TP1 - ALGORITMOS I

Universidade Federal de Minas Gerais

Luiz Felipe Couto Gontijo - 2018054524

Objetivo

O objetivo do trabalho consiste em realizar a modelagem e implementação de algoritmos e soluções dado o problema do grupo de BlackJack da UFMG. Mais especificamente, modelar as informações em um problema inicial de grafos, além de desenvolver algoritmos que retornem corretamente os dados pedidos a partir de uma instância do problema e de entradas previamente estabelecidas. Também é pedido que o algoritmo implementado seja da ordem de complexidade O(A + V), onde A é o número de arestas do grafo modelado e V o número de vértices.

Ambiente de desenvolvimento e Compilação

O projeto foi desenvolvido em C++.

Para compilar, basta rodar o comando *make* no diretório onde se encontra o arquivo *tp1.cpp.* Após gerado o executável, execute-o passando o arquivo texto de entrada como parâmetro. Segue um exemplo:

```
> make
> ./tp1 teste.txt
```

O problema

Um grupo de alunos se reúnem para formar uma equipe de BlackJack. Nessa equipe, há hierarquias, ou seja, um aluno pode comandar outro aluno. Além disso, os alunos são identificados também por sua idade, que é única em relação à todo o grupo.

A entrada do problema consiste em *n* alunos e suas respectivas idades, além de *m* inteiros do tipo '*X e Y*', indicando que o aluno *X* comanda o aluno *Y.* Seguem-se *i* linhas, cada uma descrevendo uma instrução. As instruções podem ser do tipo:

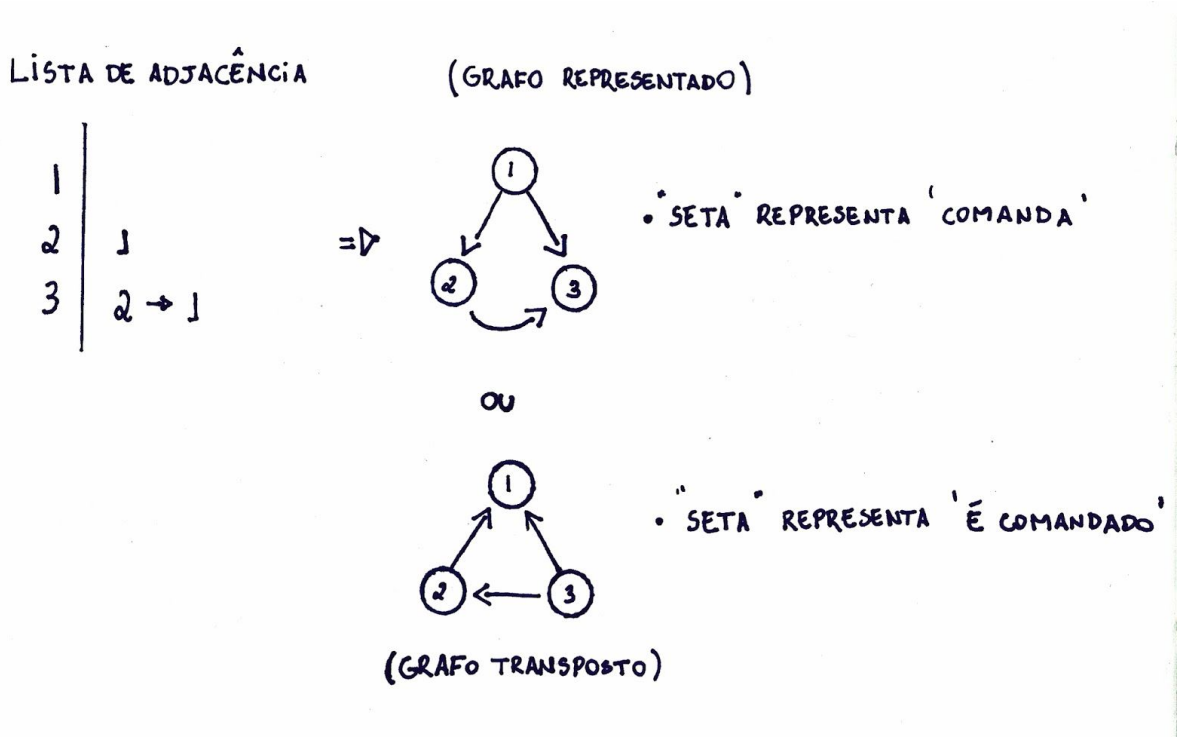
- COMMANDER: Recebe um aluno como entrada. A saída deve ser o aluno mais jovem que o comanda (direta ou indiretamente).

- SWAP: Recebe dois alunos como entrada. Verifica se existe uma aresta entre eles. Caso exista, verifica se é possível inverter a direção da aresta (ou seja, se o aluno A comandava o aluno B agora o aluno B comanda o A) , de forma a não obter ciclos no grafo. A saída deve ser 'S T' caso seja possível realizar o SWAP. Caso contrário, é impresso 'S N'.

- MEETING: Não recebe entrada. Imprime a ordem de fala na reunião da equipe. Essa ordem deve obedecer a hierarquia, ou seja, se A comanda B, A deve falar antes de B.

Implementação

Para a implementação das informações/grafo recebido como entrada, foi criada uma lista de adjacência “ao contrário”, na qual a estratégia implementada foi a de, ao invés de cada aluno B da linha de um aluno A representar que A comanda B, na verdade, representa que B comanda A. Então, para exemplificar, se temos a lista de adjacência “ao contrário” a seguir:



Essa lista de adjacência também pode ser vista como a lista de adjacência comum do grafo recebido como entrada, porém, transposto. Essa estratégia vai facilitar algumas operações que serão vistas posteriormente. Para a implementação da lista foram criadas duas classes:

- Graph: Essa classe é responsável por armazenar a lista dos alunos (a primeira coluna da lista de adjacência). Consiste em *n* ponteiros (onde *n* é o número de alunos), no qual cada um aponta para o primeiro elemento de sua respectiva *LinkedList.* Lembrando que cada elemento B da *LinkedList* do aluno A representa que B comanda A. Também são armazenados as idades de cada um dos estudantes, em um array. As funções presentes nessa

classe modelam as operações feitas no grafo, como adicionar arestas, trocar elementos etc.

- LinkedList: Essa classe funciona como um objeto de lista encadeada comum, onde cada nó armazena, além de seus dados próprios, um ponteiro para o próximo elemento da lista. As funções dessa classe passam por adicionar/remover elementos ou retornar um elemento a partir de seu índice.

As funções em particular serão melhor apresentadas quando os algoritmos para resolver cada um dos comandos recebidos como entrada forem explicados.

Por último, a função *main é* responsável por ler a entrada do arquivo texto, processá-las, e criar o objeto *Graph* que representa a instância do problema apresentada.

Criando o Grafo

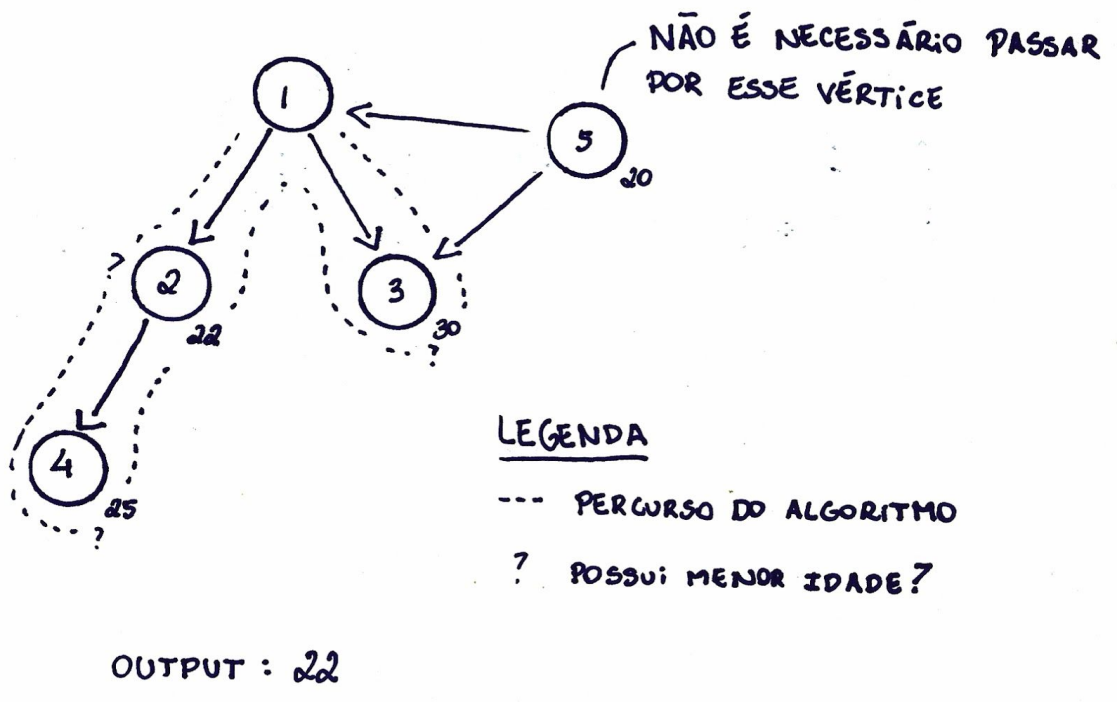
Para a criação da lista de adjacência "ao contrário", a função *main* primeiramente lê a entrada do arquivo texto e cria um array de idades (sendo que a idade do aluno 1 está na primeira posição do array, a do aluno 2 está na segunda posição, e assim por diante). Após isso, é criada a instância do objeto Graph (com a quantidade *n* de alunos e com o array de idades) e, para cada uma das *m* linhas seguintes, ele adiciona as arestas do grafo. Nesse processo, vale lembrar que se temos a linha de entrada 'X Y' (significando que X comanda Y), então X é adicionado na lista encadeada de Y. Novamente, vale pensar como a lista de adjacência do grafo transposto do grafo original de entrada.

Processando os comandos

Commander:

O objetivo é encontrar o aluno de menor idade que comanda **v.** Para isso, foi criado uma função recursiva que passa por todos os vértices que estão conectados com este vértice inicial**.** Assim, o algoritmo verifica na lista encadeada de **v** todos que o comandam**.** Seleciona o primeiro (v1) e define a menor idade encontrada até agora (já que por enquanto só passamos por 1 vértice). Então, itera recursivamente em v1 da mesma forma que em v. Após verificar todos os que derivam de v1, passa para o próximo vértice da lista encadeada de **v.** Caso algum dos vértices encontrados tenha idade menor que a menor idade encontrada até o momento, o valor da menor idade muda para a idade desse vértice. Pensando em termos de grafos, é feita uma busca em pós-ordem (que nada mais é que um tipo de DFS -

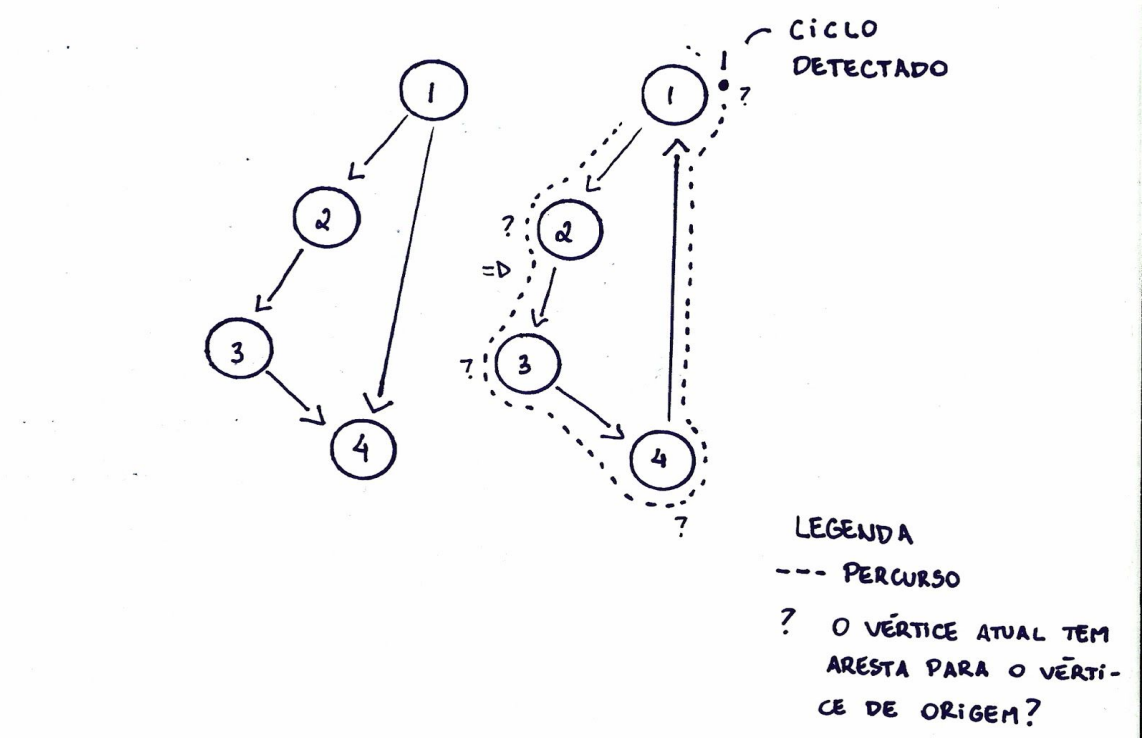
Depth First Search) na árvore que tem como raiz o vértice **v.** Lembrando que estamos analisando o grafo transposto ao grafo original, ou seja, uma aresta de A para B significa que A é comandado por B. O esquema abaixo exemplifica a execução do algoritmo.



Swap:

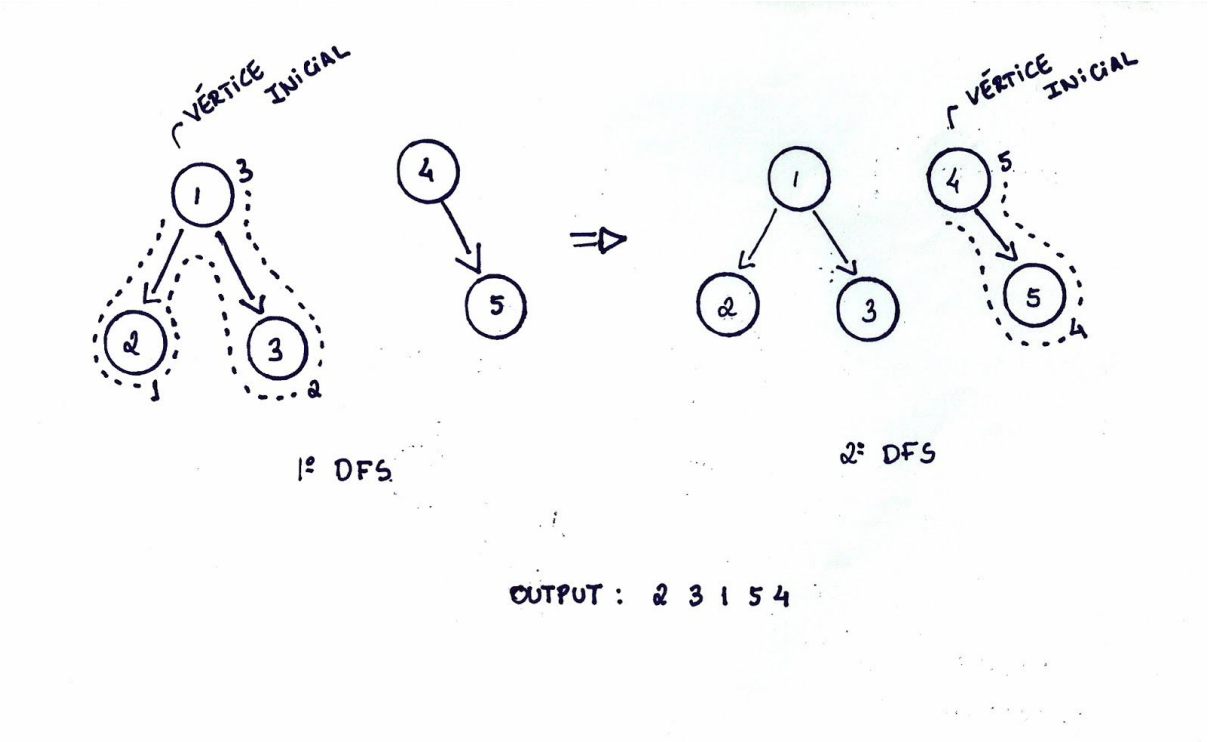
Para o comando swap, a primeira etapa é verificar se há ou não uma aresta entre os vértices A e B passados como parâmetro. Caso não haja, a função já retorna false. Caso contrário, a direção da aresta é invertida e é passada para a fase de verificação de ciclo. Para checar se um ciclo foi criado, foi implementada uma função recursiva que passa por todos os vértices que comandam direta ou indiretamente o vértice A (bastante parecido com a função commander). Porém, dessa vez, o que é verificado é se algum vértice que comanda A(direta ou indiretamente) também é comandado por A. Mais explicitamente, checamos se em algum momento, partindo do vértice A, atingimos ele próprio novamente. Caso positivo, a função detectou um ciclo, então é retornado false. Caso contrário, não há ciclos, e o swap é feito com sucesso.

Lembrando que caso não ocorra o swap, o algoritmo retorna à aresta original, ou seja, inverte novamente a aresta entre A e B. Em relação ao grafo, o que estamos fazendo também é uma Depth First Search a partir do vértice v.



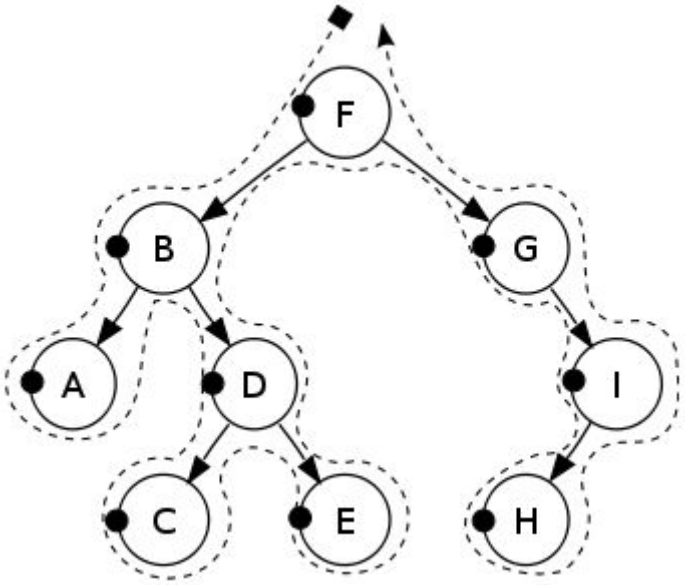
Meeting

O comando meeting, por sua vez, começa no vértice de índice 1 (o primeiro criado - mas poderia começar em qualquer um) e também executa uma DFS, porém, agora, é criada uma pilha (a partir da própria recursão da função) que só é esvaziada quando se atinge o último vértice atingível a partir desse primeiro vértice. No esvaziar da pilha, é impresso cada vértice que estava na pilha. Lembrando que a lista de adjacência "ao contrário" pode ser vista como o grafo original transposto, assim os primeiros vértices a serem impressos são aqueles que estão mais "em baixo" do grafo (nas folhas da árvore do vértice inicial), ou seja, aqueles que possuem hierarquias maiores. Durante o executar da função, também é mantido um array que possui os índices dos *n* alunos. Cada vez que um vértice é impresso, é trocado o valor de seu índice nesse array por -1. Isso é feito pois, além de evitar duplas impressões, podem existir instâncias do problema(e provavelmente há) em que vértices não estão conectados com o primeiro vértice (então não são atingíveis a partir dele). Assim, é verificado se ainda existe algum vértice que não foi impresso. Caso positivo, o algoritmo da DFS é executado novamente para esse vértice. Isso é feito até que todos os vértices tenham sido impressos.



Complexidade

A complexidade do algoritmo como um todo pode ser avaliada a partir da complexidade das operações citadas acima. Em todas elas, fazemos uma busca em pós-ordem pela árvore do vértice em que iniciamos tal busca, com o objetivo de passar por todos os vértices conectados à este primeiro.



Busca em pós-ordem (Imagens - Google)

A busca em pós-ordem nada mais é que um tipo de DFS (Depth First Search), que possui complexidade O(A+V) . A seguir temos algumas avaliações mais específicas para cada função:

- No caso do meeting, temos também um array de inteiros sendo percorrido (para que se verifique se o vértice já foi visitado ou não) com tamanho igual ao número de alunos, portanto O(V) para a complexidade de espaço e para a complexidade de tempo. Então sua complexidade final é O(V) + O(V+A) = O(V+A).

- No caso do commander, são feitas comparações para verificar se o vértice encontrado possui idade menor do que a menor idade até aquele momento. Portanto, dado um vértice V, no pior caso, temos que ele está conectado com todos os outros alunos (lembrando que estamos avaliando o grafo transposto ao original), e ,então, são feitas V-1 comparações. Além, é claro, da DFS. Portanto, a complexidade final é O(V-1) + O(V+A) = O(V+A), lembrando que A nunca é negativo.

- Já no swap, no pior caso (que é quando há a aresta entre A e B porém foi formado um ciclo) sempre são feitas 4 operações de troca de aresta (2 de adicionar e 2 de remover). Como esse custo é constante, O(1). Porém, para encontrar se a aresta B está na lista encadeada de A, temos o custo de, no máximo, o grau de A, que é, no pior caso, O(V-1). No final, e lembrando de considerar a DFS, tem-se a complexidade de O(1) + O(V-1) + O(V+A) = O(V+A).

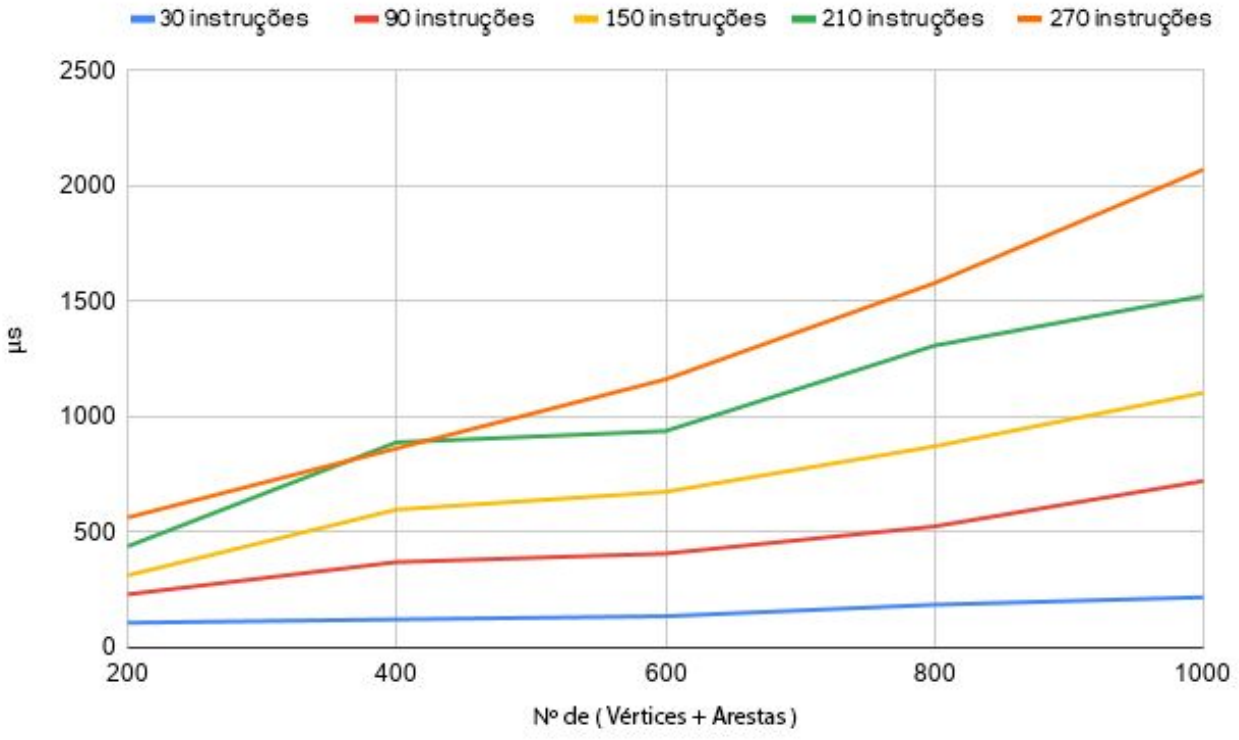
Considerando todas as complexidades citadas acima, é possível concluir que a complexidade do algoritmo criado, como um todo, equivale a O(V+A). Os testes realizados a seguir comprovam tal ordem de complexidade.

Testes

Para a realização dos testes, foram gerados automaticamente casos de testes que englobam tamanhos diferentes de (número de alunos + número de arestas) e, para cada um deles, várias quantidades diferentes de comandos. Para cada caso, foram feitas 20 execuções independentes e, então, calculado o tempo médio de execução. Também foi calculado o desvio padrão médio para cada uma das instâncias de (alunos+relações). O objetivo aqui é comprovar a natureza linear do algoritmo, pensando que, quando há uma alteração no (número de alunos + número de arestas), ocorre uma variação proporcional à esta no que diz respeito ao tempo necessário para se concluir a execução do algoritmo. Segue a tabela com os dados coletados:

	Nº de alunos+arestas	200	400	600	800	1000
Nº Comandos	-	-	-	-	-	-
30	-	108,3	122,8	136,6	186,3	218,65
90	-	231,25	371	407,85	525,45	722,35
150	-	313,5	598,5	675,25	872,4	1104,05
210	-	438,95	890	938,7	1308,8	1523,85
270	-	564	863,5	1162,65	1579,35	2071,6
		$\sigma_{médio}$ = 35,5	$\sigma_{médio}$ = 44,1	$\sigma_{médio}$ = 60,3	$\sigma_{médio}$ = 50,1	$\sigma_{médio}$ = 70,5

O tempo calculado está em microssegundos. Já a partir da tabela, é possível ter uma intuição da linearidade, já que, quando o número de (alunos+arestas) dobra, o tempo de execução tende a dobrar também. Vale ressaltar que para pequenos números de comandos e de (alunos+arestas) - que é o caso da primeira linha da tabela - a variação no tempo de execução é bastante pequena, mas, ainda assim, perceptível. A partir do gráfico abaixo, fica evidente a tendência à linearidade quando o número de comandos cresce.



Portanto, a partir dos dados coletados e da visualização do gráfico, é plausível concluir que a complexidade do algoritmo condiz com as previsões feitas anteriormente. Sendo, assim, o tempo de execução cresce proporcionalmente em relação ao número de alunos somado com o número de relações (comandante-comandado) entre eles. A complexidade comprovada é, por consequência, O(V+A).

Questões

O grafo deve ser dirigido?

Sim, o grafo deve ser dirigido pois a relação de comandante-comandado não é mútua, ou seja, a direção da aresta deve indicar se o aluno A comanda o aluno B **ou** se o aluno B comanda o aluno A, não podendo ocorrer os dois ao mesmo tempo.

O grafo pode ter ciclos?

Não, pois caso houvesse ciclos, não seria possível identificar a hierarquia da instância do problema. Por exemplo, caso A comandasse B e B comandasse A (mesmo que indiretamente) não conseguiríamos distinguir a hierarquia entre A e B e comandos de commander e meeting não seriam aplicáveis.

O grafo pode ser uma árvore? Ele necessariamente é uma árvore?

Sim, o grafo pode ser uma árvore. Nesse caso, todos os vértices estariam conectados a partir de um vértice raíz, sendo aquele com maior hierarquia. Quanto mais próximo das folhas, menor seria a hierarquia. Porém, as instâncias do problema não precisam representar necessariamente uma árvore, já que podem existir instâncias em que nem todos os vértices estão conectados a partir de um primeiro. Nesse caso, teríamos uma floresta, ou seja, um conjunto de árvores.

Referências para a realização do trabalho e Bibliografia

Algorithm Design, by Jon Kleinberg, Eva Tardos & Iva Tardos.

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática-3a. Edição, Elsevier, 2012.