- Commander: Comando que retorna a idade da pessoa mais nova que comanda diretamente ou indiretamente um membro.
- Meeting: Comando que retorna uma ordem de fala dos membros respeitando a hierarquia.

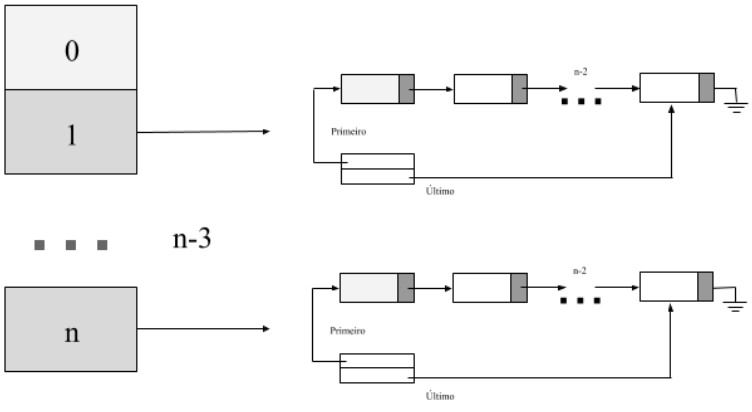
2. Implementação

Todo o programa foi desenvolvido na linguagem C, utilizando o compilador GCC (GNU Compiler Collection).

2.1. Estruturas de dados

A estrutura de dados utilizada foi um vetor de listas encadeadas, onde cada lista possui uma "célula cabeça" que corresponde ao vértice que comanda os demais vértices da lista de adjacência e cada célula possui o id e a idade do membro.

Para facilitar o uso e por possuir maior compatibilidade com o problema proposto, foram ignoradas todas as posições 0 do programa, começando então sempre pela posição 1:



2.2. Algoritmos utilizados

As principais funções do programa são: adicionaRelacao, swap, DFS/DFSRec, meeting/meetingDFSRec, commander/commanderDFSRec. As funções swap, meeting e commander, foram implementadas utilizando o caminhamento DFS (*Depth First Search*) com alterações que satisfazem o que era pedido.

A função adicionaRelacao é responsável por colocar as relações que estão no arquivo para as listas de adjacências, sendo essa uma das primeiras etapas do código em conjunto com a fazGrafo, que inicia a estrutura de dados em cada posição do vetor, e a inserePrimeira, que coloca o id e a idade de cada vértice na célula cabeça da lista da sua exata posição do vetor (que corresponde ao id).

Já a função swap, recebe como parâmetro os id's dos vértices A e B e verifica se existe relação entre os dois valores. Caso haja a relação A comanda B, é realizada a troca da relação através do uso da função removeVertice, que irá excluir o B da lista de A, e depois adicionar A na lista de B através da função adicionaRelacao. Então, é verificado se existe algum ciclo fazendo-se o uso da função DFS, que irá percorrer todo o grafo recursivamente pelo DFSRec, onde irá sempre procurar por vértices que possuam como status 'C', caso possua, isto significa que o vértice encontrado já era um vértice que estava sendo visitado e não havia terminado, retornando algum valor maior do que 0, caso não haja ciclos o seu retorno é 0. De volta para o swap, se não houver ciclos, a função então irá imprimir "S T", ou seja, a troca foi bem sucedida, caso contrário, são desfeitas as trocas anteriores e é impresso "S N", troca mal sucedida. O mesmo vale se houver a relação B comanda A, realizando as mesmas operações mudando apenas a ordem dos parâmetros. E por fim, se não houver aresta entre A e B, é retornado "S N".

Meeting é uma função que simplesmente realiza um caminhamento DFS, porém é criado um vetor para armazenar a ordem de fala. Percorrendo o grafo recursivamente através da função meetingDFSRec, ao terminar os vértices (colocando 'P' no status), é inserido o valor do id do vértice que se encontra de trás pra frente no vetor ordemFala (pois precisamos dos vértices com maiores tempos de término, então a inserção é feita ao contrário). No final é impresso "M " + o vetor ordemFala.

E por fim, a função commander, mais uma vez um caminhamento DFS, mas dessa vez é criado um grafo transposto do original, para mudar o problema de "Quem é o membro mais novo que comanda diretamente ou indiretamente o membro B" para "Quem é o membro mais novo que é comandado diretamente ou indiretamente pelo membro B", facilitando então o uso da DFS. A função percorre toda a lista de adjacência do membro passado como parâmetro, tendo uma variável menorIdade (inicializada com 101, uma idade maior do que a maior idade permitida) que sempre irá checar se o membro que está visitando possui idade menor do que já se encontra nela e se o membro é diferente do passado, mudando então seu valor quando ambas as condições forem verdadeiras. Ao final, caso a variável menorIdade continuar com o valor 101, isso significa que ninguém o comanda (ou pro caso transposto, não comanda ninguém), imprimindo "C *", caso contrário, é impresso "C " + o valor armazenado na variável.

3. Instruções de compilação, execução e formato de entrada/saída

A compilação é feita através de um arquivo Makefile, digitando “make” em um terminal Linux é gerado o executável “tp1”. Já a execução é realizada da seguinte forma:

./tp1 nomedoarquivo.txt

4. Análise de complexidade

Na complexidade em relação ao tempo, iremos analisar o código principal main e assim percorremos todas as funções.

Primeiramente, definindo V como o número total de vértices (membros da equipe) e A o número total de arestas (relação de comandar), o arquivo é lido e é alocado um vetor TipoGrafo com custo de espaço O(V) e construindo o grafo através das funções fazGrafo que possui custo O(1), inserePrimeira que adiciona os valores id e idade para cada célula cabeça possuindo custo O(1) e adicionaRelacao que possui também custo O(1), entretanto cada uma dessas funções estão em um loop que rodam V ou A vezes, tendo então complexidade de tempo O(V + A). Já o espaço, temos que armazenar os V vértices (o vetor criado) e as A arestas (criadas por adicionaRelacao), sendo então O(V + A).

Após a construção, partimos para cada uma das 3 principais funções:

Swap: Irá apenas checar a existência de uma relação, percorrendo a lista de adjacência, tendo então essa checagem no pior caso O(V), após isso, é removido o membro, que também possui custo de O(V) pois é necessário percorrer toda a lista de adjacência até achar o membro desejado. Swap também utiliza a função DFS para checar se a alteração produziu ciclos, então, é um simples DFS que retorna 0 caso não haja ciclos e qualquer valor maior do que 0 caso exista, possuindo complexidade O(V + A). Logo, somando tudo, temos a complexidade de tempo O(V + A). Em espaço, é utilizado apenas um vetor de tamanho V para armazenar o status dos vértices (não visitado, visitando ou visitado), possuindo complexidade O(V).

Commander: Uma outra adaptação de DFS. Entretanto, criamos outro vetor do TipoGrafo, tendo custo de espaço O(V), além disso, criamos todas as relações transpostas do grafo original nesse vetor criado com a função fazGrafoTransposto, que irá ter custo de espaço O(A), e em relação ao tempo, irá percorrer todas as listas de adjacência, tendo custo O(V + A). Temos também o vetor de status, que possui custo de espaço O(V). Finalmente, chamamos a função commanderDFSRec, que irá realizar o caminhamento DFS de fato, mas partindo do vértice passado como parâmetro, contudo, esse caminhamento no pior caso continua sendo O(V + A). A complexidade de tempo da função então é O(V + A) e a complexidade de espaço O(V + V + A) que é igual a O(V + A).

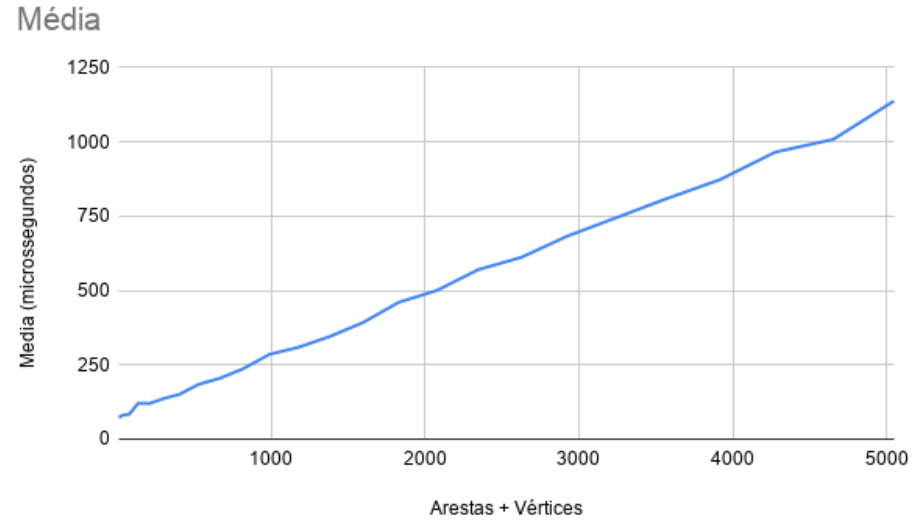
Meeting: Por fim, a função meeting que retorna a ordem de fala dos membros da equipe. Essa função também é uma adaptação do DFS, que irá guardar em um vetor de tamanho V a ordem topológica. Como temos o vetor de status também, temos a complexidade espaço O(V + V) que resulta em simplesmente O(V). Em questão de tempo, temos a mesma complexidade de antes, O(V + A) do caminhamento DFS.

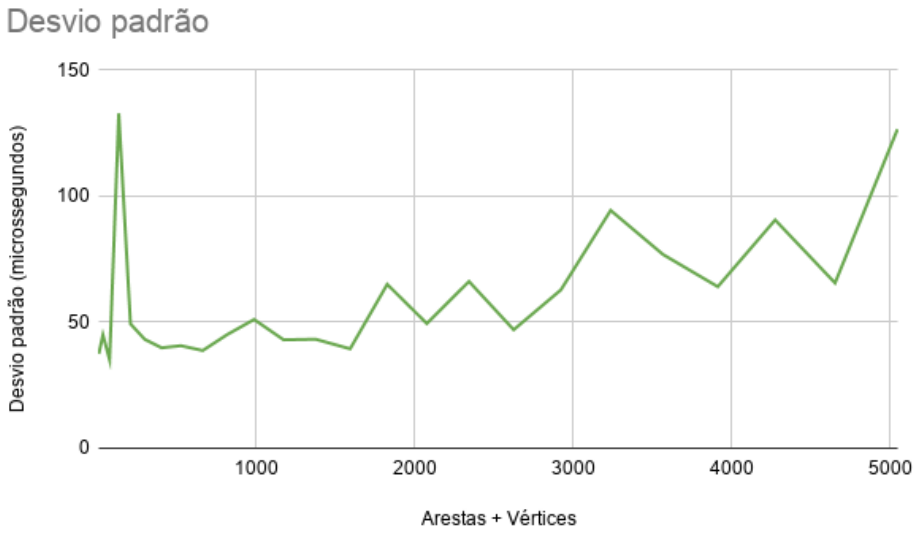
Outras funções: A função removeGrafo irá simplesmente dar free em todas as listas, precisando percorrer todas as listas de adjacência, tendo complexidade de tempo O(V + A) e de espaço O(1). Já a função vazia, apenas checa se o apontador Primeiro e Ultimo apontam pro mesmo lugar, sendo usada nas funções de caminhamento DFS, custo O(1).

Portanto, a complexidade total do programa em relação ao tempo possui custo O(V+A) e da mesma forma, em relação ao espaço possui custo O(V + A).

5. Análise experimental

Para realização do experimento, foram feitos 25 grafos diferentes com vértices variando de 4 a 100 de 4 em 4, possuindo a maior quantidade de arestas possíveis (ou seja, V*(V-1)/2 arestas) tendo então o tamanho V + A = V + (V*(V-1)/2). Isso nos dá os valores 10, 36, 78, 136, …, 5050. Cada um dos 25 grafos foi testado 20 vezes, pegando-se a média do tempo e o desvio padrão em microssegundos. O resultado se encontra nos gráficos abaixos:





Pelo gráfico de Média, podemos certificar que a análise feita anteriormente está correta, pois o crescimento da reta está linear em relação à quantidade de Arestas + Vértices.

6. Perguntas

1. Por que o grafo tem que ser dirigido?

O grafo ser dirigido serve para que haja uma hierarquia entre os membros, pois se um membro X comanda um membro Y, necessariamente temos que o Y não pode comandar X.

2. O grafo pode ter ciclos?

Não. Se houver ciclo, a ordem topológica é quebrada, pois haverá membros comandando diretamente ou indiretamente membros que o comanda.

3. O grafo pode ser uma árvore? O grafo necessariamente é uma árvore?

Árvore é um grafo não direcionado, conectado e acíclico. Como o grafo do problema é direcionado e nem todo vértice é alcançável a partir de outro, então pode-se concluir que o grafo não pode ser uma árvore.

Porém, se considerarmos as Poliárvores (*Polytree* ou *Directed Tree*) como árvores, então sim, o grafo pode e necessariamente é uma árvore pois todos os grafos serão uma DAG.

7. Bibliografia

KLEINBERG, Jon.; TARDOS, Éva. Algorithm Design.

ALMEIDA, Jussara.; Grafos - caminhamento. PDF disponibilizado via Moodle UFMG.