Trabalho Prático de Algoritmos 1

Roberto Gomes Rosmaninho Neto

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

robertogomes@dcc.ufmg.br

1. Introdução

O problema proposto por esse trabalho foi modelar e implementar um programa para o Grupo de Blackjack de alunos da UFMG. Nesse programa, cada aluno possui número identificador e idade únicos, além disso há uma hierarquia dentro do grupo, sendo assim, cada aluno pode comandar e ser comandado por outro. Logo, dada essa estrutura, o programa foi modelado como um grafo, onde cada vértice representa um aluno e cada aresta uma relação de comando. Esse grafo não deve possuir ciclos e deve dirigido com arestas sem peso. Dessa forma, o grafo foi implementado utilizando listas encadeadas, visando a baixa complexidade para caminhar pelos vértices. O programa requisitado pelo grupo de alunos é composto por 3 operações principais:

- **Commander:** Essa operação é responsável por retornar a idade do aluno mais novo que comanda de forma direta ou indireta o aluno A.
- **Swap:** Essa operação é responsável por inverter a relação de comando entre dois alunos quando existente, ou seja, se o Aluno A comanda o aluno B após o comando Swap o aluno B comandará o aluno A.
- **Meeting:** Essa operação é responsável por retornar uma possível ordem de fala durante uma reunião do grupo.

As operações realizadas nesse programa são baseadas em 2 algoritmos famosos, Busca em Profundidade(DFS) e Ordenação Topológica, ambos com complexidade $O(|V| + |E|)$, obedecendo uma das restrições para a construção do programa. Outras duas restrições importantes são nas operação de Swap e na operação de Meeting. Na primeira, a troca de comando só ocorre se já existe uma relaçãp e, caso exista a troca não pode gerar um ciclo no gráfico e na segunda operação, um aluno A não pode falar antes de um aluno B se B comandar A de forma direta ou indireta. A operação Commander, por sua vez não possui restrições.

A entrada do programas deveria ser feita via argumento, pela linha de comando, passando o nome do arquivo de entrada logo após o arquivo executável. A saída do programa poe sua vez é dada em um arquivo ”out.txt” obedecendo a formatação que será detalhada posteriormente, assim como a formatação utilizada no arquivo de entrada.

2. Implementação

Este trabalho prático foi implementado na linguagem C++, utilizando as boas práticas de orientação à objetos. Os Ambientes de testes utilizado para executar o código foram um Macbook Air Core i5, dual core de 1.8Ghz, 8GB de RAM e SSD de 128GB e um Dell OptiPlex 7040M com processador i7vPRO. Os compiladores utilizados para

testes foram o Apple LLVM version 10.0.1 (clang-1001.0.46.4) e o GCC 5.4.0 da GNU Compiler Collection.

A implementação desse programa foi realizada utilizando apenas 3 arquivos: main.cpp, graph.h e graph.cpp, sendo todas as operações e funções declaradas no segundo arquivo e implementados no terceiro. Assim no main.cpp apenas foram realizadas as operações de leitura e escrita de arquivos, além das chamadas de funções quando necessárias. As principais estruturas de dados utilizadas foram lista encadeada, utilizando Vector, e uma pilha, utilizando Stack, ambas da biblioteca padrão de c++.

2.1. Funções

Esse programa possui apenas uma classe: Graph. Esta é responsável por criar o grafo de alunos e realizar todas as operações neste.

As funções presentes na classe Graph e suas descrições são as seguintes:

- ***Graph***: Essa função inicializa o grafo com algumas variáveis que são usadas como varáveis globais durante o decorrer do programa, além dos números de vértice e arestas recebidos por ela como parâmetro, 3 listas são inicializadas, a que conterá as relações de comando, a que conterá a relação inversa de comando e a de nós visitados. Um *array* contendo as idades de todos os alunos também é criado junto com uma variável que indica quando há ciclo no grafo, nesses dois casos, todos os valores são iguais a zero.
- ***~Graph***: Responsável por desalocar a memória utilizada pelo grafo durante a execução do progama.
- ***addNode***: Adiciona a idade de um aluno na posição correspondente de seu indicador.
- ***addEdge***: Recebe um identificador A e um B representando uma aresta de A para B, assim o identificador B é adicionado na lista de A na lista de adjacência normal e o identificador de A é adicionado na lista de B na lista de adjacência invertida.
- ***reset_visited***: Responsável por zerar a variável de ciclo, colocar a maior idade possível na variável de idade mínima e zera cada elemento do vetor de visitados.
- ***DFS***: Realiza uma busca em profundidade a partir de um aluno A enquanto atualiza a idade mínima de um aluno B que comanda A (quando a busca é realizada na lista de relações invertidas) e caso encontre um ciclo retorna falso, caso contrário retorna verdadeiro e a variável min contém o valor esperado.
- ***BuildStack***: Por meio de chamadas recursivas em nós não visitados, utiliza a mesma estratégia da DFS para empilhar os nós que não possuem filhos.
- ***Meeting***: Responsável por chamar a *BuildStack* cada nó
- ***Commander***: Cria um novo grafo com a lista de relações inversa como a lista principal e ordena uma DFS nela, assim o valor da variável auxiliar de menor idade possui a idade do aluno mais novo que comanda o aluno recebido como argumento.
- ***swapEdge***: Se existe uma relação entre os alunos A e B, tal que, A comanda B, então o aluno B é apagado da lista de A e o aluno A é adicionado na lista de B. Há a verificação se essa lista gerou ciclo no grafo por meio de uma chamada da DFS e caso tenha gerado A é deletado da lista de B e B é realocado na lista de A, caso contrário retorna valor verdadeiro.

2.2. Fases de Implementação

A implementação do programa ocorreu em tres fases: Operações necessárias para manipulação do grafo e implementação dos algoritmos básicos, operações *Swap*, *Commander* e *Meeting*, e por fim leitura de entrada, criação e operações no grafo e escrita de saída. Cada fase será detalhada posteriormente, assim como as decisões que justificam a adoção desse formato de implementação.

2.2.1. Operações Primárias

Todas as funções do programa utilizam as variáveis globais inicializadas no método de criação do grafo. Importante destacar que o grafo com as relações invertidas, apesar de aumentar a complexidade de espaço do programa, evita a criação de uma função para inverter todas as arestas do grafo para que a Operação *Commander*, a partir da Busca em Profundidade em um nó A, possa retornar a menor idade de um dos comandantes diretos ou indiretos de A.

Um detalhe chave para a utilização de varáveis globais foi a utilização de um *array* para guardar as idades, evitando assim a necessidade da criação de um vetor de estruturas de nó contendo idade e identificador de cada aluno o que dificultaria a iteração pelos nós do grafo e possivelmente aumentando a complexidade do problema. Utilizando apenas o identificador dos alunos como nó, foi possível a utilização de uma lista de inteiros para representar o grafo e quando necessário descobrir a idade de um aluno a operação de consulta no *array* é simplesmente $O(1)$

A variável auxiliar de ciclo garante quando um ciclo é gerado, pois a DFS só a atualiza quando encontra um ciclo e esta só é zerada novamente antes de uma nova DFS, utilizando para isso a função reset_visited.

A variável auxiliar min, apenas guarda a menor idade de um aluno encontrada pela DFS.

Além disso, nessa fase também foram criadas as funções addNode, addEdge, reset_visited, DFS e *BuildStack*. Na primeira apenas a idade do aluno é atualizada no *array _ages*, pois implementação do grafo nesse programa já cria uma nó para cada aluno que possa ter uma relação com outro, é sabido que essa não é a estratégia ótima quando se utiliza listas para representar um grafo, no entanto, essa estratégia se mostrou significante e útil para a manipulação de dados, visto que a criação e ordenação da lista para cada nova relação teria custo maior que assumir a existência do nó na lista, mesmo sem relações com outros nós.

A segunda função apenas cria uma relação na lista principal e sua inversa na lista de *reverse_adj*. A terceira é necessária para sempre executarmos uma DFS de forma segura. A DFS em si teve seu algoritmo original modificado para a que pudesse satisfazer tanto as necessidades da operação *Commander*, descobrir a menor idade, quanto da Operação *Swap*, descobrir quando há vértice num grafo modificado. A última função dessa fase é parte do algoritmo de ordenação topológica utilizando uma stack para armazenar a ordem de fala dos alunos.

2.2.2. Operações Principais

A operação *Swap* apenas realiza a verificação do comando de A para B, conforme interpretação do requisitado no documento de apresentação do problema, dessa forma se há uma relação de comando de B para A, esta permanece inalterada.

A operação de *Commander*, apesar aumentar a complexidade de espaço do programa, evita a passagem da lista de adjacência para toda operação com a DFS e evita maiores erros.

A operação *Meeting* assume que o primeiro nó não é comandado por ninguém e assim chama a função *BuildStack* para cada nó também no topo da hierarquia, devido a construção da ordenação topológica com a estratégia da busca em profundidade.

2.2.3. Leitura, Escrita e Chamadas de Operações na Função Principal

A leitura do arquivo de entrada, bem como a escrita do arquivo de saída, é realizada utilizando a biblioteca *fstream*. O grafo é criado logo após a leitura da primeira linha que contém o número de vértice, arestas e comandos a serem realizados. Após a criação do grafo, dois *arrays* são inicializados com o tamanho do número de operações, um para guardar qual operação deve ser realizado e outro para guardar o resultado desta.

Então as idades são lidas do arquivo e atualizadas no grafo, bem como as arestas são lidas e criadas. Após a criação e atualização de todos os elementos necessários do grafo as operações são lidas, armazenadas no *array*, assim como a função referente a operação é chamada e também possuí seu valor armazenado, caso as operações sejam de *Swap* e *Comander*.

Por fim, outro laço iterando nas operações é realizado para escrever no arquivo de saída o resultado de cada Operação logo após a letra que a identifica.

2.3. Entradas e Saídas

O programa deve receber o nome do arquivo de entrada via linha de comando logo após o executável como mostra o exemplo a seguir:

```
$ ./tp1 input.txt
```

O arquivo de entrada deve conter três inteiros na primeira linha: N, M, I. O primeiro representando o número de vértices, o segundo representando o número de arestas e o terceiro representando o número de operações a serem realizadas. A próxima linha contém N inteiros representando as idades de cada aluno, cada idade pode variar entre 1 e 100 inclusive. As próximas M linhas representam as relações de comando entre A e B, sendo A e B inteiros, tal que $1 <= A, B <= N, A! = B$. Por fim, as próximas I linhas contém uma letra representando o tipo da operação. Caso a operação seja de *Meeting* apenas a letra M deve aparecer, caso seja de *Commander* a letra C deve aparecer seguida de um espaço e um inteiro A, tal que $1 <= A <= N$. Finalmente, se a operação for de *Swap* a letra S deve aparecer seguido de um espaço, um inteiro A, espaço e um inteiro B nessa ordem, tal que $1 <= A, B <= N, A! = B$.

O arquivo de saída deve conter I linhas e o resultado de cada linha depende da da i operação, tal que $1 <= i <= I$. Caso a operação i seja de *Commander* a letra C deve aparecer seguida de um espaço e do resultado da operação ou * caso o aluno não seja comandado por ninguém. Caso a operação serja de *Swap* a letra S deve aparecer seguida de um espaço e da letra T caso a operação tenha sido bem sucedida ou da letra F caso contrário. Por fim, se a operação i for de *Meeting* a letra M deve aparecer seguida de uma possivel ordem de fala dos alunos.

3. Análise Experimental

A principal requisição para a implementação desse programa é que ele seja linear no número de vértices + o número de arestas no grafo, ou seja $O(|V|+|E|)$. Dessa forma, será mostrado a seguir que todas as operações são realizadas neste tempo linear.

3.1. Premissas

Os resultados analisados nesse documento se referem aos testes realizado no OptiPlex 7040M com processador i7vPRO, o programa foi compilado com o GCC 5.4.0 no sistema operacional Ubuntu 16.04.10.

3.2. Metodologia

Os arquivos de entrada foram gerados de modo aleatório por um programa em *Python* seguindo as restrições presentes nesse documento. A quantidade de vértices variou de 10 em 10, entre 10 e 100. A quantidade de arestas é aleatória seguindo as restrições já apresentadas. Cada arquivo de entrada possui um tipo de operação realizada 1000 vezes.

Foram criados 10 arquivos de entrada para cada tipo de operação. Foram necessárias 10 execuções do programa para cada arquivo de entrada para que a média e o desvio padrão dessas entradas pudessem ser analisados, para isso utilizou-se um *script* em *bash*.

O tempo de execução do programa foi contato em Nanosegundos e os gráficos representando o Tempo pela soma de vértice e arestas de cada execução pode ser visualizado a seguir:

3.2.1. Análise de Complexidade

O programa é construído praticamente por completo em cima da estratégia de busca em profundidade, cujo algoritmo executa em tempo $O(|V| + |E|)$.

As funções do programa possuem a seguinte complexidade:

- ***Graph***: $O(|V|)$, para a criação dos vetores de tamanho V.
- ***~Graph***: $O(|V|)$ para a exclusão de cada aresta.
- ***addNode***:$O(1)$, atualização de vetor de index conhecido.
- ***addEdge***:$O(1)$, adiciona elemento atrás do vetor, sem necessidade de qualquer alteração em sua ordem.
- ***reset_visited***: $O(|V|)$, para zerar os Z elementos do vetor de visitados.
- ***DFS***: $O(|V|+|E|)$, implementação da DFS padrão, as alterações modificadas sao de desvio de atualização de variável, ambas $O(1)$.

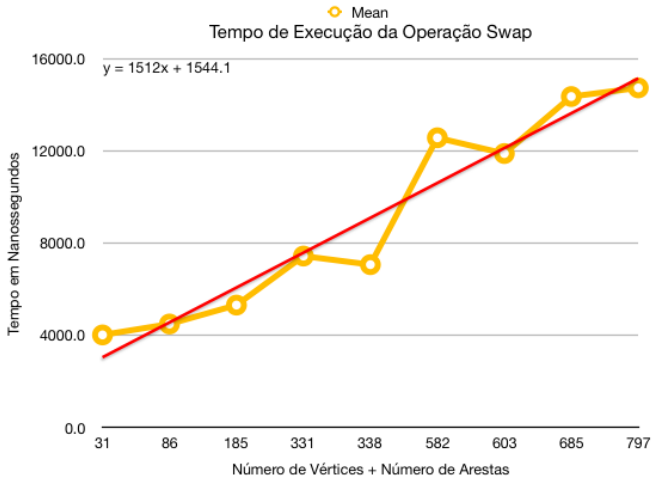

Figura 1. Tempo de Execução da Operação Swap

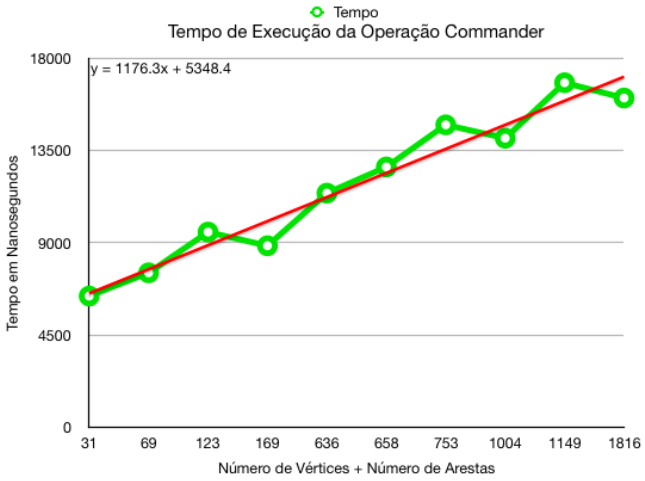

Figura 2. Tempo de Execução da Operação Commander

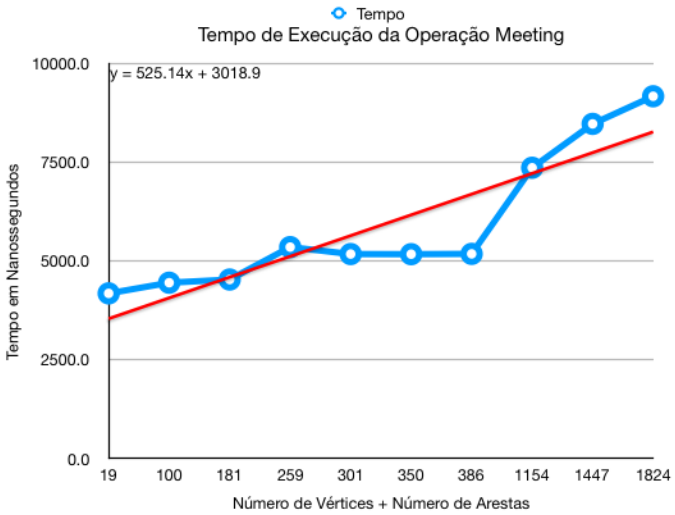

Figura 3. Tempo de Execução da Operação Meeting

- ***BuildStack e Meeting***: $O(|V|+|E|)$, Ordenação topológica baseada no algoritmo de DFS.
- ***Commander***: $O(|V| + |E|)$, cria um novo grafo em $O(|V|)$, mas a prinfipal operação é a DFS que é $O(|V| + |E|)$.
- ***swapEdge***: $O(|E|)$ para verificar a posição do elemento que será trocado, $O(1)$ para realizar a troca e $O(|V| + |E|)$ para realizar a DFS procurando por ciclos, logo é $O(|V| + |E|)$.