Trabalho Prático 1 - Grupo de Blackjack de Alunos da UFMG

(Grafos e Caminhamento em Grafos)

Caio Guedes de Azevedo Mota

2018054990

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

caioguedes@ufmg.br

1. Introdução

O intuito deste trabalho prático é praticar os conceitos de grafos e entender a implementação de algoritmos de caminhamento em grafos, assim como entender como fazer modificações nelas.

O problema é o seguinte: são providenciados dados de equipes de alunos da UFMG, com um número N de alunos. Nessas equipes, há M relações de comando, em que um aluno pode ser comandado por outros membros e comandar outros membros. Essas relações são assimétricas, ou seja, se um aluno A comanda um aluno B, B não comanda A também.

Deve ser feito um programa que realiza 3 tipos de instruções sobre dados desse grafo:

- ➔ **Swap**: verifica se um aluno A comanda B e, caso comande, troca a relação de comando para B comandar A;
- ➔ **Commander**: retorna a idade do comandante mais novo de um dado aluno A (direta ou indiretamente);
- ➔ **Meeting**: imprime uma ordem de alunos para falar em reuniões, dado que nenhum aluno fala antes de qualquer dos seus comandantes.

2. Solução do Problema

Foi construído um grafo direcionado e não ponderado com as informações iniciais, usando uma lista de adjacência como estrutura de dados. Ao receber o número N de alunos, o grafo é construído com N vértices, cada vértice um aluno, armazenando sua idade e posição no grafo. Além disso, ao receber o número M de relações de comando, cada uma é adicionada como uma aresta do grafo. Dados dois alunos A e B, se A comanda B, é adicionada uma aresta direcionada (A,B) representando essa relação.

As instruções foram implementadas usando adaptações da busca em largura ou *Depth-First Search* (DFS), realizando operações intermediárias ao longo das buscas para cada instrução. Uma breve explicação de cada uma está a seguir:

2.1. Swap

A instrução SWAP para trocar uma relação de comando entre A e B simplesmente procura se há aresta entre A e B e, se há, realiza uma troca. Caso ela encontre um ciclo, reverte a troca. Da maneira como está implementada, a ordem dos vértices não importa para a troca (ou seja, Swap(A,B) é igual a Swap(B,A)). É usado um outro procedimento para checar ciclos, que realiza uma DFS, armazenando os vértices não totalmente explorados em um vetor. Caso a busca resulte em um desses vértices que já foi armazenado durante a busca, há um ciclo.

Descrevendo o funcionamento da função em pseudo-código:

```
Swap(u,v):
	Para cada aresta (u,w) saindo de u:
		se w == v:
			Remove (u,v)
			Adiciona (v,u)
			se ChecarCiclo() == true:
				Remove (v,u)
				Adiciona (u,v)
				Retorna falso
			senão:
				Retorna verdadeiro
	Retorna falso caso não ache o vértice v

ChecarCiclo():
	inicializar Visitados como vértices visitados
	inicializar Descobertos como vértices descobertos mas não explorados totalmente
	Visitados[i] = falso para todo vértice i
	Descobertos[i] = falso para todo vértice i
	Para todos os vértices u do grafo:
		se DFSChecarCiclo(u) == verdadeiro:
			retorna verdadeiro pois achou ciclo
		se não:
			retorna false pois não achou

DFSChecarCiclo(u):
	se Visitados[u] == falso:
		Visitados[u] = verdadeiro
		Descobertos[u] = verdadeiro
		Para cada aresta (u,v) saindo de u:
			se Visitados[v] == falso e DFSChecarCiclo(v) == verdadeiro:
				retorna verdadeiro
			senão se Descobertos[v] == verdadeiro:
				retorna verdadeiro
	Descobertos[u] = falso
	retorna falso
```

2.2. Commander

A instrução COMMANDER procura o ancestral mais novo de um dado vértice u do grafo, representando o comandante mais novo do aluno do dado vértice. Ele realiza uma DFS no grafo inteiro, guardando, para cada início de busca, a idade do vértice de início, e atualizando esse valor caso encontre uma idade menor. Também é guardada uma idade mais nova de comandantes de u, e, caso u seja encontrado pela DFS, atualiza esse valor caso a idade mais nova da busca seja menor que a atual idade menor de comandantes de u.

Descrevendo o funcionamento da função em pseudo-código:

```
DFSCommander(w):
	Descobertos[v] = falso para todo v do grafo
	mais_novo_até_w = MAXINT
	Para todos os vértices s:
		DFSRecurCommander(s,w,s.idade,mais_novo_até_w)
	retornar mais_novo_até_w


DFSRecurCommander(u, w, mais_novo, mais_novo_até_w):
	se Descobertos[u] == falso:
		Descobertos[u] = verdadeiro
		se u.idade < mais_novo:
			mais_novo = u.idade

		Para cada aresta (u,v) saindo de u:
			se v = w e mais_novo < mais_novo_até_w:
				mais_novo_até_w = mais_novo
			senão se Descobertos[v] = falso:
				DFSRecurCommander(v, w, mais_novo, mais_novo_até_w,
			Descobertos)
```

2.3. Meeting

A instrução MEETING imprime a ordem topológica do grafo. Dado que a ordem topológica é uma ordem na qual um vértice v não precede um outro vértice u caso haja a aresta (u,v), isso significa que nessa ordem ninguém fala antes de alguém que o comanda. A ordem é obtida usando mais uma DFS, e empilhando vértices assim que são totalmente explorados. No final, a pilha é impressa do topo até o final. A implementação é bem simples a partir da descrição, então nenhum pseudo-código será colocado aqui por objetividade.

3. Instruções de Compilação e Execução

O programa é compilado usando o **g++** (compilador GNU para C++). Para a compilação do programa, deve-se utilizar o Makefile incluído com os arquivos do trabalho. Ao digitar o comando *make* será feito um executável *tp1*, que pode ser executado da forma:

```
./tp1 [entrada.txt]
```

Sendo *entrada.txt* um nome de arquivo de entrada genérico passado como parâmetro de execução do programa.

O programa e os testes foram feitos em uma máquina com as seguintes especificações técnicas:

- **Processador:** Intel Core i7-7500U @ 2.7 Ghz
- **Memória RAM:** 8GB

- **Memória Secundária:** SSD Kingston A400 120GB
- **Sistema Operacional:** Linux (distribuição: Manjaro Linux 18.0.4)

Caso seja de interesse, o código para o trabalho pode ser encontrado no repositório público em https://github.com/caioguedesam/graph_algorithms_alg1

4. Análise Experimental e Complexidade

A ordem de complexidade de tempo e espaço do programa é de *O(N+M)*, sendo N o número de vértices e M o número de arestas no grafo. O detalhamento dessa ordem de complexidade é justificado a seguir:

- ➔ A construção do grafo é simplesmente adicionar vértices e arestas. São um número constante de operações para adicionar cada elemento, e essas operações são feitas uma vez por vértice ou aresta. Logo, a complexidade de tempo é O(N+M). Adicionalmente, é necessário N+M de espaço para armazenar os vértices e as arestas no grafo, logo a complexidade de espaço é O(N+M).
- ➔ O método para checar ciclos é uma simples implementação da DFS com vetores adicionais. A complexidade de tempo disso é a mesma da DFS, O(N+M). Além disso, esses vetores incluem a lista de vértices visitados (tamanho N) e a lista de vértices descobertos mas não totalmente explorados (tamanho N), logo complexidade de espaço O(N).
- ➔ O método SWAP(A,B) verifica todas as arestas (A,V) partindo de A procurando B. Depois disso, é realizada a troca da aresta (que é constante), a checagem de ciclo, e, caso haja ciclo, são realizadas operações constantes para trocar de volta a aresta trocada. Logo, a complexidade de tempo é O(M + N + M + 1) = O(N+M). Não é utilizado espaço adicional além do utilizado na checagem de ciclo, logo complexidade de espaço O(N).
- ➔ O método COMMANDER é uma implementação da DFS com algumas variáveis adicionais e operações sobre essas variáveis. São feitas operações sobre cada vértice ao visitá-lo pela primeira vez. Logo, a complexidade de tempo é O(N+M). O espaço adicional usado é o vetor de vértices visitados usado pela DFS, que é O(N) espacialmente.
- ➔ O método MEETING é, novamente, uma simples modificação do DFS, com operação de empilhar cada vértice quando foi totalmente explorado, além das operações para imprimir a pilha resultante. Logo, a complexidade de tempo é O(N+M). O espaço adicional é o vetor de vértices visitados (tamanho N) e a pilha para guardar os vértices explorados (tamanho N), o que é O(N).

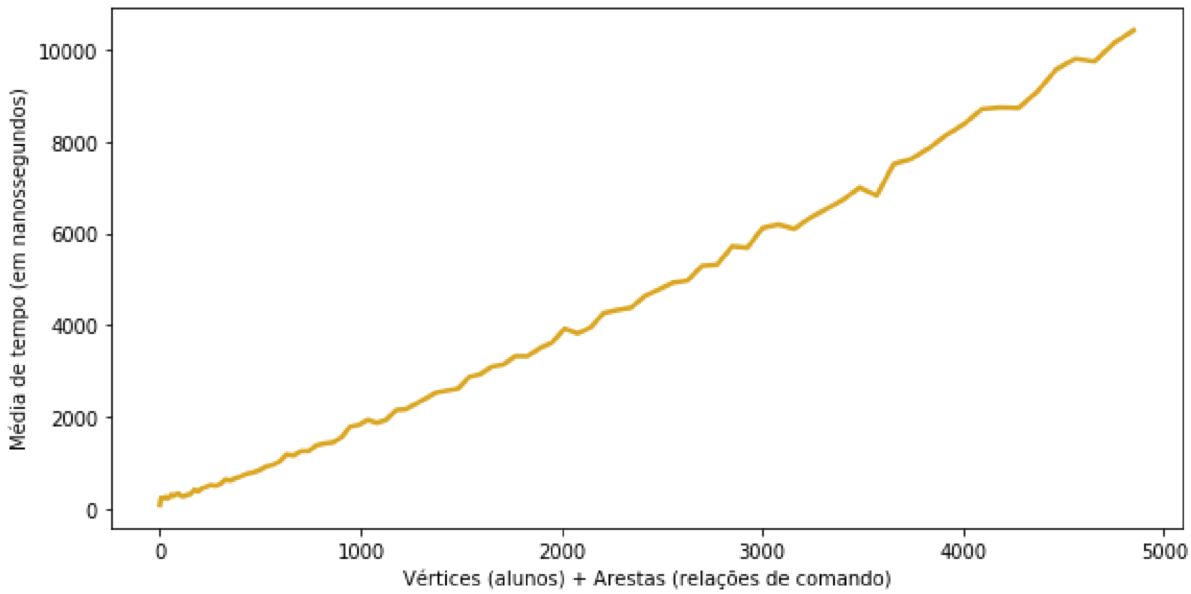
Ao somar todas as ordens descritas acima, no final a ordem de complexidade de tempo é **O(N+M)** e a ordem de complexidade de espaço é **O(N+M)**.

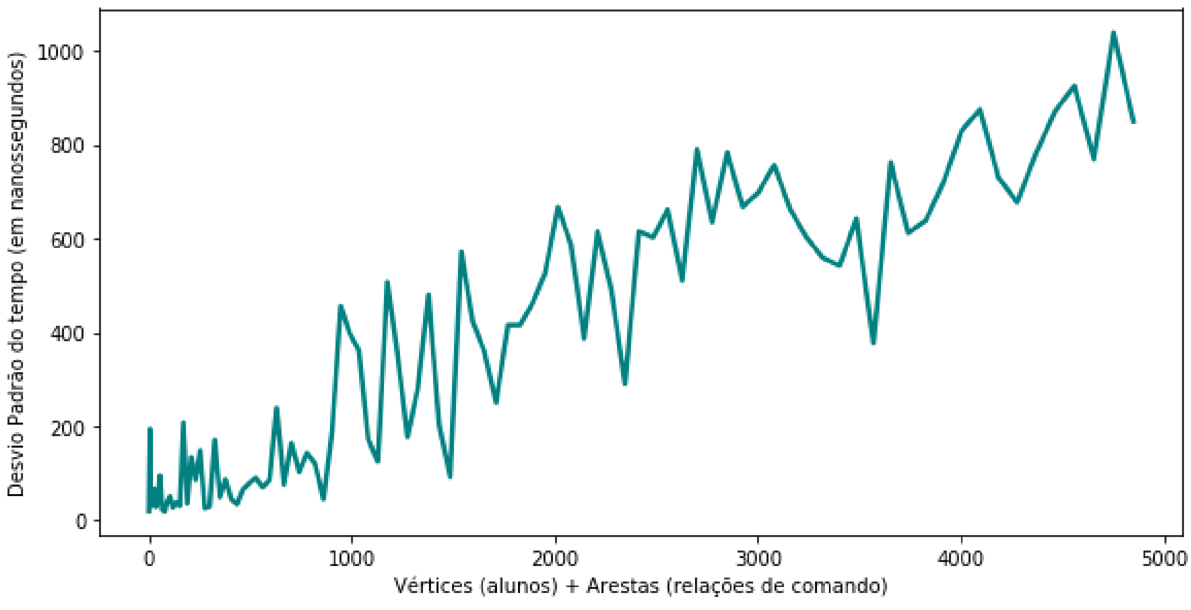
Foram feitos testes para experimentação e verificação da ordem de complexidade do programa. Os testes foram feitos da seguinte maneira:

- ➔ Foram estabelecidos uma lista de tamanhos para os grafos dos testes. Já que o tamanho é limitado pelo número de vértices, são 101 tamanhos, indo de 0 a 100.
- ➔ Para cada tamanho, foram feitos os números máximos de arestas entre eles (isso é, para N vértices, são M = (N * (N-1))/2 arestas.

➔ Para cada tamanho, foram feitos 10 testes, e foi retirada a média de tempo de execução e o desvio padrão entre esses.

Os resultados dos testes estão exibidos nos gráficos a seguir (gerados com a biblioteca matplotlib do Python 3):





Examinando o gráfico das médias de tempo, é possível verificar experimentalmente a ordem de complexidade de tempo linear proposta anteriormente.

5. Algumas Questões

Por que o grafo tem que ser dirigido?

Pois as relações de comando que geram as arestas não são assimétricas. Isto é, a aresta (A,B) significa que o aluno A comanda o aluno B, mas não que o aluno B também comanda o aluno A. Isso é importante para preservar a hierarquia dentro do grupo de estudantes.

O grafo pode ter ciclos?

Não. Caso o grafo contenha algum ciclo, significa que há uma falha na hierarquia entre os alunos, e um aluno poderia comandar direta ou indiretamente alguém que o comanda direta ou indiretamente. Isso também é importante para preservar a hierarquia entre estudantes.

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

Considerando árvores como apenas grafos não direcionados, o grafo nunca é uma árvore.

Considerando árvores direcionadas como árvores, o grafo pode ser mas não necessariamente é uma árvore. Isso se dá pelo fato de haver a possibilidade de um mesmo aluno ser comandado diretamente por dois alunos diferentes, que não acontece numa árvore em que cada vértice tem apenas um pai. Entretanto, isso não é exclusivo para todas as possibilidades de grupos, e pode haver uma hierarquia na qual nenhum estudante tem mais de um comandante direto, que resultaria numa árvore direcionada.

6. Referências

Kleinberg, J.; Tardos, É. *Algorithm Design*: 1 ed. Boston, MA: Pearson Education Inc, 2006.