Trabalho Prático 1

Grupo de Blackjack UFMG

Guilherme Mendes de Oliveira
Departamento de Ciência da Computação

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

guilhermemendes@ufmg.br

1. Introdução

Este trabalho tem como objetivo implementar a organização de uma equipe de *blackjack* da UFMG inspirada na história dos alunos do MIT. Estes alunos se organizarão para efetuarem a contagem de cartas de tal modo a não serem identificados.

A contagem de cartas tem como princípio fundamental a premissa de que cartas de pontuação alta favorecem o jogador, cartas de pontuação mais baixa favorecem a mesa.

Figura 1 – Esquema da contagem de cartas.

Para tal, além de armazenar a organização previamente fornecida, é preciso que o programa a altere quando solicitado a ordem de comando de dois membros com a função *swap*, retorne o responsável por um subgrupo a partir de determinado parâmetro, no caso aquele que possui a menor idade, com a função *commander*, e retorne em uma ordem específica a composição do grupo naquele determinado momento com a função *meeting*.

2. Implementação

2.1 Instruções de compilação e execução

O programa foi desenvolvido na linguagem C++ e compilado pelo G++ da Minimalist GNU for Windows (MinGW) no ambiente Linux distribuição Ubuntu versão 18.04.02.

2.2 Estrutura de Dados

Para a implementação foram utilizados dois tipos abstrato de dados.

Vértice: Representa o objeto a ser armazenado no Grafo, ele possui uma identificação e armazena a idade do aluno e um vector com todos os vértices vizinhos a ele, por se tratar de um Grafo direcionado, todos os vértices que possuem arestas saindo dele para algum outro.

```
class Vertice{
    public:
    int idade;
    int indiceEntrada;
    vector<Vertice> adj;
    bool visitado;
    char corVertice;
    Vertice(int Idade, int indiceEntrada);
    void printList();
};
```

Figura 2 – Implementação do TAD'S Vertice.

Graph: Representa o grafo direcionado como um arranjo de Vértices, e todos os métodos que o envolvam em sua manipulação.

```
class Graph{
    public:
    int numVertices;
    vector<Vertice> Vertices;

    Graph(int nVertices);
    bool verificaCiclos();
    bool verificaCiclosAux(int idVertice, bool visited[], bool *recStack);
    void addVertices(int indiceEntrada, int idade);
    void addArestas(int idA, int idB);
    void verListaVertices();
    void swap(int idA, int idB, bool* troca);
    void commander(int idA);
    void meeting();
    void meetingAux(int idVertice, bool visited[], stack<int>& Stack) ;

};
```

Figura 3 – Implementação do TAD'S *Graph*.

2.3 Modularização

Foram criados além dá *main* outro arquivo, graph.hpp que apresenta a estrutura dos TAD's utilizados para a implementação e o graph.cpp que corresponde a implementação dos métodos correspondentes aos TAD's.

2.4 Código

A função *main* é responsável por criar a o Grafo dos alunos a partir de um arquivo de texto (equipe.txt) lido pela biblioteca *fstream* com as instruções para a formação da equipe, e os comandos desejados.

A função *swap* é responsável por efetuar a troca de comando entre dois membros do grupo, no entanto não é simplesmente criar uma aresta (u, v), é necessário verificar se há realmente uma ligação entre os vértices u e v, se sim, é necessário verificar se a inversão gerará um ciclo no Grafo.

Esta função auxiliar denominada Verifica Ciclos, é uma variante do algoritmo de DFS (Busca em profundidade), basicamente consiste em a partir do vértice inicial, percorrer um caminho sistematicamente até o último vértice não visitado, e só a partir disto, retornar e explorar outro caminho. Ao percorrer todas as arestas, se em algum momento passar por um vértice já visitado retorna a presença de um ciclo no Grafo.

A função *meeting* retorna ao usuário a ordem topológica do Grafo em sua atual organização, isto é a ordem de prioridade linear de cada vértice, partindo do princípio que o primeiro vértice não terá ninguém acima de sua hierarquia, caso contrário há um ciclo no grafo.

A função *commander* tem como responsabilidade a partir de determinado vértice descobrir qual o vértice que tem caminho até ele cuja chave idade é a menor, caso não haja retorna "*".

3. Analise de Complexidade:

Criação do Grafo: A criação do grafo vazio consiste apenas na instancia de um vector de vértices, criando *n* objetos, portanto tem custo linear.

Criação de Vértice: A criação de um novo Vértice também consiste na inicialização de um objeto com id, idade, e uma lista de adjacência inicialmente vazia.

Preencher lista de adjacência: A inserção de um vértice v na lista de adjacência de u é constante, pois é feita através de acesso direto pelo índice u, no entanto serão múltiplas chamadas, portanto teremos um custo também linear.

Verificação de ciclos: Baseado no algoritmo de busca em profundidade (DFS) para cada vértice teremos sua lista de adjacência verificada uma única vez, portanto esta etapa possui custo O (V+E).

Swap: A troca de comando (u, v) para (v, u) tem custo linear uma vez que é dominada pela remoção da primeira aresta, a busca de v na lista de adjacência de u é linear, precisa percorrer toda a lista até encontrar o vértice desejado. A inclusão de u na lista de adjacência de v é constante pois o acesso ao vértice v é realizado via índice.

Meeting: Baseado no algoritmo de Kahn seleciona inicialmente vértices que não possuem arestas chegando, a partir disto estabelece um caminhamento linear dos vértices vizinhos em sequência, até que todos os vértices do grafo direcionado sejam perfilados, possui custo O (V+E) uma vez que passa por todas as arestas dos vértices do grafo uma única vez e não visita o mesmo vértice mais de uma vez.

COMMANDER: Para esta função receber o id do vértice v e conferimos nos outros vértices do grafo aquele que tem v em sua lista de adjacência caso haja mais de um é tido como *commander* aquele que é mais novo.

Podemos perceber um crescimento linear no tempo de execução do programa a medida em que aumentamos o número de vértices e relacionamentos entre eles conforme gráfico e tabela abaixo:

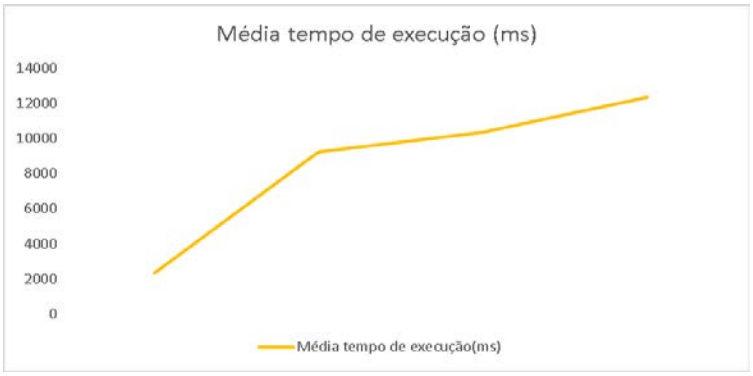

Gráfico 1 – Média de tempo de execução do programa com diversos *Datasets*.

Numero de vértices	3	7	13	21
Numero de arestas	3	8	16	23
Média tempo de execução(ms)	2335	9220	10338	12344

Tabela 1 – Teste de execução do programa com diversos *Datasets*.

4. Avaliação Experimental

Podemos concluir que:

O relacionamento hierárquico proposto para a organização do grupo é modelado por um grafo direcionado uma vez que a representação das arestas não é simétrica. Devido a este fato não é permito a presença de ciclos no grafo, porque isto representaria uma relação simétrica entre os vértices envolvidos no ciclo.

Por se tratar de uma hierarquia podemos representar o Grafo através de uma arvore tendo a raiz como os vértices que só comandam e as folhas como os comandados. No entanto podemos ter dois ou mais vértices que só comandam outros e assim teríamos uma floresta, não conseguindo representar o conjunto inteiro em uma única árvore.

5. Conclusão

Ao iniciarmos a leitura do problema proposto temos a impressão que a implementação será complexa pois trabalhar com grafos não é algo comum para a nossa realidade e esse distanciamento gera certo receio. A medida em que buscamos referencias e conseguimos entender a modelagem necessária percebemos que os TAD's necessários são de fácil implementação o Grafo é basicamente um arranjo de Vértices e cada Vértice por sua vez possui uma chave que o identifica, um atributo que o diferencia e uma lista de vizinhos.

O resultado da implementação foi bem satisfatório uma vez que com poucas funções baseadas em algoritmos já idealizados como a Ordem Topológica e a Busca em Profundidade conseguimos entregar o sistema funcionando de modo completo, além da eficácia o código também é eficiente, ao observamos a **Análise de Complexidade** o custo operacional, isto é, desconsiderando o custo de implementação da estrutura, é O (V+E).

6. Referências

KLEINBERG, J; TARDOS, E. Algorithm Design. Capítulo 3: Graphs. Editora Pearson.

FEOFILIFF, Paulo. Busca em profundidade. Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html./> Acesso em: 23 de Setembro de 2019;

PONTI, Moacir. Grafos: ordenação topológica. Disponível em:

<http://wiki.icmc.usp.br/images/9/93/Alg2_05.Grafos_ordenacaotopologica.pdf/> Acesso em:23 de Setembro de 2019

Depth First Search. Disponível em: <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>. Acesso em: 25 Setembro de 2019.

Chrono Reference C++ - CPPLUS. Disponível em: <http://www.cplusplus.com/reference/chrono/> Acesso em: 25 de Setembro de 2019.