Trabalho Prático 1 - Algoritmos 1

Victor Hugo Silva Moura - 2018054958

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

1 Introdução

O grupo de BlackJack da UFMG necessita de organizar sua equipe de forma hierárquica de modo que as responsabilidades do time possam ser subdivididas. Para evitar suspeitas, a distribuição hierárquica do grupo deve ser alterada após certo tempo. Essa distribuição, no entanto, deve seguir certas regras. Ela nunca pode gerar um ciclo (pois, dessa forma, haveria um comandado comandando um comandante) e, nas reuniões, a ordem de fala deve seguir a hierarquia, ou seja, as pessoas com hierarquias maiores devem falar antes das outras.

O objetivo do trabalho prático é, dada uma certa distribuição hierárquica (um grafo) fornecida como entrada, facilitar o gerenciamento da equipe através de comandos que checam certas propriedades da equipe (por exemplo, se é possível inverter a posição de dois membros da equipe sem gerar ciclos, que é uma regra da distribuição). O comandos necessários para o gerenciamento da equipe são:

- 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.

2 Implementação

A solução para o problema foi implementada por meio de um grafo direcionado e não ponderado G = (V, A), onde V são os vértices do grafo, que

representam os integrantes do grupo de BlackJack, e A são as arestas, que representam as relações hierárquicas entre os integrantes. Para facilitar as operações requisitadas sobre o grafo foi utilizada a representação em listas de adjacência. Sendo assim, cada integrante do grupo tem sua lista de adjacência que contém os integrantes que ele comanda. A Figura 1 mostra um exemplo de grafo e sua respectiva lista de adjacência.

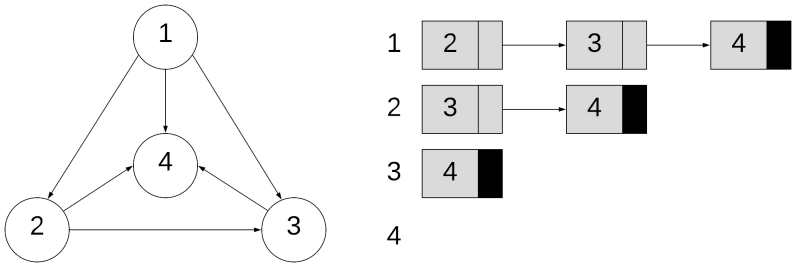


Figura 1: Exemplo de Grafo e Lista de Adjacência

Além das listas de adjacência, o grafo contém um vetor que guarda as idades de cada membro da equipe. A solução de cada um dos três comandos principais foi feita de forma individual. Sendo assim, as próximas subseções tratam a respeito de cada comando individualmente:

2.1 Commmander

O comando Commander requisita que seja encontrado o comandante mais novo de um integrante do grupo, seja ele comandante direto ou indireto. Para fazer isso, a abordagem tomada se baseou no seguinte pensamento: *Se existe um caminho, em um grafo direcionado, de um vértice A para um vértice B, ao inverter todas as arestas do grafo existe o caminho de B para A. Basta percorrer a ordem inversa das arestas.* Em outras palavras, dizer que um vértice A comanda um vértice B, significa dizer que existe um caminho de A para B. Ao inverter todas as arestas do grafo, o caminho se inverte também e, dessa forma, temos um caminho de B para A.

Com essa intuição em mente, o primeiro passo do Commander é inverter todas as arestas do grafo. Dessa forma temos um caminho de um integrante do grupo para todos os seus comandantes diretos e indiretos. O próximo passo é encontrar o comandante mais novo. Para isso, foi utilizado o algoritmo de Busca em Profundidade (DFS) com algumas modificações. A Busca em Profundidade, em poucas palavras, explora o mais fundo possível no grafo retornando apenas quando necessário, ou seja, quando não há mais nenhum vértice adjacente (e não explorado) que possa ser alcançado.

Nessa variação de DFS, além de manter a marcação de quais vértices já foram visitados, é necessário manter a informação de qual vértice é o mais novo encontrado. O vértice mais novo inicia com idade infinita. Cada vértice (com

exceção do original) confere primeiramente se ele é mais novo que o mais novo atual e atualiza esse dado caso for. Após isso é feita uma chamada recursiva para o próximo nó adjacente (que ainda não foi visitado) que faz a mesma verificação. Caso o vértice não tenha sucessores, retorna a idade mais nova encontrada até o momento. Ao final, retorna-se a idade do mais novo. Caso o retorno seja infinito, a saída é interpretada como se o vértice não tivesse comandantes (caso do vértice que não tem sucessores/adjacentes).

2.2 Meeting

O commando Meeting requisita que seja retornada uma ordem de fala dos integrantes em uma reunião. Essa ordem segue a hierarquia, ou seja, um comandante fala antes de seus comandados. Como o grafo para esse problema é um DAG (Grafo Direcionado Acíclico), sempre é possível gerar essa ordem de fala.

A solução para esse comando se baseou completamente no conceito de ordem topológica. A ordenação topológica é um tipo de ordenação $v_1, v_2, ..., v_n$ dos vértices onde, para cada aresta (v_i, v_j) nós temos que $i < j$. A Figura 2 mostra um exemplo de ordem topológica:

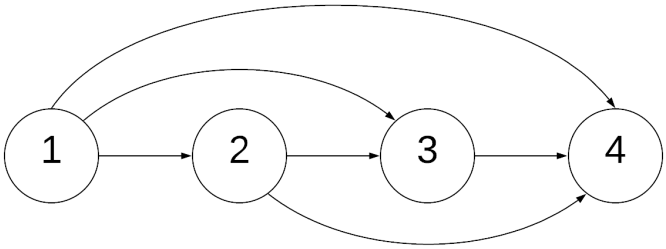


Figura 2: Ordem Topológica do Grafo da Figura 1. As arestas sempre vão de um comandante para quem ele comanda. Seguindo essa ordem, o vértice 1 deve ser o primeiro a falar, seguido do vértice 2, depois o 3 e por último o 4.

Para conseguir a ordenação topológica de um grafo, utiliza-se o algoritmo de Busca em Profundidade levando em consideração o tempo de finalização desse vértice na busca, ou seja, quando o vértice já não tem mais adjacentes para explorar. Ao finalizar a busca em um vértice, ele é adicionado em uma pilha. O resultado final da pilha é a ordenação topológica do grafo. No caso do grafo da Figura 1, o vértice 4 é o primeiro a finalizar, seguido do vértice 3, depois o 2 e por fim o vértice 1. Ao retirá-los da pilha, temos a ordem {*1, 2, 3, 4*}.

2.3 Swap

O comando Swap requisita que seja feita a troca da direção da aresta entre dois vértices caso ela exista e a troca não gere um ciclo no grafo. Em outras

palavras, se existia uma relação direta entre um integrante A e um integrante B, onde A comanda B, após o comando Swap o integrante B passa a comandar o A, caso isso não quebre a hierarquia da equipe gerando um ciclo. Ao final, deve-se retornar se a troca foi bem sucedida ou não.

O primeiro passo para resolver esse comando é verificar a existência de uma aresta entre os vértices nas duas direções, ou seja, de A para B e de B para A. Caso a aresta não exista, a troca não é possível e retorna-se que a troca não foi bem sucedida. Caso contrário, o próximo passo é verificar se a troca gera um ciclo no grafo. Para isso, utiliza-se novamente o algoritmo DFS, porém com outro tipo de checagem. Dessa vez, o que será checado é se durante o caminhamento um vértice "cinza" é atingido. *Um vértice "cinza" é um vértice que começou a sua busca no grafo, mas ainda não finalizou.* Caso um vértice desse tipo seja atingido durante a DFS, isso indica que há um ciclo no grafo. Caso contrário, não há ciclos. A Figura 3 mostra os dois casos.

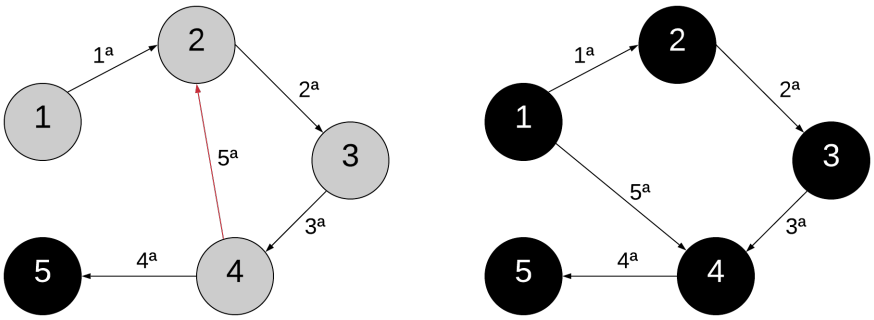


Figura 3: No primeiro grafo é detectado um ciclo ao atingir o vértice 2, que ainda não havia terminado sua exploração e estava marcado como cinza. No segundo grafo, não é detectado nenhum ciclo. O vértice 1 só encontra o 4 após este ter finalizado sua exploração e ter sido marcado como preto. A ordem de verificação das arestas é indicada pelos números próximos a elas.

3 Análise de Complexidade

3.1 Complexidade de Tempo

A análise da complexidade de tempo pode ser divida pelos comandos, da mesma forma que a implementação. Vamos começar com o comando Commander. A primeira tarefa executada nesse comando é inverter todas as arestas do grafo. Para isso, é necessário percorrer todas elas e inverter o sentido delas. Cada aresta é percorrida apenas uma vez e o custo de inserção é linear (pois cada aresta invertida sempre é inserida ao final da lista de adjacência). Sendo assim, a complexidade é da ordem do número de arestas do grafo, ou

seja, $O(|A|)$, sendo $|A|$ o número de arestas. Agora temos que analisar a complexidade para encontrar o comandante mais novo de um vértice. No pior caso, temos que o vértice que estamos observando é comandado diretamente (ou indiretamente) por todos os outros vértices. Nesse caso, a DFS percorre todos os vértices e arestas do grafo, o que resulta na complexidade geral de uma DFS, que é $O(|V| + |A|)$, sendo $|V|$ o número de vértices e $|A|$ o número de arestas do grafo. Assim, para o comando Commander, a complexidade final é dada pela soma das duas complexidades:

$$O(|A|) + O(|V| + |A|) = O(|V| + |A|)$$

Agora vamos analisar o próximo comando, que é o comando Meeting. A execução desse comando é feita por meio de uma DFS que adiciona o vértices que finalizaram sua busca em uma pilha. O custo de adicionar os vértices na pilha é constante $O(1)$, pois os vértices sempre são adicionados no topo da pilha (ou no começo da lista, caso a estrutura usada seja uma lista). Sendo assim, o custo total dessa operação está na DFS, que tem como complexidade $O(|V|+|A|)$ pois percorre todos os vértices e arestas do grafo. Assim, para o comando Meeting, a complexidade final é:

$$O(|V| + |A|)$$

O último comando a ser analisado é o Swap. Para fazer o Swap, primeiro temos o custo de verificar a existência de uma aresta no grafo. No pior caso, temos um vértice que conecta a todos os outros e o vértice com o qual queremos fazer o Swap é o último da lista de adjacência. Dessa forma, teremos que percorrer completamente a lista de adjacência do vértice principal, o que dá um custo $O(|V|)$. Após isso, fazemos a troca de direção da aresta, que tem um custo constante (a busca pela aresta já foi feita, então, ao usar o resultado dessa busca, a remoção dela no grafo principal é constante. Além disso, a inserção da nova aresta é feita inserindo o índice do vértice de origem no final da lista de adjacência do antigo vértice de destino, o que também é constante). A próxima operação a ser feita é descobrir se o grafo resultante tem um ciclo. Para isso, utilizamos novamente uma DFS que irá conferir se, durante a busca, algum nó "cinza" (que ainda esteja em seu processo de exploração) é atingido. A DFS, como mostrado nas análises anteriores, tem um custo $O(|V| + |A|)$. A última operação, se necessária, é desinverter o sentido da aresta, caso tenha sido encontrado um ciclo. Essa operação tem custo constante caso feita de uma das duas formas a seguir:

- Inserção do vértice de destino original no final da lista de adjacência do veŕtice de origem e remoção do último elemento da lista de adjacência do vértice de destino (original);
- Manter a informação da posição do item removido na lista de adjacência no segundo passo. Dessa forma, podemos adicionar o vértice removido na exata posição onde ele estava. Além disso, remover o elemento adicionado na lista de adjacência do vértice de destino (que está na última posição).

Assim, para o comando Swap, a complexidade final é dada pela soma das complexidades:

$$O(|V|) + O(1) + O(|V| + |A|) + O(1) = O(|V| + |A|)$$

Juntando todas as complexidades, temos:

$$O(|V| + |A|) + O(|V| + |A|) + O(|V| + |A|) = O(|V| + |A|)$$

que é a complexidade de tempo total da solução.

3.2 Complexidade de Espaço

Na análise de complexidade de espaço, vamos análisar a criação do grafo principal, a função comander (que gera um novo grafo com as arestas invertidas) e a função meeting (que cria uma pilha que armazena a ordem topológica).

A criação do grafo principal armazena um vetor de idades de tamanho $|V|$, sendo $|V|$ a quantidade de vértices. Sendo assim o vetor é $O(|V|)$ em custo de espaço. Além disso, é criado um vetor de listas de adjacência, que armazena a lista de cada vértice. Como cada vértice tem uma lista e o número de elementos total das listas é equivalente ao número de arestas, temos um custo de espaço $O(|V| + |A|)$, sendo $|V|$ a quantidade de vértices e $|A|$ a quantidade de arestas no grafo. Somando os dois custos, temos o custo final de criação do grafo, que é:

$$O(|V|) + O(|V| + |A|) = O(|V| + |A|)$$

A função comander gera um novo grafo, porém invertendo as arestas. O grafo gerado tem o mesmo número de vértices e arestas do grafo principal, e possui o mesmo vetor de idades do membros da equipe. Sendo assim, o custo de espaço dessa função é da mesma ordem da criação do grafo principal, ou seja,

$$O(|V| + |A|)$$

Por fim, a função meeting cria uma pilha que armazena a ordem topológica dos vértices. A pilha contém apenas uma permutação dos vértices que corresponde a ordenação topológica do grafo, ou seja, ela contém todos os vértices do grafo e cada um só aparece uma vez na pilha. Com isso, podemos concluir que o tamanho da pilha é equivalente ao número de vértices no grafo, e o custo de espaço disso é

$$O(|V|)$$

Juntando todas as complexidades, temos:

$$O(|V| + |A|) + O(|V| + |A|) + O(|V|) = O(|V| + |A|)$$

que é a complexidade de espaço total da solução.

4 Análise Experimental

Para a análise experimental, foram utilizados grafos de tamanho 5 até 100, incrementando em 5 o tamanho do grafo a cada teste. Para as arestas, foi utilizado o maior número de arestas possível para o grafo que não gere um ciclo. Nesse caso, nós temos que o 1º vértice comanda os demais, o segundo comanda todos com exceção do primeiro, e assim por diante. Disso podemos extrair o número total de arestas, que é dado pela fórmula:

$$\sum_{i=1}^{|V|} |V| - i = \frac{|V|(|V|-1)}{2}$$

sendo $|V|$ o número de vértices do grafo. Além disso, são executados 6 comandos por teste, com a intenção de cobrir ao máximo o funcionamento de cada comando. Os comandos executados são:

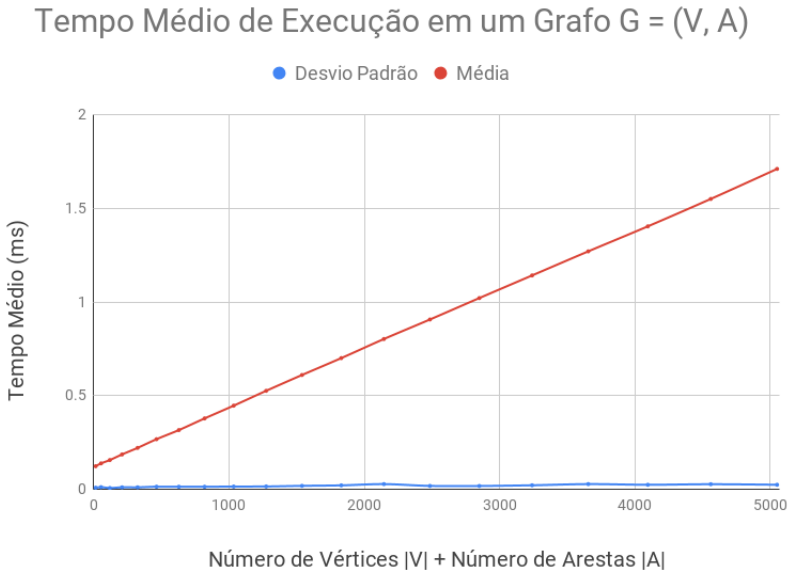
- 2 Commanders - 1 para um vértice sem comandante e outro para um vértice com comandante;
- 3 Swaps - 1 para cada caso (vértices sem aresta entre si, swap que gera ciclo e swap válido);
- 1 Meeting - Apenas para imprimir a ordem topológica do grafo.

Cada teste foi realizado 100 vezes para garantir uma média consistente de tempo. Abaixo temos uma tabela com o resultado dos testes:

Nº de Vértices	Nº de Arestas	Vértices + Arestas	Média do Tempo de Execução (ms)	Desvio Padrão do Tempo de Execução
5	10	15	0.12235	0.00750
10	45	55	0.13747	0.01054
15	105	120	0.15537	0.00594
20	190	210	0.18532	0.00954
25	300	325	0.22046	0.00965
30	435	465	0.26686	0.01286
35	595	630	0.31559	0.01269
40	780	820	0.37789	0.01258
45	990	1035	0.44617	0.01371
50	1225	1275	0.52549	0.01418
55	1485	1540	0.61011	0.01750
60	1770	1830	0.70040	0.02004
65	2080	2145	0.80244	0.02676
70	2415	2485	0.90674	0.01711
75	2775	2850	1.02133	0.01676
80	3160	3240	1.14236	0.02051
85	3570	3655	1.27108	0.02681
90	4005	4095	1.40474	0.02321
95	4465	4560	1.55126	0.02616
100	4950	5050	1.71181	0.02326

Tabela 1: Tamanho dos grafos, Média do Tempo de Execução e o Desvio Padrão do Testes

Com os dados dessa tabela, foi montado um gráfico para verificar a relação entre o número de vertices e arestas, e o tempo de execução.



Por meio desse gráfico, podemos ver que o tempo tem uma relação linear com a soma do número de vértices e arestas, que é o esperado, visto que a ordem de complexidade, como mostrado na seção anterior é $O(|V| + |A|)$.

5 Respostas Para As Questões Propostas

5.1 Por que o grafo tem que ser dirigido?

Porque as relações entre os membros não podem ser bidirecionais. Em outras palavras, um integrante A da equipe não pode comandar um integrante B e, ao mesmo tempo, o integrante B comandar o A. Isso geraria uma quebra de hierarquia.

5.2 O grafo pode ter ciclos?

Não, pois assim teriamos um comandante sendo comandado, diretamente ou indiretamente, por alguém de hierarquia mais baixa do que ele, o que também geraria uma quebra de hierarquia.

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

Se considerarmos árvore como grafos não direcionados exclusivamente, então o grafo não pode ser uma árvore. Caso árvores direcionadas também sejam consideradas como árvore, o grafo pode ser uma árvore, mas não é necessariamente uma árvore. Basta observar que um vértice pode ser comandado por 2 ou mais vértices diferente, o que já não é correspondente a estrutura de uma árvore.

Referências

[1] B. Kleinberg and É. Thardos. *Algorithm Design*. Pearson, 2005.

[2] N. Ziviani. *Projeto de Algoritmos com Implementações em Pascal e C*. Cengage, 2011.