Trabalho Prático 1
Decifrando os Segredos de Arendele

Jean George Alves Evangelista - 2018047021

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

jeanalves@dcc.ufmg.br

1. Introdução

O objetivo deste trabalho é desenvolver um programa na linguagem C/C++ que realiza operações de SWAP, COMMANDER e MEETING tendo como base o problema descrito no enunciado, relacionado aos alunos da UFMG e o jogo Blackjack. Um grafo (V,A) deve ser modelado de tal forma que os vértices representem os alunos e uma aresta (u,v) existe somente se o aluno u comanda o aluno v.

As três operações são:

- SWAP(S): caso exista uma aresta entre os alunos A e B, troca a ordem de comando "A comanda B" para "B comanda A", desde que isso não origine um ciclo no grafo.
- COMMANDER(C): Recebe como entrada um aluno e retorna o aluno mais novo que comanda A (diretamente ou indiretamente).
- MEETING(M): Apresenta uma ordem de fala na reunião. Essa ordem deve respeitar a hierarquia, caso A comande B diretamente ou indiretamente B não pode falar antes de A.

O programa deve primeiramente ler três números inteiros, representando o número de alunos (vértices), o número de relações diretas entre os alunos (arestas) e o número de instruções a serem realizadas (SWAP, COMMANDER ou MEETING). Em seguida programa deverá ler as idades dos alunos, as relações e os comandos.

Os seguintes aspectos devem ser respeitados: não existir alunos com mesma idade, não existir circularidade (se A comanda B, B não pode comandar A) e as entradas e saídas devem ser de acordo com o padrão especificado.

2. Implementação

Optou-se por realizar a implementação na linguagem de programação C++.. O programa possui três classes: *Node*, *Edge* e *Graph*.

2.1 Classes

Node: Uma classe bastante simples que representa cada nó (ou vértice) do grafo. Possui como atributos dois inteiros: um id para identificar cada estudante e a idade do estudante.

Edge: Classe que representa uma aresta no grafo. Possui como atributos dois objetos do tipo *Node*, que são o vértice de origem e o vértice de destino.

Graph: Classe principal do programa. Trata-se do grafo que será utilizado para representar a estrutura especificada, bem como três funções principais. Possui como atributos o número de vértices do grafo (redundante mas bastante útil na implementação das funções), uma lista de vértices do grafo e um array de listas do grafo representando a lista de adjacência (essas duas listas também ficaram redundantes mas facilitaram na implementação). A classe *Graph* possui inúmeros métodos, que serão detalhados abaixo.

2.2 Estrutura de Dados

Tal como especificado, a estrutura de dados principal do trabalho é um grafo direcionado, acíclico e conectado. A figura abaixo mostra o grafo resultado de um dos testes realizados.

- As idades dos alunos estão omitidas, mas foram recebidas nessa ordem na entrada: 21 33 34 18 42 22 26 42.
- A relações de comando são (leia-se a comanda b em (a,b)): (1,2), (1,3), (2,5), (3,5), (3,6) e (4,6)

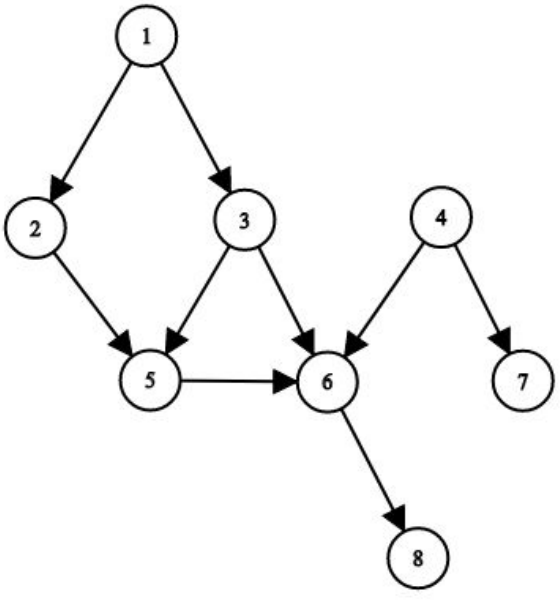


Figura 1. Exemplo de grafo com 8 vértices, utilizado nos testes

Outras estruturas presentes são somente as listas (*vector* da *standard template library*) e arrays simples.

2.3 Descrição da execução, dos principais métodos e das escolhas de implementação

Como todo programa desenvolvido em C++, a execução inicia-se no arquivo *main.cpp*. Na função *main* a primeira coisa feita é tentar abrir o arquivo (cujo nome foi recebido como parâmetro na execução), caso o arquivo não seja aberto por qualquer motivo o programa se encerra. Em seguida, inicia-se o processo de leitura do arquivo conforme especificado: obtêm-se o número de vértices, arestas e instruções. Dois laços do tipo *for* são executados para realizar a leitura das idades e das relações.

É instanciado então um objeto do tipo *Graph*, passando para seu construtor uma lista de vértices (montada no primeiro laço *for*), uma lista de arestas (montado no segundo laço *for*), o número de vértices e o número de arestas. No construtor o grafo é montado por meio da lista de adjacência (um array de listas do tipo inteiro). Optou-se pela implementação com lista de adjacência para manter a complexidade O (V+A) nas operações de consulta, pesquisa, etc.

Após montar o grafo, na *main* é executado um terceiro laço *for*, para ler as instruções do arquivo. Após ler o primeiro caracter a execução varia:

- caso o caracter lido seja ‘S’ significa que é um comando de SWAP, dois inteiros que representam os identificadores dos alunos que deve-se trocar a ordem de comando são lidos, o método *swap* da classe *Graph* é chamado e imprime-se “S T” caso a troca de comando tenha sido realizada e “S N” caso não tenha sido.
- caso o caracter lido seja ‘C’ temos uma instrução de COMMANDER, é lido o identificador do aluno que deseja-se saber o comandante mais novo. O método *commander* do objeto do tipo *Graph* é chamado, passando-se o id do estudante. Caso não tenha comandante é mostrado “C *” na tela e caso tenha é mostrado “C” + a idade do comandante mais novo.
- caso o caracter lido seja ‘M’ temos a instrução de MEETING. O método *meeting* é chamado e ele mostra uma possível ordem de fala.

2.3.1 SWAP(S)

O comando SWAP deve trocar a ordem de comando entre dois vértices a e b. Isso deve ser feito se, e somente se, a comandar b diretamente e a troca não acarretar num ciclo no grafo.

O método *swap* recebe dois inteiros a,b representando os identificadores dos alunos. Primeiramente é verificado se a comanda b diretamente por meio do método *directCommand*, que percorre a lista de adjacência na posição a e verifica se b está presente ali. Caso não haja comando direto a função *swap* já retorna *false*. Caso tenha comando direto a relação é alterada de a->b para b->a na função *swapCommand*, que

remove b da posição a da lista de adjacência e adiciona a na posição b. Após trocar o comando, é verificado se o grafo deixou de ser acíclico no método *hasCycle*. Essa verificação é feita de maneira recursiva e bem semelhante a uma busca em profundidade (DFS), de tal modo que os adjacentes de um vértices são visitados e os adjacentes de cada adjacente e assim sucessivamente. Dois arrays do tipo booleano são utilizados para controle dos vértices já visitados ou não. Caso o grafo possua um ciclo, a troca é desfeita (b -> a vira a -> b novamente) e o método *swap* retorna *false*, caso contrário é retornado *true*.

2.3.2 COMMANDER (C)

O COMMANDER deve retornar a idade do comandante mais novo de um vértice a.
O método *commander* recebe um inteiro a, representando o identificador de um estudante. A primeira coisa feita dentro deste método é transpor (inverter todos os comandos) o grafo atual por meio do método *transpose*, que basicamente cria uma nova lista de adjacência, tomando o cuidado de mudar toda relação do tipo a -> b para b -> a. Após transpor o grafo é feito uma busca em largura (BFS) a partir de a no método *connected*. Como o grafo foi transposto, obter todos os comandados de a equivale a obter todos os seus comandantes (originalmente). A implementação da busca em largura é muito parecida com a vista em sala de aula, utilizando um array de valores booleanos para saber quais vértices foram visitados e uma espécie de fila para armazenar os vértices. O método *connected* retorna para o *commander* uma lista de vértices que são comandados por a. Caso a lista esteja vazia, significa que não existem comandantes, retorna-se -1 para a *main*. Caso a lista não esteja vazia, ela é percorrida e a menor idade é buscada e retornada para a *main*.

2.3.3 MEETING (M)

O MEETING deve apresentar uma possível ordem de fala na reunião dos membros do time, respeitando a hierarquia.
O método *meeting* imprime a ordem de fala utilizando da ordem topológica do grafo. A ordem topológica é obtida de maneira similar a uma busca em profundidade (DFS). Uma das diferenças é a utilização de uma espécie de pilha para armazenar os vértices na ordem.

3. Instruções de Compilação de Execução

Compile com make e rode com ./tp1 <nomedoarquivo>.

4. Análise de Complexidade

Será utilizado V para abreviar “número de vértices” e A para abreviar “número de arestas”.

4.1 Complexidade de Espaço

Inicialmente temos uma complexidade de espaço O(V+A), por representar o grafo utilizando lista de adjacência. Resta saber se em algum lugar do código é utilizado mais espaço do que V+A. Ao analisarmos cada um dos métodos do programa vemos que em nenhum momento isso ocorre, pois toda a memória alocada é igual ou inferior a V+A. Temos então que a complexidade de espaço do programa é O(V+A).

4.1 Complexidade de Tempo

A princípio também temos uma complexidade de tempo O(V+A), por utilizar de lista de adjacência. Porém vamos avaliar cada método principal individualmente para garantir isso:

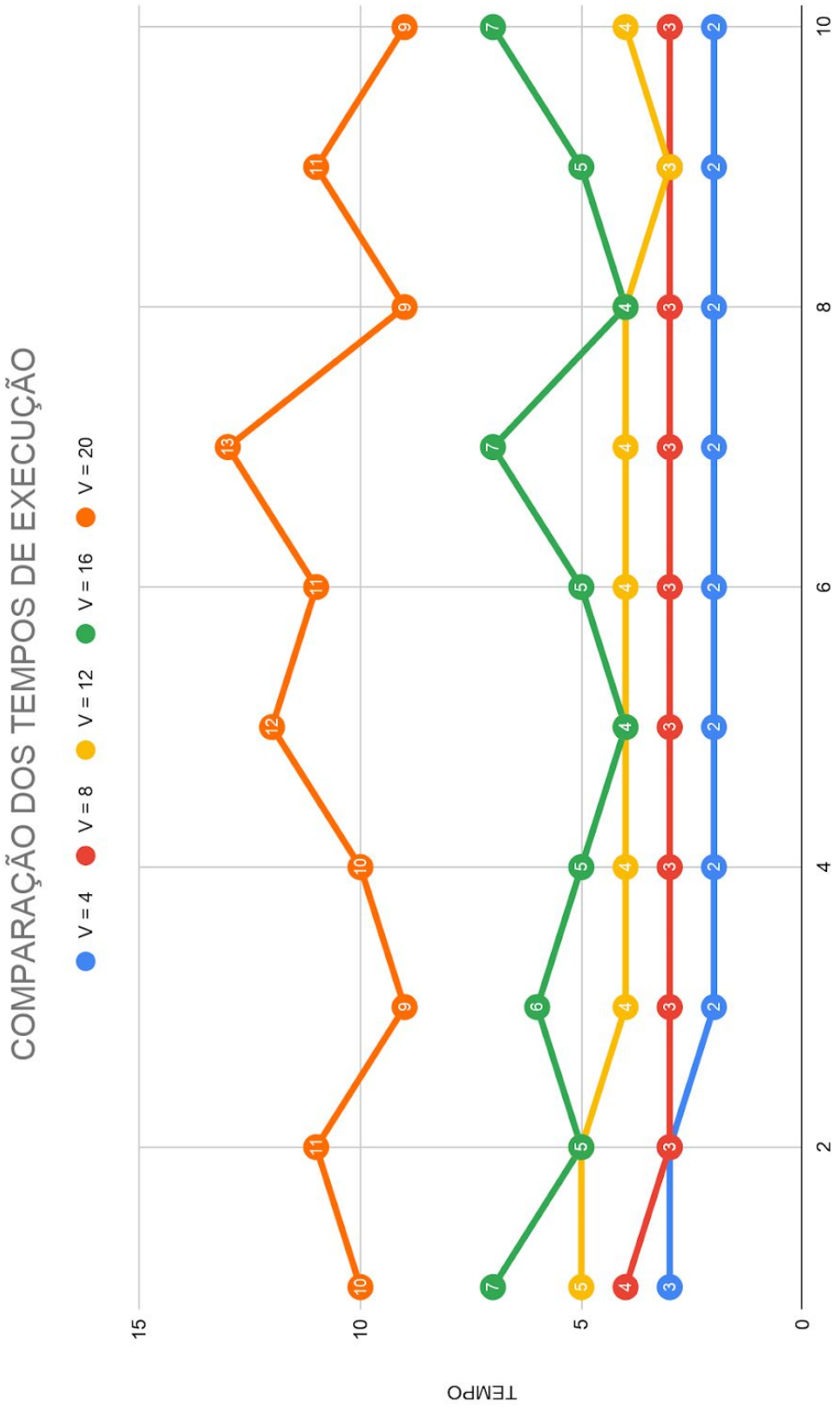
- *swap*
    - *directComand*: A complexidade será exatamente o número de filhos/ comandados de a, portanto nunca excederá o número de vértices, sendo O(V).
    - *swapCommand*: Mesma avaliação da função anterior, percorre os comandados de a do início ao fim, O(V). Esse comando pode ser executado até duas vezes no pior dos casos (2V), mas ainda assim O(2V) = O(V).
    - *hasCycle*: Trata-se de uma busca em profundidade com algumas modificações. Complexidade O(V+A)
    - Total: 3V + A = O(V+A)
- commander
    - *transpose*: Cria uma nova lista de adjacência, portanto complexidade O(V+A). É executado 2 vezes mas ainda assim é O(V+A).
    - *connectedNodes*: Busca em largura, complexidade O(V+A).
    - buscar o menor das idades: O(V), no pior dos casos um vértice será pai de todos os outros.
    - Total: Teremos algo como 3V + 2A ou 4V + 2A, o que ainda é O(V+A).
- meeting: o meeting se resume a uma busca em profundidade com algumas modificações para obter a ordem topológica (O(V+A)) e em percorrer os vértices e printar (O(V)). Portanto a complexidade total será O(V+A).

Assim concluímos que o programa nunca excederá a complexidade de tempo igual a O(V+A).

5. Avaliação Experimental

Foram realizados testes com grafos com quantidades diferentes de vértices (4, 8, 12, 16 e 20). Para cada grafo, mediu-se o tempo de execução dez vezes. No fim calculou-se a média do tempo e o desvio-padrão do tempo. O gráfico abaixo mostra os resultados em cada medição de cada grafo.

Gráfico 1. Tempos obtidos ao executar cada teste inúmeras vezes.



A tabela abaixo apresenta os tamanhos, relações e instruções (em quantidades) e também a média do tempo de execução e o desvio padrão do tempo de execução. Não foram realizadas medições de tempo para grafos sem vértices, arestas e instruções.

Abreviações:

- V: número de vértices
- A: número de arestas
- I: número de instruções
- S: número de swaps
- C: número de commands
- M: número de meetings
- MED: média do tempo de execução (em microssegundos)
- DP: desvio padrão do tempo de execução (em microssegundos)

Tabela 1. Quantidades das entradas nos testes realizados.

V	A	I	S	M	C	MED	DP
0	0	0	0	0	0	-	-
4	5	3	1	1	2	2,20	0,42
8	9	2	2	2	2	3,10	0,32
12	13	10	4	4	2	4,10	0,57
16	17	14	6	6	2	5,50	1,18
20	20	18	8	8	2	9,00	1,35

Analisando a tabela, vemos que o tempo de execução tende a aumentar à medida que o número de vértices, arestas e instruções cresce. Esse era um resultado previsível, uma vez que geralmente um problema com mais dados exige mais tempo para ser solucionado. É importante observarmos que ao dobrar, tripi JEAN
Também é possível observar que o tempo é de certa forma proporcional ao número de vértices e arestas, o que era esperado, tendo em vista que o algoritmo tem complexidade tempo O(V+A).

5.1 Questões levantadas na especificação

- Por que o grafo tem que ser dirigido?
    - Porque não reciprocidade nas relações de comando. O enunciado deixa bem explícito que se A comanda B, o contrário não pode ser válido (B não pode comandar A diretamente ou indiretamente).
- O grafo pode ter ciclos?

    - Não. Se houver algum ciclo no grafo um estudante estará comandando outro que o comanda e isso não é permitido.
- O pode ser uma árvore? O grafo necessariamente é uma árvore?
    - O grafo pode sim ser uma árvore, caso um estudante seja comandado e comande apenas um outro estudante. Mas o grafo não necessariamente é uma árvore, pois pode acontecer de um estudante ter mais de um comandante e esses comandantes não serem comandados por ninguém. Teríamos uma árvore com mais de uma raiz.

6. Conclusão

Conclui-se que foi possível implementar um programa funcional que realize as três operações especificadas na complexidade adequada. Este trabalho mostrou-se bastante proveitoso, permitindo exercitar e praticar os conceitos vistos em sala de aula.

Referências

StackOverflow. Disponível em: <https://stackoverflow.com/>. Acesso em: 26 set. 2019.

C++ Language. Disponível em: <http://www.cplusplus.com/doc/tutorial/>. Acesso em: 26 set. 2019.

Geeks for Geeks. Disponível em: <https://www.geeksforgeeks.org/selection-sort/>. Acesso em: 26 set. 2019.

Lecture Slides for Algorithm Design by Jon Kleinberg And Éva Tardos. Princeton Univeristy. Disponível em: <https://www.cs.princeton.edu/~wayne/kleinberg-tardos/>. Acesso em: 26 set. 2019.

Ziviani, N. (2006). *Projetos de Algoritmos com Implementações em Java e C++: Capítulo 7: Algoritmos em Grafos*. Editora Cengage.