



UNIVERSIDADE FEDERAL DE MINAS GERAIS
ENGENHARIA DE SISTEMAS

TRABALHO FINAL DE PROGRAMAÇÃO ORIENTADA OBJETOS
IMPLEMENTAÇÃO DE CLASSE PARA OPERAR GRAFOS

DISCIPLINA: PROGRAMAÇÃO ORIENTADA A OBJETOS
PROFESSORA: Raquel Mini
ALUNOS: Antonio Carlos da Anunciação
Warley França Abreu

2018019443
2018438098

Sumário

I. INTRODUÇÃO:	3
II. IMPLEMENTAÇÃO:	4
III. CONCLUSÕES E DISCUSSÃO DO TRABALHO:	15
IV. REFERENCIAS BIBLIOGRAFICAS:	16
V. CODIGO FONTE DO PROGRAMA:	17

I. INTRODUÇÃO:

Um grafo é uma estrutura composta por um conjunto de entidades e relações de adjacência entre estas entidades. É uma abstração matemática que pode representar sistemas físicos como circuitos elétricos, redes de computadores, sistemas telefônicos, redes de distribuição de água, sistemas de transporte, encadeamento de tarefas em uma linha de produção, etc. As entidades e relações em um grafo são representadas, respectivamente, por vértices e arestas, os quais podem possuir propriedades adicionais.

Nosso objetivo nesse trabalho é construir um programa que crie e execute determinadas operações em grafos, tais operações estão descritas na sequência.

- a) *Um construtor, que receberá como parâmetro um inteiro indicando o número de vértices do grafo;*
- b) *Um destrutor, que se incumbirá de fazer a desalocação de memória eventualmente utilizada na representação do grafo;*
- c) *Função para inserir uma aresta no grafo: `bool Graph::insert(const Edge&)`. A função retornará `true` se a inserção ocorrer com sucesso e `false` caso a aresta que se está tentando inserir já exista no grafo.*
- d) *Função para retirar uma aresta do grafo: `bool Graph::remove(const Edge&)`. A função retornará `true` se a remoção ocorrer com sucesso e `false` caso a aresta que se está tentando remover não exista no grafo.*
- e) *Funções para buscar o número de vértices e o número de arestas do grafo. Para que a função que retorna o número de arestas seja eficiente, é interessante que a classe mantenha um atributo interno que faça esta contagem. O atributo deve ser atualizado em todas as inserções e remoções de aresta que ocorrerem com sucesso;*
- f) *Função para verificar a existência de uma aresta do grafo: `bool Graph::edge(const Edge&) const`. A função retornará `true` se a aresta estiver presente no grafo e `false` em caso contrário.*
- g) *Função booleana para verificar se o grafo desenhado é completo.*
- h) *Função para completar o grafo desenhado.*
- i) *Função para realizar a busca em largura (Breadth First Search - BFS). Essa função deve receber o índice de um vértice e apresentar os índices dos vértices na ordem do caminhamento em largura a partir do vértice recebido como parâmetro. Este caminhamento deve ser feito apenas no componente do vértice inicial.*
- j) *Função para realizar a busca em profundidade (Depth First Search – DFS). Essa função deve receber o índice de um vértice e apresentar os índices dos vértices na ordem do caminhamento em profundidade a partir do vértice recebido como parâmetro. Este caminhamento deve ser feito em todos os componentes do grafo.*
- k) *Função para retornar o número de componentes conectados do grafo. A determinação do número de componentes conectados pode ser feita usando busca em profundidade no grafo.*
- l) *Função para encontrar o menor caminho através do Algoritmo de Dijkstra. Essa função deverá receber o índice do vértice inicial e final e retornar os vértices contidos no menor caminho bem como o comprimento desse menor caminho.*
- m) *Função para resolver o Problema do Caixeiro Viajante. Essa função deverá completar o grafo caso o mesmo não seja completo.*
- n) *Função para encontrar uma árvore geradora mínima de um grafo com peso nas arestas.*

II. IMPLEMENTAÇÃO:

O programa foi construído de tal maneira que ele deverá ser executado diretamente no prompt de comando ou terminal, e o arquivo contendo as informações iniciais do **grafo** deverá ser passado na linha de comando:

```
>grafo.exe grafo.txt
```

Executado o comando o programa cria automaticamente no grafo usando as informações do arquivo *grafo.txt* e abre sua interface:

```
*****
* Escolha o que voce quer fazer:
* 01 - Imprime Grafo
* 02 - Adicionar Aresta
* 03 - Remover Aresta
* 04 - Busca Aresta
* 05 - Verificar vertices e Arestas
* 06 - Verificar Grafo
* 07 - Completar Grafo
* 08 - BFS: Busca em Largura
* 09 - DFS: Busca em Profundidade
* 10 - Dijkstra, Menor distancia entre dois vertices
* 11 - Arvore Geradora Minima
* 12 - Caixeiro viajante
* 13 - Numero de componentes
* 50 - Salvar Grafo atual
* 99 - Salvar dados e sair
*****
```

FIG.01: Interface do Programa

Se escolhermos a opção 1 ele nos dará a matriz de adjacência do grafo referente ao arquivo de entrada, grafo com 6 vértice e 8 arestas:

```
Entre com sua opcao: 1
0 1 1 0 0 1
1 0 0 1 1 1
1 0 0 1 0 0
0 1 1 0 0 0
0 1 0 0 0 1
1 1 0 0 1 0
```

FIG.02: Matriz de adjacência – grafo 6-8

Utilizando^[2] podemos visualizar nosso grafo:

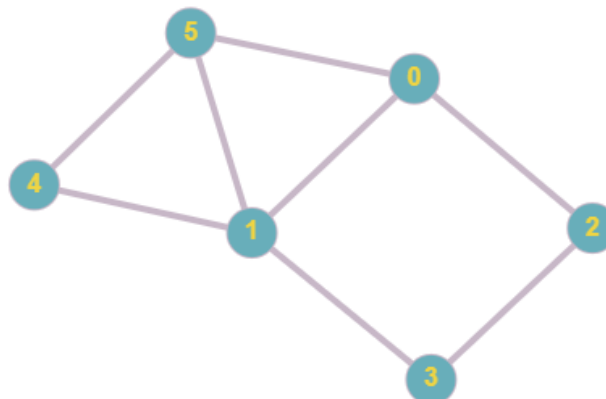


FIG.03: Grafo 6-8

Vale notar que o meio utilizado para a visualização começa a contagem dos vértices a partir de Zero, e não do Um, como é o caso do nosso programa implementado.

Opção 02 - Adicionar Aresta:

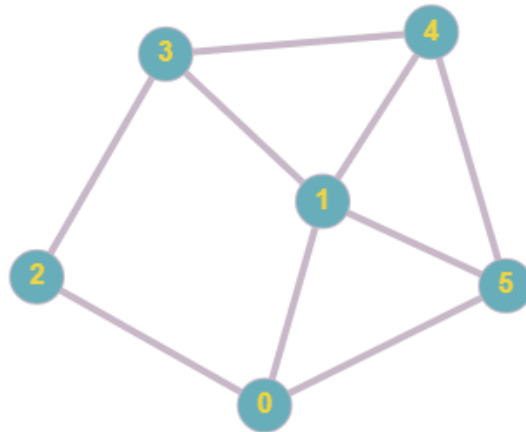


FIG.04: Adicionado aresta 3-4 no Grafo

Esta opção adiciona uma aresta no nosso grafo, seus parâmetros de entradas são os vértices e o peso, que por *default* é 1, mas pode ser utilizado qualquer valor para o peso.

Opção 04 – Busca Aresta:

```
Entre com sua opcao: 4
Entre com os vertices da aresta desejada: 1
2
Parabens, essa aresta existe!
Peso da aresta: 1
```

FIG.05: Busca Aresta 0-1

Se buscamos a aresta por exemplo 0 e 1 (1 e 2 no nosso programa), teremos como retorno uma mensagem de parabenização e o peso dessa aresta.

Opção 03 - Remover Aresta:

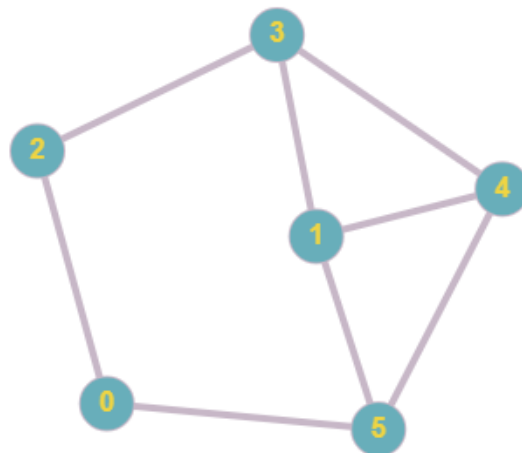


FIG.06: Removendo aresta 0-1 no Grafo

Se buscamos novamente a aresta 0 e 1 (1 e 2 no nosso programa), teremos dessa vez um retorno *false* acusando que essa aresta não existe mais.

```
Entre com sua opcao: 4
Entre com os vertices da aresta desejada: 1
2
Aresta inexistente!
```

FIG.07: Buscando aresta 0-1 no Grafo

Opção 05 – Verifica Vértices e Arestas:

```
Entre com sua opcao: 5
O grafo contem:
Arestas: 7, Vertices: 6
```

FIG.08: Vertice e Arestas do Grafo

Esta opção retorna à quantidade de vértices e arestas que nosso grafo possui.

Opção 06 – Verificar Grafo:

```
Entre com sua opcao: 6
Grafo incompleto!
Deseja completa-lo?
(1) - Sim
(2) - Nao
```

FIG.09: Verifica se um é completo

O opção 06 verifica a completude do nosso Grafo, ou seja, se todos os vértices possuem o número máximo de arestas conectados a eles, para isso ele simplesmente conta as arestas do grafo e compara com o valor máximo possível, que é $[(numero_de_vertices^2 - numero_de_vertices)/2]$. Na sequência ele nos oferece a opção de completa-lo.

Opção 10 – Dijkstra, Menor distância entre dois vertices:

```
Entre com o valor os valores dos vertices: 1
2
1 6 2 - Distancia: 2
```

FIG.10: Aplica o algoritmo Dijkstra

Aplicando o algoritmo de busca de menor distancia entre dois vértices o programa nos retorna o menor valor das somas dos pesos do caminho entre os vértices desejados e os vértices que estavam no caminho, no nosso caso os vértices 0 e 1 da FIG.06 o caminho será $0 > 5 > 1$ ($1 > 6 > 2$ no programa), e obviamente como os pesos são unitários a soma será 2.

Opção 07 – Completar Grafo:

```
Entre com sua opcao: 1
0 1 1 1 1 1
1 0 1 1 1 1
1 1 0 1 1 1
1 1 1 0 1 1
1 1 1 1 0 1
1 1 1 1 1 0
```

FIG.11: Matriz de Adjacência Completa

A opção 07 completa as arestas nulas com peso 1, por padrão, se desenharmos nosso grafo:

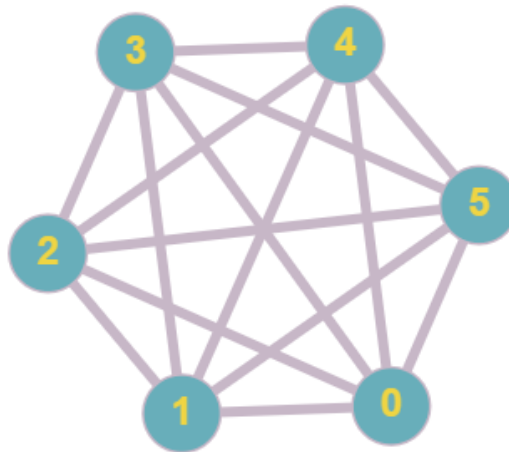


FIG.12: Grafo completo

Opção 12 – Caixeiro Viajante:

```
Entre com sua opcao: 12
1 6 5 4 3 2
```

FIG.13: Ordem dos vértices

Utilizando a opção 12 temos a solução do problema do Caixeiro Viajante, o retorno do programa é a ordem sequencial visitada, no caso da FIG.12 ficaria (0, 5, 4, 3, 2, 1).

Para demonstrar as outras opções do programa iremos utilizar um grafo maior, com 20 vértices e 30 arestas iniciais.

Imprimindo a Matriz de Adjacência:

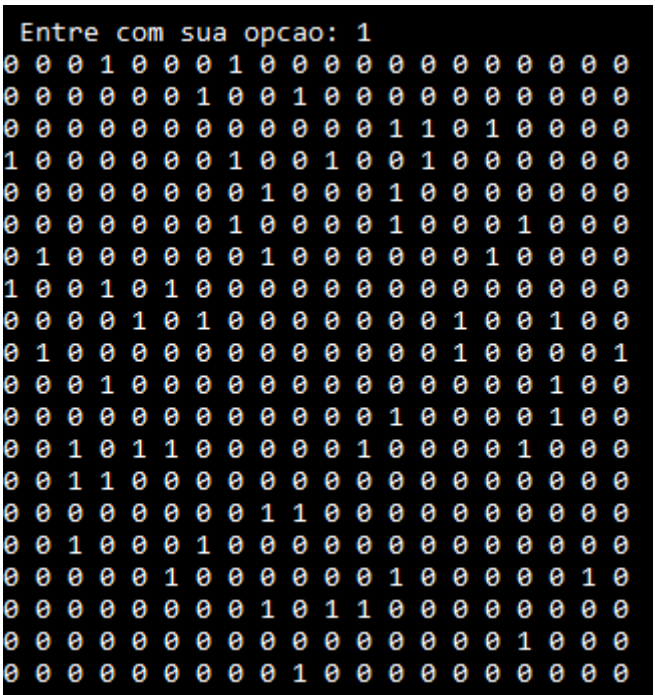


FIG.14: Matriz de adjacência – grafo 20-30

Visualizando^[2] graficamente:



FIG.15: Grafo 20-30

Adicionado uma aresta entre os vértices 1 e 20 (0 e 19 da FIG.15).

Opção 02 - Adicionar Aresta:

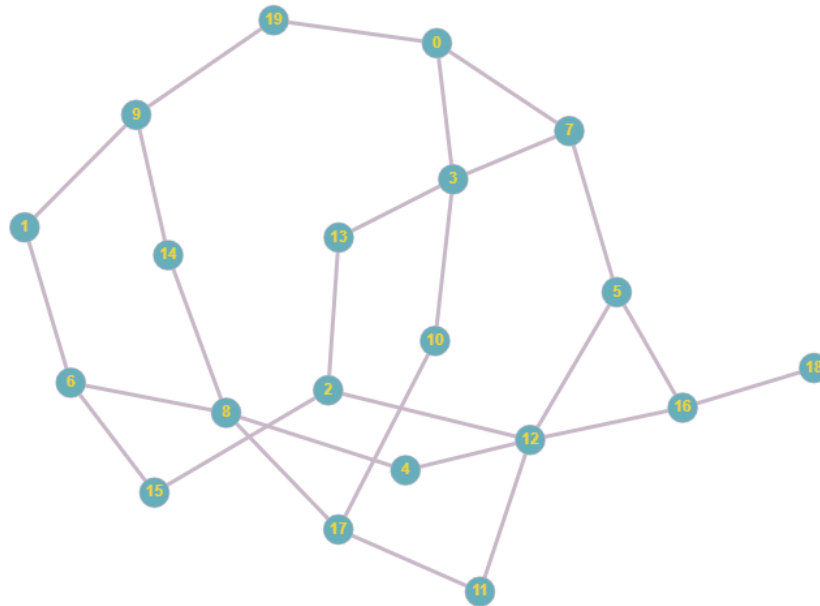


FIG.16: Adicionando Aresta 0-19

Removendo arestas 4-14(3-13) e 11-18(10-17).

Opção 03 – Remover Aresta:

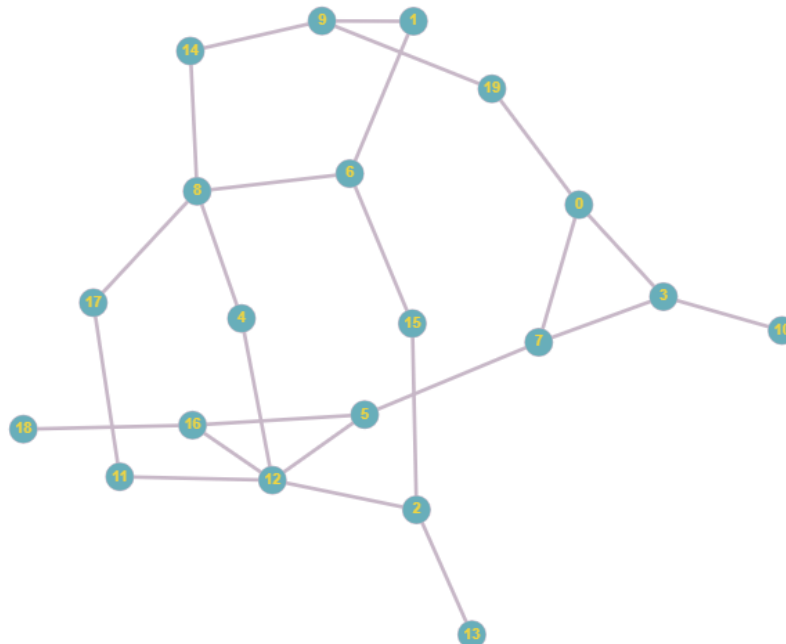


FIG.17: Removendo Arestas 3-13 e 10-17

Opção 08 – BFS: Busca em Largura:

Esta opção é melhor visualizada com grafos maiores, aqui nosso programa retorna a ordem sequencial de visitas até atingir o vértice mais distante em largura.

```
Entre com o valor do vertice: 1
1 4 8 20 11 6 10 13 17 2 15 3 5 12 19 7 9 14 16 18
```

FIG.18: Vértices Visitados

Graficamente temos:

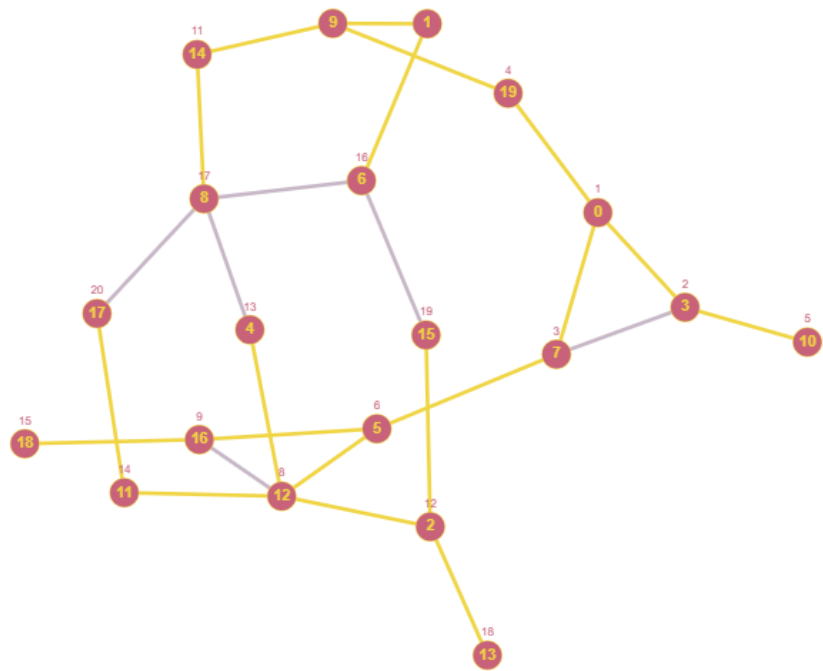


FIG.19: BFS - Vértices Visitados

Opção 09 – DFS: Busca em Profundidade:

Aqui o objetivo é retornar a ordem sequencial de visitas até atingir o vértice mais distante em profundidade.

```
Entre com o valor do vertice: 1
1 20 10 15 9 18 12 13 17 19 6 3 16 14 7 5 2 8 4 11
```

FIG.20: Vértices Visitados

Graficamente:

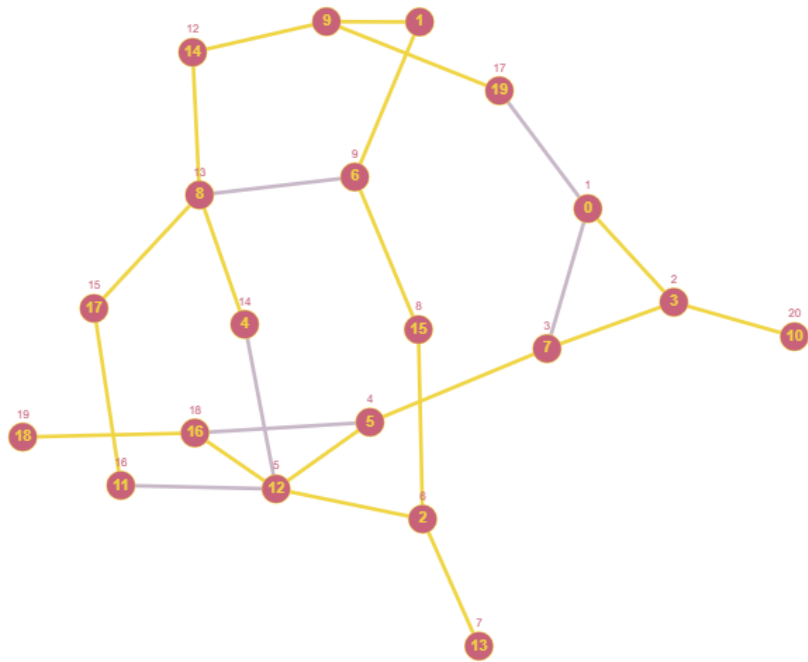


FIG.21: DFS - Vértices Visitados

Opção 10 – Dijkstra, Menor distância entre dois vertices:
 Calculo da distancia entre os vértices 1 e 14(0 e 13 da FIG.23).

```

Entre com o valor os valores dos vertices: 1
14
1 8 6 13 3 14 - Distancia: 5
  
```

FIG.22: Vértices Visitados

Graficamente:

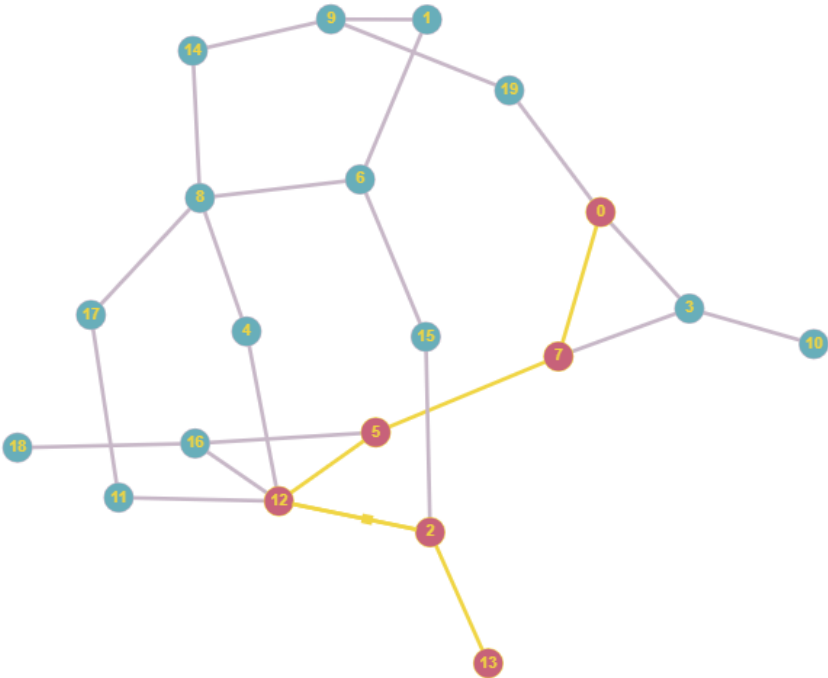


FIG.22: Dijkstra - Vértices Visitados

Opção 07 – Completar Grafo:

```

Entre com sua opcao: 1
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
  
```

FIG.23: Matriz de Adjacência Completa

Para nosso grafo de 20 vértices completo:

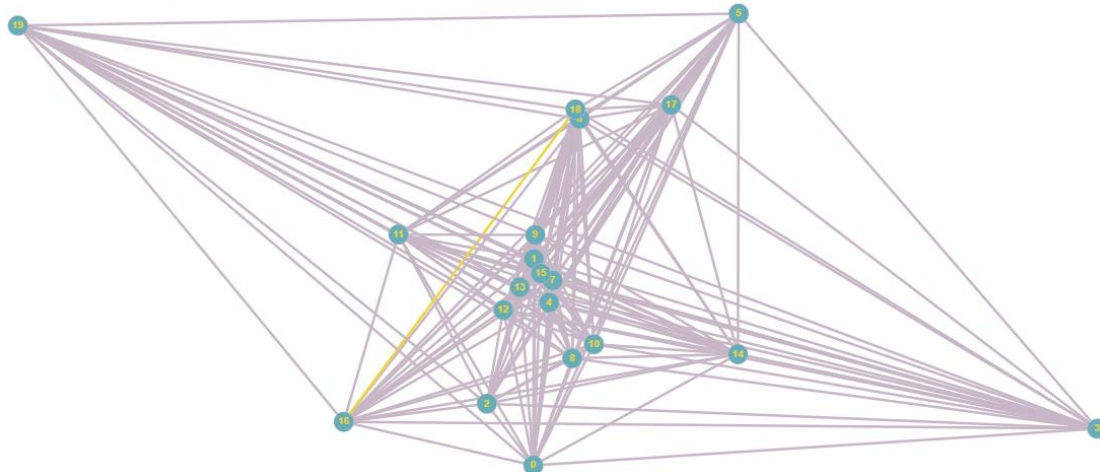


FIG.24: Grafo completo

Se verificarmos a quantidade de arestas sabemos de antemão que este grafo deverá ter $(20^2-20)/2$ arestas, ou seja:

$$\text{Number_of_edge} = 190$$

Opção 05 – Verifica Vértices e Arestas:

```
Entre com sua opcao: 5
0 grafo contem:
Arestas: 190, Vertices: 20
```

FIG.25: Vértice e Arestas do Grafo

Opção 06 – Verificar Grafo:

```
Entre com sua opcao: 6
Grafo completo!
```

FIG.26: Verifica se um é completo

As duas operações acima nos retornaram os valores esperados.

Por fim temos a opção de verificar a quantidade de componentes de um grafo, obviamente os dois exemplos cada grafo é composto de um componente único, sendo assim se aplicarmos a operação 13 ele deverá retornar 1.

Opção 13 – Numero de Componentes:

```
Entre com sua opcao: 13
1
```

FIG.27: Número de Componentes de um Grafo

Se reconstruirmos o Grafo da FIG.15:

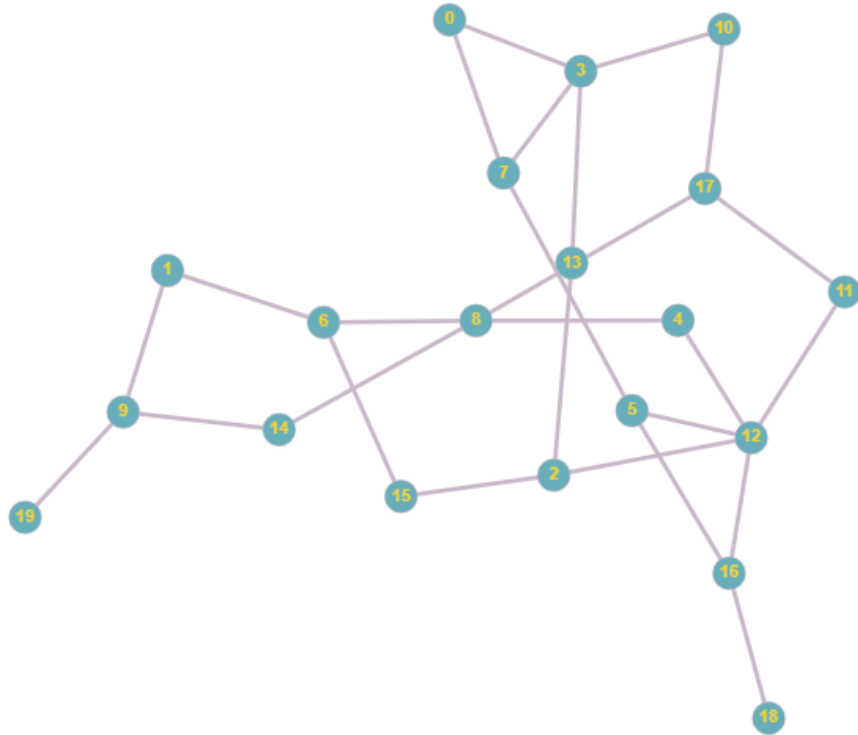


FIG.28: Grafo 20-30 reconstruído

Uma vez reconstruído iremos remover as arestas (5,7), (4,12), (11,12), (2,12) para criarmos um grafo não-conectado.

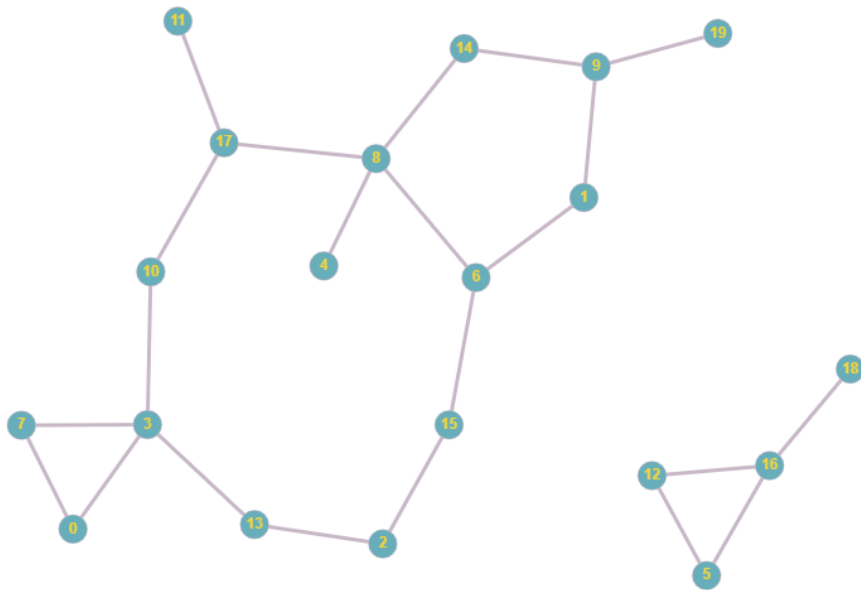
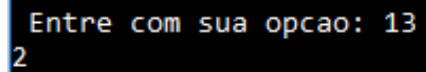


FIG.29: Grafo 20-30 Não-conecto

Se calcularmos a quantidade de componentes o programa deverá retornar 2.

Opção 13 – Numero de Componentes:



```
Entre com sua opcao: 13
2
```

FIG.30: Número de Componentes de um Grafo

Conforme esperado o programa retornou o valor correto.

Por fim cada operação feita no grafo gera um arquivo .txt baseado no que foi feito, com valores atualizados da matriz de adjacência, caminhos de vértices para os problemas de buscas ou Caixeiro Viajante. E utilizando a **Opção 99** o programa encerra, salvando os arquivos automaticamente

III. CONCLUSÕES E DISCUSSÃO DO TRABALHO:

Após implementação e apresentação dos resultados e funcionamento do programa concluímos que o programa funcionou corretamente para a maioria dos casos, aqui vale ressaltar que houve grandes desafios para o Problema do Caixeiro Viajante, para funcionar com grafos de 20 ou mais vértices, o programa não apresentou resposta em tempo hábil para análise.

Para a implementação dos algoritmos de buscas foi utilizado artifícios não vistos em salas, como manipulação de pilha e fila de execução, tais algoritmos foram encontrados prontos, havendo a necessidade de pouco ou mínima alterações para funcionarem de acordo com nossas demandas.

Sendo assim, damos como encerrado com muita satisfação e aprendizado este trabalho.

IV. REFERENCIAS BIBLIOGRAFICAS:

[1] – Programação em C++, Algoritmos, estruturas de dados e objetos, Luis Joyanes Aguilar, São Paulo, McGraw-Hill, 2008

[2] - <https://graphonline.ru/en/>

[3] - https://www.ime.usp.br/~pf/algoritmos_para_grafos/

V. CODIGO FONTE DO PROGRAMA:

```
Graph.h

#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <algorithm>
#include <string>
#include <cstdio>
#include <queue>
#include <stack>
#include <limits>
#include <fstream>
#include <stdlib.h>

typedef struct Edge {
    int vertex_A, vertex_B; // Define os valores dos vertices conectados a uma aresta.
    int weight; // Define o peso de uma aresta.
} Edge;

class Graph {
private:
    int num_vertex, num_edge; // Atributos privados da Classe, numero de vértices e arestas
    int** adjacency_matrix; // Matriz de adjacência da classe, registra os pesos entre as arestas

    bool BFS(const int&, bool[]) const; // Metodo auxiliar das funções de buscas
    void permutation(const Graph&, int*, int*, int&, const int&) const; // Metodo auxiliar das funções de buscas

public:
    Graph(int);
    ~Graph();

    int number_of_edge() const; // Metodo retorna o numero de arestas
    int number_of_vertex() const; // Metodo retorna o numero de vertices

    int weight(int, int) const; // Metodo retorna o peso de uma aresta

    bool insert(const Edge&); // Metodo insere uma aresta
    bool remove(const Edge&); // Remove uma aresta
    bool verify_edge(const Edge&) const; // Metodo verifica se uma aresta é valida
    bool verify_graph() const; // Metodo verifica se um grafo esta completo
    void complete_graph(); // Metodo complete um grafo
    Graph fill_graph(const int&) const; // Metodo auxiliar para completar grafo dentro do metodo do Caixeiro Viajante
    void BFS(const int&) const; // Metodo busca na largura
    void DFS(const int&) const; // Metodo busca na profundidade
    void print() const; // Metodo imprime matriz de adjcencia na tela
    void save(); // Metodo salva matriz de adjacencia
    int number_of_component() const; // Metodo retorna o numero de componentes de um grafo
    void Dijkstra(int, int, int**, int&, int&); // Metodo calcula a menor distancia entre dois vertices
    Graph TSM(const int&, const int&) const; // Metodo auxiliar par ao Caixeiro Viajante
    Graph MST() const; // Metodo auxiliar par ao Caixeiro Viajante
};

void menu(int&); // Funcao para apresentar o Menu da programa

#endif // GRAPH_H
```

Graph.cpp

```
#include "Graph.h"
using namespace std;

void menu(int &opcao) {
    cout << "\n *****\n";
    cout << " * Escolha o que voce quer fazer: * \n";
    cout << " * 01 - Imprime Grafo * \n";
    cout << " * 02 - Adicionar Aresta * \n";
    cout << " * 03 - Remover Aresta * \n";
    cout << " * 04 - Busca Aresta * \n";
    cout << " * 05 - Verificar vertices e Arestas * \n";
    cout << " * 06 - Verificar Grafo * \n";
    cout << " * 07 - Completar Grafo * \n";
    cout << " * 08 - BFS: Busca em Largura * \n";
    cout << " * 09 - DFS: Busca em Profundidade * \n";
    cout << " * 10 - Dijkstra, Menor distancia entre dois vertices * \n";
    cout << " * 11 - Arvore Geradora Minima * \n";
    cout << " * 12 - Caixeiro viajante * \n";
    cout << " * 13 - Numero de componentes * \n";
    cout << " * 50 - Salvar Grafo atual * \n";
    cout << " * 99 - Salvar dados e sair * \n";
    cout << " *****\n";
    cout << " Entre com sua opcao: ";
    cin >> opcao;
};

Graph::Graph(int num_vertex = 0) {
    this->num_vertex = num_vertex;
    this->num_edge = 0;

    adjacency_matrix = new int*[num_vertex];
    for(int i = 0; i < num_vertex; i++) adjacency_matrix[i] = new int[num_vertex]();
}

Graph::~~Graph() {
    for (int i = 0; i < num_vertex; ++i) delete[] adjacency_matrix[i];
    delete[] adjacency_matrix;
}

int Graph::number_of_edge() const {
    return num_edge;
}

int Graph::number_of_vertex() const {
    return num_vertex;
}

int Graph::weight(int vertex_a, int vertex_b) const {
    if(vertex_a > num_vertex || vertex_a < 1 || vertex_b > num_vertex || vertex_b < 1) return -1;
    else return adjacency_matrix[vertex_a-1][vertex_b-1];
}

bool Graph::insert(const Edge& edge) {
    if(verify_edge(edge) == false) {
        cout << "\nAresta nao existente!\n";
        return false;
    }
    else if(adjacency_matrix[edge.vertex_A-1][edge.vertex_B-1] != 0){
        adjacency_matrix[edge.vertex_A-1][edge.vertex_B-1] = edge.weight;
        adjacency_matrix[edge.vertex_B-1][edge.vertex_A-1] = edge.weight;
        return true;
    }
    else {
        num_edge++;
        adjacency_matrix[edge.vertex_A-1][edge.vertex_B-1] = edge.weight;
```

```

        adjacency_matrix[edge.vertex_B-1][edge.vertex_A-1] = edge.weight;
        return true;
    }
}

bool Graph::remove(const Edge& edge) {
    if(verify_edge(edge) == false) {
        cout << "\nAresta nao existe!\n";
        return false;
    }

    else {
        num_edge--;
        adjacency_matrix[edge.vertex_A-1][edge.vertex_B-1] = 0;
        adjacency_matrix[edge.vertex_B-1][edge.vertex_A-1] = 0;
        cout << "\nAresta removida com sucesso!\n";
        return true;
    }
}

bool Graph::verify_edge(const Edge& edge) const {
    if(edge.vertex_A < 1 || edge.vertex_B < 1 || edge.vertex_A > num_vertex || edge.vertex_B > num_vertex) return false;
    else return true;
}

bool Graph::verify_graph() const {
    if((number_of_edge() < ((number_of_vertex()*number_of_vertex()) - number_of_vertex())/2)) {
        cout << "\nGrafo incompleto!\n";
        return false;
    }
    else {
        cout << "\nGrafo completo!\n";
        return true;
    }
}

void Graph::complete_graph() {
    Edge aux_edge;
    for(int i = 0; i < num_vertex; i++){
        aux_edge.vertex_A = 1 + i;
        for(int j = 0; j < num_vertex; j++)
            if((adjacency_matrix[i][j] == 0) && (i != j)) {
                aux_edge.vertex_B = 1 + j;
                aux_edge.weight = 1;
                insert(aux_edge);
            }
    }
}

Graph Graph::fill_graph(const int& weight = 1) const {
    Graph grafo(num_vertex);
    Edge edge;

    for(int i = 0; i < num_vertex; i++) {
        edge.vertex_A = i+1;
        for(int j = 0; j < num_vertex; j++) {
            edge.vertex_B = j+1;
            edge.weight = adjacency_matrix[i][j];
            if(i != j && edge.weight == 0) edge.weight = weight;
            else grafo.insert(edge);
        }
    }
    return grafo;
}

void Graph::print() const {
    for(int i = 1; i <= num_vertex; i++) {

```

```

        for(int j = 1; j <= num_vertex; j++) cout << weight(i, j) << " ";
        cout << endl;
    }
}

void Graph::save() {
    ofstream matrix;
    matrix.open("adjacency_matrix.txt", ios::out);
    for(int i = 1; i <= num_vertex; i++) {
        for(int j = 1; j <= num_vertex; j++) matrix << weight(i, j) << " ";
        matrix << endl;
    }
    matrix.close();
}

void Graph::BFS(const int& vertex) const {
    if(vertex < 1 || vertex > num_vertex) {
        cout << "\nVertice inicial invalido!\n";
        return;
    }
    queue<int> q; // Cria uma fila
    bool* visited = new bool[num_vertex](); // Indica as posicoes ja visitadas
    q.push(vertex-1); // Insere o vertice inicial na fila
    visited[vertex-1] = true; // Indica que o vertice inicial foi visitado

    int a, i; // Elemento atual a ser lido
    ofstream bfs;

    bfs.open("BFS.txt", ios::out);

    // Enquanto a fila nao estiver vazia, execute o codigo
    do {
        // Remove um elemento da fila
        a = q.front();
        q.pop();

        cout << a+1 << " ";
        bfs << a+1 << " ";

        for(i = 0; i < num_vertex; i++) {
            // Verifica se o vertice ja foi visitado e se tem peso entre os vertices
            if(visited[i] == false && adjacency_matrix[a][i] != 0) {
                // Insere o vertice na fila
                q.push(i);
                // Indica que o vertice foi visitado
                visited[i] = true;
            }
        }
    } while(!q.empty());

    cout << endl;
    bfs << endl;

    bfs.close();

    delete[] visited;
}

void Graph::DFS(const int& vertex) const {
    if(vertex < 1 || vertex > num_vertex) return;

    stack<int> s; // Cria uma pilha
    bool* visited = new bool[num_vertex](); // Indica as posicoes ja visitadas
    s.push(vertex-1); // Insere o vertice inicial na pilha
    visited[vertex-1] = true; // Indica que o vertice inicial foi visitado

    int a, i; // Elemento atual a ser lido

```

```

        ofstream dfs;

        dfs.open("DFS.txt", ios::out);

// Enquanto a pilha nao estiver vazia, execute o codigo
do {
    // Remove um elemento da pilha
    a = s.top();
    s.pop();

    cout << a+1 << " ";
    dfs << a+1 << " ";

    for(i = 0; i < num_vertex; i++) {
        // Verifica se o vertice ja foi visitado e se tem peso entre os vertices
        if(visited[i] == false && adjacency_matrix[a][i] != 0) {
            // Insere o vertice na pilha
            s.push(i);

            // Indica que o vertice foi visitado
            visited[i] = true;
        }
    }
    } while(!s.empty());

    cout << endl;
    dfs << endl;
    dfs.close();
    delete[] visited;
}

bool Graph::BFS(const int& vertex, bool visited[]) const {
    queue<int> q; // Cria uma fila
    q.push(vertex); // Insere o vertice inicial na fila
    visited[vertex] = true; // Indica que o vertice inicial foi visitado
    bool con = false; // Indica se houve conexao de componentes

    int a, i; // Elemento atual a ser lido

// Enquanto a fila nao estiver vazia, execute o codigo
do {
    // Remove um elemento da fila
    a = q.front();
    q.pop();

    for(i = 0; i < num_vertex; i++) {
        // Verifica se o vertice ja foi visitado e se tem peso entre os vertices
        if(visited[i] == false && adjacency_matrix[a][i] != 0) {
            // Insere o vertice na fila
            q.push(i);

            // Indica que o vertice foi visitado
            visited[i] = true;

            // Indica que houve conexao de componentes
            con = true;
        }
    }
    } while(!q.empty());
    return con; // Indica se houve conexao de componentes
}

int Graph::number_of_component() const {
    // Numero de componentes do grafo
    int n = 0;

    // Indica as posicoes ja visitadas
    bool* visited = new bool[num_vertex]();

    // Verifica em todos vertices suas conexoes, sem repetir conexoes ja feitas

```

```

for(int i = 0; i < num_vertex; i++)
    if(!visited[i] && BFS(i, visited)) n++;

delete[] visited;
// Mostra o numero de componentes conectados
return n;
}

// Algoritmo de Dijkstra
void Graph::Dijkstra(int vertex_a, int vertex_b, int** path, int& vnum, int& lenght) {

    // Verifica se existe os vertices
    if(vertex_a < 1 || vertex_b < 1 || vertex_a > num_vertex || vertex_b > num_vertex) return;

    // Distancia da fonte para posicao
    int* d = new int[num_vertex]();

    // Vertice anterior
    int* prev = new int[num_vertex]();

    // Indica as posicoes ja visitadas
    bool* visited = new bool[num_vertex]();

    // Numero de vertices visitados
    int numvisited = 0;

    // Auxiliar de minima distancia
    int mindist, minpos;
    int i;

    // Inicializacao das variaveis de controle
    for(i = 0; i < num_vertex; i++) {
        d[i] = numeric_limits<int>::max();
        prev[i] = -1;
    }
    // Distancia do inicio para o inicio
    d[vertex_a-1] = 0;

    while(numvisited != num_vertex) {
        // Reinicia a minima distance
        mindist = numeric_limits<int>::max();

        // Descobre o vertice de distancia minima da atual
        for(i = 0; i < num_vertex; i++) {
            if(!visited[i]) {
                if(mindist >= d[i]) {
                    mindist = d[i];
                    minpos = i;
                }
            }
        }

        // Indica que a posicao foi visitada
        visited[minpos] = true;
        numvisited++;

        // Atualiza a distancia dos vizinhos do vertice atual
        for(i = 0; i < num_vertex; i++) {
            if(!visited[i] && adjacency_matrix[minpos][i] != 0) {
                mindist = d[minpos] + adjacency_matrix[minpos][i];

                if(mindist < d[i]) {
                    d[i] = mindist;
                    prev[i] = minpos;
                }
            }
        }
    }
}

```

```

// Inicializa as variaveis de saida
*path = new int[num_vertex];

// Comeca do fim
minpos = vertex_b-1;

// Inicializa a distancia como 0
lenght = d[minpos];

// Numero de vertices inicialmente como 0
vnum = 0;

// Verifica qual a distancia
for(vnum = 0; minpos != -1; vnum++) {
    minpos = prev[minpos];
}
// Comeca do fim
minpos = vertex_b-1;
for(i = 0; i < vnum; i++) {
    (*path)[vnum-i-1] = minpos+1;
    minpos = prev[minpos];
}

delete[] d;
delete[] prev;
delete[] visited;
}

// Problema do caixeiro viajante
Graph Graph::TSM(const int& start, const int& weight) const {
    // Grafo de solucao do problema
    Graph grafo (num_vertex);

    // Posicao inicial invalida
    if(start < 1 || start > num_vertex ) return grafo;

    // Grafo auxiliar
    Graph aux_grafo = fill_graph(weight);

    // Maior distancia inicialmente e infinita
    int ldistance = numeric_limits<int>::max();

    // Vetor auxiliar de caminho
    int* pvector = new int[num_vertex+1];

    // Vetor de menor caminho
    int* lpath = new int[num_vertex+1];

    Edge edge;
    int i;

    // Primeira e ultima posicao e a cidade inicial
    pvector[0] = start;
    lpath[0] = start;
    pvector[num_vertex] = start;
    lpath[num_vertex] = start;

    // Armazena o caminho inicial
    for(i = 1; i < num_vertex; i++) {
        if(i >= start) pvector[i] = i+1;
        else pvector[i] = i;
    }
    // Faz a permutacao do vetor pvector
    permutation(aux_grafo, pvector, lpath, ldistance, num_vertex);

    // Armazena o caminho e distancia no grafo de saida

```

```

    for(i = 0; i < num_vertex; i++) {
        edge.vertex_A = lpath[i];
        edge.vertex_B = lpath[i+1];
        edge.weight = adjacency_matrix[edge.vertex_A-1][edge.vertex_B-1];
        grafo.insert(edge);
    }
    delete[] pvector;
    delete[] lpath;
    return grafo;
}

void Graph::permutation(const Graph& grafo, int* pvector, int* lpath, int& ldistance, const int& m) const {
    int i;

    if(m == 2) {
        int distance = 0;
        // Armazena as distancias entre as cidades no caminho dado pelo vetor
        for(i = 0; i < num_vertex; i++) distance += grafo.weight(pvector[i], pvector[i+1]);

        // Se a distancia calculada for menor que a armazenada, renova o valor
        if(distance < ldistance) {
            for(i = 1; i < num_vertex; i++) lpath[i] = pvector[i];
            ldistance = distance;
        }
    }
    else {
        for (i = 1; i < m; i++) {
            permutation(grafo, pvector, lpath, ldistance, m-1);
            if (m % 2 == 1) swap(pvector[1], pvector[m-1]);
            else swap(pvector[i], pvector[m-1]);
        }
    }
}

// Arvore Geradora Minima pelo algoritmo de Prim
Graph Graph::MST() const {
    Graph grafo (num_vertex);
    bool* visited = new bool[num_vertex](); // Indica as posicoes ja visitadas
    visited[0] = true; // Primeiro vertice ja visitado

    // Vertice aresta com menor peso verificado
    Edge lowerededge;
    int i, j, k;
    // Checa todos vertices
    for (i = 0; i < num_vertex; i++)
    {
        // Reseta o peso
        lowerededge.weight = numeric_limits<int>::max();
        for (j = 0; j < num_vertex; j++) {
            if(visited[j]) {
                // Verifica os pesos dos vertices conectados
                for (k = 0; k < num_vertex; k++) {
                    if(!visited[k]) {
                        // Se o peso entre dados vertices for menor que o armazenado, troca por este valor
                        int weight = adjacency_matrix[j][k];
                        if(weight != 0 && weight < lowerededge.weight) {
                            lowerededge.vertex_A = j+1;
                            lowerededge.vertex_B = k+1;
                            lowerededge.weight = weight;
                        }
                    }
                }
            }
        }
    }

    // Se encontrou um peso diferente de 0 entre os vertices
    if(loweredge.weight != numeric_limits<int>::max()) {

```



```

        visited[loweredge.vertex_B-1] = true; // Indica que o vertice ja foi visitado
        grafo.insert(loweredge); // Insere o vertice na arvore geradora minima
    }
}
delete[] visited;
return grafo;
}

```

Main.cpp

```

#include <iostream>
#include "Graph.h"

using namespace std;

int main(int argc, char *argv[]) {
    ifstream entrada;
    streambuf *oldbuff = cin.rdbuf();

    int *path, vnum, lenght, i, j, vertex_A, vertex_B, num_vertex, num_edge;
    int opcao = 0;
    bool operacao;

    //_____
    //CARREGANDO OS DADOS DO ARQUIVO PARA A MEMORIA:
    if(argc > 1) {
        entrada.open(argv[1]);
        if(!entrada.good()) {
            cout << "O arquivo \"" << argv[1] << "\" nao existe!\n";

            return 0;
        }
        cin.rdbuf(entrada.rdbuf());
    }
    else {
        cout << "Nenhum arquivo foi passado!\n";

        return 0;
    }

    do {
        cin >> num_vertex >> num_edge;
        if(num_vertex < 1) {
            cout << "Numero de vertices invalido\n";
            cout << num_vertex << " " << num_edge;
            return 0;
        }
        else if(num_edge < 0) {
            cout << "Numero de arestas invalido\n";
            cout << num_vertex << " " << num_edge;
            return 0;
        }
    } while(num_vertex < 1 || num_edge < 0);

    Graph *grafo = new Graph(num_vertex);
    Edge edge;
    for(i = 0; i < num_edge; i++) {
        cin >> edge.vertex_A >> edge.vertex_B >> edge.weight;
        grafo->insert(edge);
    }

    if(argc > 1) {
        entrada.close();
        cin.rdbuf(oldbuff);
    }

    //_____

```

```

//FIM DO CARREGAMENTO

//_____
//LOOP DO MENU INICIAL

while( opcao != 99 ) {
    operacao = false;

    if( opcao == 0 ) {
        system("cls");
    }

    else if( opcao == 1 ) {
        grafo->print();
    }

    else if( opcao == 2 ) {
        Edge aux_edge;
        cout << "\n Entre com os vertices e peso da aresta que deseja adicionar: ";
        while( operacao == false ) {
            cin >> aux_edge.vertex_A >> aux_edge.vertex_B >> aux_edge.weight;
            if(grafo->verify_edge(edge) == true) {
                operacao = grafo->insert(aux_edge);
            }
            else cout << "\nAresta nao existe, entre com uma aresta existente!\n";
        }
        cout << "\nPeso da aresta atualizado!\n";
    }

    else if( opcao == 3 ) {
        cout << "\n Entre com a aresta que deseja remover: ";
        while( operacao == false ) {
            cin >> edge.vertex_A >> edge.vertex_B;
            if(grafo->verify_edge(edge) == true) operacao = grafo->remove(edge);
            else cout << "\nAresta nao existe, entre com uma aresta existente!\n";
        }
        cout << "\nAresta removida!";
    }

    else if( opcao == 4 ) {
        cout << "Entre com os vertices da aresta desejada: ";
        cin >> vertex_A >> vertex_B;
        if(grafo->weight(vertex_A, vertex_B) <= 0) cout << "\nAresta inexistente!\n";
        else {
            cout << "\nParabens, essa aresta existe!" << endl;
            cout << "Peso da aresta: " << grafo->weight(vertex_A, vertex_B) << endl;
        }
    }

    else if( opcao == 5 ) {
        cout << "\nO grafo contem:\n" << "Arestas: " << grafo->number_of_edge()
        << ", Vertices: " << grafo->number_of_vertex();
        cout << endl;
    }

    else if( opcao == 6 ) {
        if( grafo->verify_graph() == false) {
            cout << "\nDeseja completa-lo? \n(1) - Sim \n(2) - Nao\n";
            while( opcao != 1 || opcao != 2 ) {
                cin >> opcao;
                if( opcao == 1 ) {
                    grafo->complete_graph();
                    break;
                }
                else if( opcao == 2 ) break;
                else cout << "\nOpcao invalida!";
            }
        }
    }
}

```

```

    }
}

else if( opcao == 7 ) {
    int aux_vertex = grafo->number_of_vertex();
    if((((aux_vertex*aux_vertex) - aux_vertex)/2) == grafo->number_of_edge()) cout << "\nEste
grafo ja esta completo!\n";
    else grafo->complete_graph();
}

else if( opcao == 8 ) {
    int vertex;
    cout << "\nEntre com o valor do vertice: ";
    cin >> vertex;
    grafo->BFS(vertex);
}

else if( opcao == 9 ) {
    int vertex;
    cout << "\nEntre com o valor do vertice: ";
    cin >> vertex;
    grafo->DFS(vertex);
}

else if( opcao == 10 ) {
    ofstream dijkstra;
    dijkstra.open("dijkstra.txt", ios::out);
    int vertex_1, vertex_2;
    cout << "\nEntre com o valor os valores dos vertices: ";
    cin >> vertex_1 >> vertex_2;
    grafo->Dijkstra(vertex_1, vertex_2, &path, vnum, lenght);
    for(i = 0; i < vnum; i++) {
        cout << path[i] << " ";
        dijkstra << path[i] << " ";
    }
    cout << "- Distancia: " << lenght << endl;
    dijkstra << "- Distancia: " << lenght << endl;
    dijkstra.close();
}

else if( opcao == 11 ) {
    grafo->MST().BFS(1);
}

else if( opcao == 12 ) {
    grafo->TSM(1, 30).DFS(1);
}

else if( opcao == 13 ) {
    cout << grafo->number_of_component() << endl;
}

else if( opcao == 50 ) {
    grafo->save();
    cout << "\nGrafo salvo no arquivo de texto!\n";
}

else {
    cout << "\nOxe! Entre com um numero dentre os das opcoes do Menu!\n";
}
menu(opcao);
}

grafo->save();

//_

```

```
//FIM DO PROGRAMA

    system("pause");
delete grafo;
return 0;
}
```

Arquivos de Entradas	
Grafo 6 vértices “ <i>grafo.txt</i> ”	Grafo 20 vértices “ <i>grafo.txt</i> ”
6 8	20 30
1 2 1	9 5 1
1 3 1	8 6 1
2 5 1	4 14 1
4 3 1	14 3 1
5 6 1	19 17 1
6 1 1	10 15 1
6 2 1	10 20 1
2 4 1	10 2 1
	18 12 1
	13 3 1
	15 9 1
	9 7 1
	13 5 1
	1 4 1
	6 13 1
	11 4 1
	16 7 1
	17 6 1
	3 16 1
	11 18 1
	12 13 1
	8 4 1
	7 2 1
	13 3 1
	8 1 1
	2 10 1
	17 13 1
	3 14 1
	3 14 1
	9 18 1