UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ALGORITMOS I - TP1

Breno Poggiali de Sousa
2018054800

September 23, 2019

Sumário

1 Introdução

O objetivo desse trabalho é desenvolver um programa utilizando um grafo e aplicar sobre esse grafo algoritmos aprendidados em sala de aula da forma mais eficiente possível.

Todas as operações que iremos fazer sobre o grafo, devem funcionar em tempo de O(V+A), onde V é o número de vértices e A o número de arestas. Dessa forma, utilizaremos no desenvolvimento desse trabalho dois algoritmos principais e muito conhecidos: BFS (Breadth First Search) e DFS (Depth First Search), que são dois algoritmos que rodam em O(V+A).

Além disso, cadastraremos o nosso grafo em uma lista de adjascência para garantir esse tempo de complexidade. Montamos essa lista de adjascência em um *vector <vector <int>>*, ou seja, um vetor de vetores de inteiros. Onde cada vetor interno é um vértice e cada inteiro nesse vetor interno, é um vértice que aquele vetor aponta.

2 Sobre o Problema

Em nosso problema, iremos construir um grafo direcionado e acíclico de listas de adjascência que representa uma equipe (time) de blackjack e teremos que fazer 3 tipos de operações nesse grafo.
São elas:

SWAP (S) - verifica se existe uma aresta entre os alunos A e B. Caso a relação anterior exista, troca a direção de uma aresta que representava que o aluno A comandava o aluno B para uma que representa que o aluno B comanda o aluno A. Essa troca só pode ser realizada se o resultado não ocasionar um ciclo. Caso um ciclo ocorra, o grafo permanece sem alteração;

COMMANDER (C) - recebe como entrada um aluno A e responde a idade da pessoa mais jovem que comanda A (direta ou indireta);

MEETING (M) - identifica uma possível ordem de fala dos alunos em uma reunião. Uma ordem de fala em uma reunão é uma ordem em que ninguém nunca fale antes de quem comanda ele diretamente ou indiretamente.

3 Entrada e Saída

Para entendermos melhor sobre o desenvolvimento desse trabalho, vamos entender primeiro como são as entradas e as saídas do nosso programa.

ENTRADA

A primeira linha da entrada contém 3 inteiros N, M e I, indicando respectivamente o número de pessoas no time, o número de relações diretas entre os membros da equipe e o número de instruções que teremos que retornar uma resposta.

Além disso, cada membro da equipe da UFMG é identificado por um número de 1 a N. A segunda linha contém N inteiros Ki ($1 \leq$ Ki ≤ 100, para $1 \leq$ i $\leq$ N), onde Ki indica a idade do membro da equipe de número i.

Cada uma das M linhas seguintes contém dois inteiros X e Y ($1 \leq$ X, Y $\leq$ N, X != Y), indicando que X comanda Y diretamente. Seguem-se I linhas, cada uma descrevendo uma instrução. Uma instrução de SWAP de comando é descrita em uma linha contendo o identificador 'S' seguido de dois inteiros A e B ($1 \leq$ A, B $\leq$ N), indicando os dois membros que devem trocar seus lugares na cadeia de comando. Uma instrução COMMANDER é descrita em uma linha contendo o identificador 'C' seguido de um inteiro E ($1 \leq$ E $\leq$ N), indicandoum membro. Uma instrução MEETING é descrita em uma linha contendo apenas o identificador 'M'.

SAÍDA

Para cada instrução SWAP o programa deve imprimir 'S T' caso a troca tenha sido um sucesso. Caso contrário, se a troca não foi possı´vel o programa deve imprimir 'S N'. Para cada instrução COMMANDER o programa deve imprimir uma linha contendo o caractere 'C' e um único inteiro que corresponde a idade da pessoa mais jovem que comanda (direta ou indiretamente) o membro nomeado na pergunta. Se o aluno não é comandado por ninguém, imprima o caractere C $*$ (asterisco). Para a instrução MEETING, o programa deve imprimir uma das

alternativas possíveis de ordem de fala de todos os grupos, ou seja, a saı´da será o caractere 'M' e os N membros do grupo considerando a ordem de fala.

Exemplo de entrada:

```
7 8 8
21 33 34 18 42 22 26
1 2
1 3
2 5
3 5
3 6
4 6
4 7
6 7
C 7
S 4 2
C 3
S 3 6
C 3
S 4 7
C 4
M
```

Exemplo de Saída:

```
C 18
S N
C 21
S T
C 18
S N
C *
M 4 6 7 1 3 2 5
```

Após entender como recebemos a entrada e as saídas, podemos passar analisar a complexidade do nosso código.

4 Análise de complexidade

Para fazermos a análise de complexidade de nosso código, iremos analisar qual o custo de cada entrada que recebemos e, assim, conseguimos entender qual a complexidade do código como um todo. Interessante perceber aqui que a nossa complexidade de espaço sempre será O(V+A), pois temos um grafo salvo em uma lista de adjascência. Todas as análises abaixo, estaremos considerando portanto, a análise da complexidade de tempo.

A começar pela primeira linha, lemos 3 inteiros (N, M e I), o que possui O(1)

Na segunda linha, recebemos o array com as idades de cada participante da equipe e salvamos ele em um vector. Isso possui uma complexidade de O(N) (ou O(V)) pois, para cada participante da equipe de blackjack, precisamos ler sua idade e dar um push_back no vector ages.

A partir daí, lemos M linhas que representam as arestas direcionadas do nosso grafo. Como sempre iremos ler grafos válidos, o custo de implementação de cada aresta é O(1). Como fazemos isso, M vezes, temos O(M) ou O(A).

Por fim, podemos chamar 3 tipos de funções: SWAP, COMMANDER e MEETING. Vamos entender melhor qual a complexidade de cada uma delas:

SWAP: O comando SWAP (representado pela letra 'S') recebe dois inteiros A e B e chama 2 funções. A primeira, *existsEdge*, que possui complexidade O(V) para encontrar se existe uma aresta de A para B. Caso retorne falso (não existe aresta A-¿B), chamamos a mesma função novamente invertendo A e B para checar se existe aresta de B para A.

Caso não exista nenhuma aresta de A-¿B nem de B-¿A, a função retorna false e imprimimos "S N" na tela do usuário. Caso exista, chamamos a função *canSwap*.

Essa função inverte a relação entre A e B (O(V), pois tenho que achar a posição da relação de A para B, deletá-la, para a partir daí criar a relação de B para A). Após inverter, chamamos a função *existsCycle* que se trata de um BFS adaptado (adaptado pois ele retorna um booleano) que retorna se existe um ciclo ou não em nosso grafo. Nessa função, temos a mesma complexidade de um BFS com um grafo em lista de adjascência, que é O(V+A). Não existindo o ciclo, realizamos a troca permanentemente (O(V)).

Dessa forma, o comando SWAP, representado pela função *getSwap* possui, no pior caso, a seguinte complexidade: 3*O(V) + O(V+A) = O(V+A).

COMMANDER: O comando COMMANDER (representado pela letra 'C') recebe um único inteiro E. Após ler esse inteiro E, ele chama a função*getCommander*.

Para retornar o comandante direta ou indiretamente mais novo de E, a função *getCommander*, primeiramente, chama a função *inverseGraph* que retorna o grafo atual invertido (O(V+A), pois passo em todos os vértices e em todas as arestas invertendo essas relações e salvando isso em um outro grafo).

Após isso, checo se nesse grafo invertido, A está relacionado com alguém. Caso não esteja, isso significa que ninguem manda em A e printamos "C *". Caso exista a relação de A para alguém no grafo invertido, chamamos a função *getChildrens* que pega todos os filhos do vértice A nesse grafo invertido, ou seja, quem manda nele. A complexidade de *getChildrens* é O(V+A). Isso porque, ela se trata de uma DFS, que vou adicionando todos os filhos de A em um array childrens e retorno isso no final.

Após isso, itero sobre esse array de filhos de A e vou sobrescrevendo caso exista um mais novo e, como A pode ter no máximo V-1 filhos, temos O(V) de complexidade.

Dessa forma, temos na função *getCommander* a seguinte complexidade: 2*O(V+A) + O(V) = O(V+A).

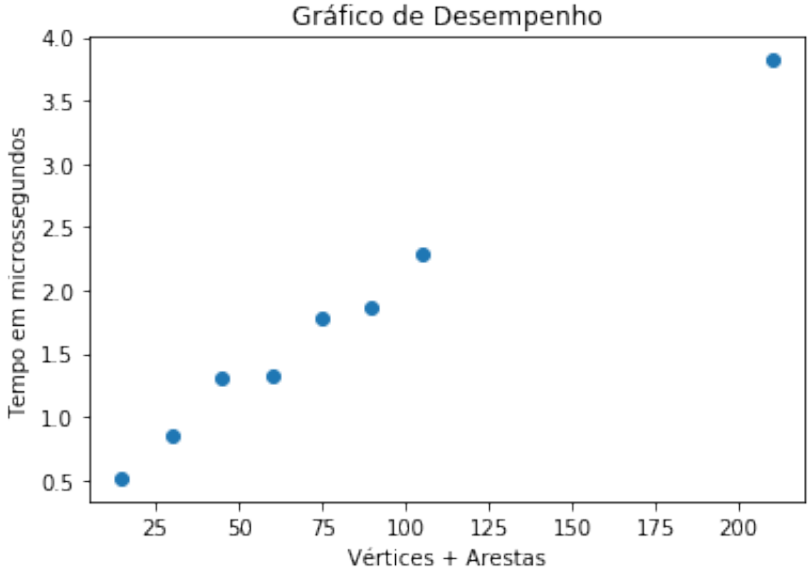
MEETING: Por fim, o comando meeting (reoresentado pela letra 'M') realiza simplesmente um topological sort que é uma adaptação da DFS recursiva. Dessa forma, a complexidade é exatamente a mesma O(V+A). Aqui eu chamei de adaptação, pois a única diferença, é que adicionamos uma pilha a mais (o que mantém a complexidade em O(V+A)). Após rodar a DFS, simplesmente desempilhamos a pilha e saímos imprimindo cada elemento, o que possui complexidade O(V).

Sendo assim, a função *getMeeting* possui a seguinte complexidade: O(V+A) + O (V) = O(V+A).

Sendo assim, como para montar o nosso grafo, possuimos complexidade O(A) e como para realizar qualquer operação nesse grafo, seja ela SWAP, COMMANDER ou MEETING, possuimos sempre a complexidade igual a O(V+A), é garantido que a complexidade de tempo do nosso programa roda dentro de O(V+A).

5 Avaliação Experimental

Para provar que o nosso algoritmo roda nessa complexidade, geramos o gráfico abaixo que evidencia que o crescimento de tempo de execução de nosso programa cresce junto com O(V+A). Ou seja, linear!



O gráfico acima, foi montado utilizando um script enviado pela monitora da disciplina para medir o tempo. Com esse script, medi o tempo de execução do meu código 10 vezes para cada tamanho. Cada tamanho de grafo foi considerado somando V+A, além disso, 3 operações foram feitas em cada um desses grafos (um Swap, um Commander e um Meeting). Após salvar o tempo, tirei a média e plotei o gráfico.

Os resultados experimentais foram:
tempo_15 = [0.341000, 0.460000, 0.585000, 0.588000, 0.456000, 0.455000, 0.430000, 0.437000, 0.453000, 0.445000, 0.451000]
tempo_30 = [0.758000, 0.770000, 0.777000, 0.813000, 0.791000, 0.833000, 0.768000, 0.834000, 0.850000, 0.765000, 0.543000]
tempo_45 = [1.272000, 1.432000, 1.262000, 1.313000, 1.449000, 1.406000, 0.898000, 1.448000, 1.277000, 1.262000]

tempo_60 = [1.470000, 1.465000, 0.994000, 1.441000, 1.288000, 1.438000, 1.341000, 0.940000, 1.468000, 1.316000]
tempo_75 = [1.616000, 0.978000, 1.831000, 1.632000, 1.838000, 1.803000, 1.503000, 1.853000, 1.420000, 2.032000, 1.338000]
tempo_90 = [2.001000, 2.183000, 1.428000, 1.560000, 1.789000, 2.188000, 1.584000, 2.168000, 1.554000, 2.171000]
tempo_105 = [2.633000, 2.794000, 1.742000, 2.499000, 1.839000, 1.961000, 2.433000, 2.078000, 2.464000, 2.447000]
tempo_210 = [5.020000, 3.005000, 4.757000, 3.104000, 4.800000, 3.086000, 4.799000, 4.800000, 2.469000, 2.385000]

6 Conclusão

Dessa forma, conseguimos criar um programa completo em C++ que satisfaz todas as exigências da equipe de BlackJack em uma complexidade muito boa (linear). Nesse programa, implementamos diversos algoritmos aprendidos em sala de aula como DFS, BFS e Topological Sorting.

Espero que a equipe esteja satisfeita! :D

Bibliografia

ALMEIDA, J. M. (2019) *Slides da disciplina de Algoritmos I*

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

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. (2009) *Introduction to Algorithms*. MIT Press.