Trabalho Programação Orientada a Objetos

- 2017089014 - Alex Souza
- 2017001320 - Raphael Ribeiro

Sumário

Introdução

Visão Geral

Trabalho final da disciplina Programação Orientada a Objetos ministrado pela professora Raquel Mini no primeiro semestre de 2019. O software desenvolvido tem como objetivo, implementar técnicas comuns de manipulação e busca em grafos. Criado com base na pesquisa feita pelos alunos envolvidos, o software possui um script principal que efetua a demonstração das diversas funcionalidades solicitadas pelo enunciado, conforme descrito em cada ponto onde os requisitos foram atendidos. Cada um dos requisitos é listado a segir, no tópico Requistos:

Requisitos

> a) Um construtor, que receberá como parâmetro um inteiro indicando o número de vértices do grafo;

Resolucão no arquivo Graph.cpp

```
/**
 * Construtor do grafo. Recebe um inteiro que será usado para determinar o
tamanho da matriz de adjacência (quadrada).
 *
 * a) Um construtor, que receberá como parâmetro um inteiro indicando o
número de
 *    vértices do grafo;
```

```
 *
 * @param int &numeroDeVertices - O número de vértices do qual o grafo é
composto.
 */
```

> b) Um destrutor, que se incumbirá de fazer a desalocação de memória eventualmente utilizada na representação do grafo;

Resolucão no arquivo Graph.cpp

```
/**
 * Destrutor da classe.
 *
 * 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.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que possibilita a inclusão de uma aresta entre dois vértices do
grafo. Preencherá a matriz de adjacëncia.
 *
 * 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.
 *
 * @param Edge &edge - Uma instäncia de Edge que encapsula as informações
de uma aresta a ser representada 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.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que remove uma aresta definida anteriormente 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.
 *
 * @param Edge &edge - Uma instância de Edge que encapsula as informações
de uma aresta a ser removida do 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;

Resolucão no arquivo Graph.cpp

```
/**
 * Método estático que retornará o número total de arestas criadas em todos
os grafos repersentados por esta classe (Graph).
 *
 * 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;
 *
 * @TODO: Revisar o método e o atributo não deveriam ser membros privados e
não estáticos.
 *
 */
```

> 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.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que indica se uma aresta existe no grafo.
 *
 * 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.
 *
 * @param Edge &edge - Uma instäncia de Edge que encapsula as informações
de uma aresta a ser removida do grafo.
 *
 */
```

> g) Função booleana para verificar se o grafo desenhado é completo.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que indica se o grafo é completo (todos os vértices possuiem
arestas entre si).
 *
 * g) Função booleana para verificar se o grafo desenhado é completo.
 *
 */
```

> h) Função para completar o grafo desenhado.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que completa a matriz de adjacência adicionando arestas em todos
os vertices para todos os outros.
 *
 * 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.

Resolucão no arquivo Graph.cpp

```
/**
 * Efetua o cálculo da (Breath First Search – BFS) a partir de um vértice
para todos os outros. Imprime
```

```
 * todas as ligações do vértice fornecido como argumento.
 *
 * 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.
 *
 * @param char &vertex - o vértice a partir do qual se deseja efetuar a
busca
 *
 */
```

> 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.

Resolucão no arquivo Graph.cpp

```
/**
 * Efetua o cálculo da (Depth First Search – DFS) a partir de um vértice
para todos os outros.
 *
 * 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.
 *
 * @param char &vertex - o vértice a partir do qual se deseja efetuar a
busca
 *
 */
```

> 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.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que fornece o número total de conexões presentes no 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.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que calcula o caminho pelo algoritmo de Dijkstra.
 *
 * 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.
 *
 * @param char &initialVertex - O vértice de partida para o cálculo.
 * @param char &finalVertex - O vértice de destino.
 *
 */
```

> m) Função para resolver o Problema do Caixeiro Viajante. Essa função deverá completar o grafo caso o mesmo não seja completo.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que resolve o problema do caixeiro viajante.
 *
 * m) Função para resolver o Problema do Caixeiro Viajante. Essa função
deverá completar o
 *    grafo caso o mesmo não seja completo.
 *
 * @param char &vertex - O vértice de partida.
 *
 */
```

> n) Função para encontrar uma árvore geradora mínima de um grafo com peso nas arestas.

Resolucão no arquivo Graph.cpp

```
/**
 * Método que produz a árvore geradora mínima do grafo.
 *
 * n) Função para encontrar uma árvore geradora mínima de um grafo com peso
nas
 *    arestas.
 *
 */
```

> Crie um programa que, utilizando a classe acima, leia em um arquivo o número de vértices de um grafo, construa o grafo, e em seguida leia do mesmo arquivo pares de inteiros que definem as arestas do grafo. As arestas lidas devem ser adicionadas ao grafo. Teste as demais funções da interface da classe.

Resolucão no arquivo main.cpp

```
/**
 * Funcão que efetua a inclusão das arestas por meio do arquivo .csv
 */
```

##Código Fonte

Abaixo, o código fonte de todos os arquivos gerados.

main.cpp

```
#ifndef MAIN
#define MAIN

#include <iostream>
#include "Graph.cpp"
#include "Edge.cpp"
#include "CharUtil.cpp"

#define NUMERO_DE_VERTICES 5
#define NOME_DO_ARQUIVO_DE_DADOS "data/vertex.csv"
#define MSG_ARESTA_ADICIONADA "Aresta adicionada com sucesso"
#define MSG_ARESTA_REMOVIDA "Não foi possíve remover a aresta"

using namespace std;

/**
 * Declarações de funções de controle do script.
 */
```

```
/**
 * Função que efetua a inclusão de algumas arestas, por meio do método
"Graph::insert", para que seja feito um teste das diversas
 * funcionalidades da classe "Graph". Especificamente, essa função efetua o
teste da funcionalidade
 * de inclusão de arestas no grafo.
 *
 */
void inserirEdges(Graph &graph);

/**
 * Funcão que efetua a inclusão das arestas por meio do arquivo .csv
 */
void inserirEdgesCsv(Graph &graph);

/**
 * Função utilitária que efetua a remoção de algumas arestas adicionadas
anteriormente
 * pela função "inserirEdges" para teste da funcionalidade de remoção,
implementada na classe "Graph"
 *
 */
void removerEdges(Graph &graph);

/**
 * Função utilitária que efetua a BFS para todos os vértices do grafo.
 *
 */
void exibirBfs(Graph &graph);

/**
 * Função utilitária que efetua a DFS para todos os vértices do grafo.
 *
 */
void exibirDfs(Graph &graph);

/**
 * Função que efetuará o disparo dos cálculos do algoritmo de Dijkstra de
todos, para todos os vértices.
 */
void exibirDijkstra(Graph &graph);

/**
 * Funcão que efetuará o cálculo do problema do caixeiro viajante a partir
de todos os vértices.
 */
void exibirTravellingSalesman(Graph &graph);

/**
 * Função que carrega o número de vértices do arquivo
NOME_DO_ARQUIVO_DE_DADOS
 */
int carregarNumeroDeVertices();
```

```
/**
 * No método main, todas as funcionalidades são testadas.
 *
 */
int main(void) {

    // O número de vértices é determinado pelo valor de NUMERO_DE_VERTICES
para simplivicação do script
    int numeroDeVertices = NUMERO_DE_VERTICES;

    // Grafo com arestas criadas uma a uma.
    cout << " >> Efetuando teste de grafo com arestas diversas << " << endl
<< endl;
    Graph graph = Graph(numeroDeVertices);
    inserirEdges(graph);
    removerEdges(graph);
    graph.printAdjacencyMatrix();
    exibirBfs(graph);
    exibirDfs(graph);
    exibirDijkstra(graph);
    exibirTravellingSalesman(graph);
    graph.mst();

    cout << endl << "Total de conexoes: " << graph.getTotalConnections() <<
endl << endl << flush;

    // Grafo completo
    cout << " >> Efetuando teste de grafo completo << " << endl << endl;
    Graph graphCompleto = Graph(numeroDeVertices);
    graphCompleto.complete();
    graphCompleto.printAdjacencyMatrix();
    exibirBfs(graphCompleto);
    exibirDfs(graphCompleto);
    exibirDijkstra(graphCompleto);
    exibirTravellingSalesman(graphCompleto);
    graphCompleto.mst();
    cout << endl << "Total de conexões: " <<
graphCompleto.getTotalConnections() << endl << endl << flush;

    // Grafo com arestas carregadas de arquivo.
    cout << " >> Efetuando teste de grafo com arestas carregadas de arquivo
.csv << " << endl << endl;
    numeroDeVertices = carregarNumeroDeVertices();
    Graph graphCsv = Graph(numeroDeVertices);
    inserirEdgesCsv(graphCsv);
    graphCsv.printAdjacencyMatrix();
    exibirBfs(graphCsv);
    exibirDfs(graphCsv);
    exibirDijkstra(graphCsv);
    exibirTravellingSalesman(graphCsv);
    graphCsv.mst();
    cout << endl << "Total de conexões: " << graphCsv.getTotalConnections()
```

```
<< endl << endl << flush;

    return 0;

}

void exibirBfs(Graph &graph){
    for(int i = 0 ; i < NUMERO_DE_VERTICES ; i++){
        graph.bfs(CharUtil::toLetter(i));
    }
}

void exibirDfs(Graph &graph){
    for(int i = 0 ; i < NUMERO_DE_VERTICES ; i++){
        graph.dfs(CharUtil::toLetter(i));
    }
}

void exibirDijkstra(Graph &graph){
    for(int i = 0 ; i < NUMERO_DE_VERTICES ; i++){
        for(int j = 0 ; j < NUMERO_DE_VERTICES ; j++){
            graph.dijkstra(CharUtil::toLetter(i), CharUtil::toLetter(j));
        }
    }
}

void exibirTravellingSalesman(Graph &graph){
    for(int i = 0 ; i < NUMERO_DE_VERTICES ; i++){
        graph.travellingSalesman(CharUtil::toLetter(i));
    }
}

void inserirEdges(Graph &graph){
    cout << " *** Criando arestas para teste do grafo *** " << endl <<
endl;

    int v1=0, v2=3, weight=5;
    if(graph.insert(Edge(v1,v2,weight))){
        cout << "- " << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2
<< ")." << endl;
    }else{
        cout << "- " << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
")." << endl;
    }

    v1=3, v2=1;
    if(graph.insert(Edge(v1,v2))){
        cout << "- " << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2
<< ")." << endl;
    }else{
        cout << "- " << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
")." << endl;
    }
```

```
    v1=3, v2=4;
    if(graph.insert(Edge(v1,v2))){
        cout << "- " << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2
<< ")." << endl;
    }else{
        cout << "- " << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
")." << endl;
    }

    v1=2, v2=2;
    if(graph.insert(Edge(v1,v2))){
        cout << "- " << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2
<< ")." << endl;
    }else{
        cout << "- " << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
")." << endl;
    }

    v1=1, v2=4, weight=2;
    if(graph.insert(Edge(v1,v2, weight))){
        cout << "- " << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2
<< ")." << endl;
    }else{
        cout << "- " << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
")." << endl;
    }

    v1=4, v2=3;
    if(graph.insert(Edge(v1,v2))){
        cout << "- " << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2
<< ")." << endl;
    }else{
        cout << "- " << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
")." << endl;
    }

    v1=4, v2=3;
    if(graph.insert(Edge(v1,v2))){
        cout << "- " << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2
<< ")." << endl;
    }else{
        cout << "- " << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
")." << endl;
    }

}

int carregarNumeroDeVertices(){
    cout << " *** Criando arestas para teste do grafo via arquivo " <<
NOME_DO_ARQUIVO_DE_DADOS << " *** " << endl << endl;

    fstream file;
    file.open(NOME_DO_ARQUIVO_DE_DADOS, ios::in);
    string line;
```

```
    getline(file, line);
    stringstream ss( line );
    string data;
    getline( ss, data, '\n' );

    return stoi(data);

}

void inserirEdgesCsv(Graph &graph){
    cout << "Criando arestas para teste do grafo via arquivo " <<
NOME_DO_ARQUIVO_DE_DADOS << " *** " << endl << endl;

    fstream file;
    file.open(NOME_DO_ARQUIVO_DE_DADOS, ios::in);
    string line;
    bool first = true;
        while (getline(file, line)){
        if(first){
            first = false;
            continue;
        }

        stringstream ss( line );
        vector<string> row = vector<string>();
        string data;
        while ( getline( ss, data, ',' ) ){
            row.push_back( data );
        }

        string v1 = row[0];
        string v2 = row[1];
        string weight = row[2];;

        if(graph.insert(Edge(stoi(v1),stoi(v2),stoi(weight)))){
            cout << MSG_ARESTA_ADICIONADA << " (" << v1 << ", " << v2 <<
")." << endl;
        }else{
            cout << "" << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2
<< ")." << endl;
        }

    }

}

void removerEdges(Graph &graph){

    int v1=3, v2=4;
    if(graph.remove(Edge(v1,v2))){
        cout << "- " << "-" << MSG_ARESTA_ADICIONADA << " (" << v1 << ", "
<< v2 << ")." << endl;
    }else{
        cout << "-" << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
```

```
").." << endl;
    }

    v1=4, v2=2;
    if(graph.remove(Edge(v1,v2))){
        cout << "- " << "-" << MSG_ARESTA_ADICIONADA << " (" << v1 << ", "
<< v2 << ")." << endl;
    }else{
        cout << "-" << MSG_ARESTA_REMOVIDA << " (" << v1 << ", " << v2 <<
").." << endl;
    }

}

#endif
```

Edge.h

```
#ifndef EDGE_H
#define EDGE_H

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


/**
 * Classe que representa uma aresta no grafo.
 *
 * @author 2017089014 - Alex Souza <alexdcesouza@gmail.com>
 * @author 2017001320 - Raphael Ribeiro <raphaelribeiro@ufmg.br>
 *
 * @see Edge.h
 *
 * Obs: Os atributos estão documentados no arquivo .h, já os métodos,
 * no arquivo .cpp
 *
 */
class Edge{
    private:
        /**
         * Vérice 1 da ligação
         */
        const int v1;

        /**
         * Vértice 2 da ligação
         */
        const int v2;

        /**
         * Peso da ligação, por padrão, igual a 1.
```

```
         */
        const int weight = 1;

    public:
        Edge(const int &v1, const int &v2, const int &weight);
        Edge(const int &v1, const int &v2);
        int getV1() const;
        int getV2() const;
        int getWeight() const;
        ~Edge();
};

#endif
```

Edge.cpp

```
#ifndef EDGE_CPP
#define EDGE_CPP

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

/**
 * Classe que representa uma aresta no grafo.
 *
 * @author 2017089014 - Alex Souza <alexdcesouza@gmail.com>
 * @author 2017001320 - Raphael Ribeiro <raphaelribeiro@ufmg.br>
 *
 * @see Edge.h
 *
 * Obs: Os atributos estão documentados no arquivo .h, já os métodos,
 * no arquivo .cpp
 *
 */

/**
 * Construtor para ser usado em casos em que o peso da aresta é importante.
 *
 * @param int &v1 - o vértice 1 da ligação
 * @param int &v2 - o vértice 2 da ligação
 * @param int &weight - o peso da ligação, quando relevante. Por padrão 1;
 *
 */
Edge::Edge(const int &v1, const int &v2, const int &weight) : v1(v1),
v2(v2), weight(weight){};

/**
 * Construtor para ser usado em casos em que o peso da aresta não é
importante.
 *
```

```
 * @param int &v1 - o vértice 1 da ligação
 * @param int &v2 - o vértice 2 da ligação
 *
 */
Edge::Edge(const int &v1, const int &v2) : v1(v1), v2(v2){};

int Edge::getV1() const{
    return v1;
}

int Edge::getV2() const{
    return v2;
}

int Edge::getWeight() const{
    return weight;
}

Edge::~Edge(){
}

#endif
```

Graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <queue>
#include "Edge.h"
#include "Vertex.h"

using namespace std;

/**
 * Classe que representa um grafo referente à tarefa:
 * "Crie uma classe para representar o conceito de
 * grafo (classe Graph), com vértices representados por números inteiros
(índices dos
 * vértices). A classe Graph utilizará uma representação interna por matriz
de adjacência."
 *
 * @author 2017089014 - Alex Souza <alexdcesouza@gmail.com>
 * @author 2017001320 - Raphael Ribeiro <raphaelribeiro@ufmg.br>
 *
 * @see Graph.h
 *
 * Obs: Os atributos estão documentados no arquivo .h, já os métodos,
 * no arquivo .cpp
```

```
 *
 */
class Graph{
    private:

        /**
         * Número de vértices indicados na construção do objeto. Indica as
dimensões da matriz de ajacência.
         */
        const int numeroDeVertices;

        /**
         * A matriz de adjacência que representa as arestas entre os
vértices.
         */
        int **adjacencyMatrix;

        /**
         * Array útil para o cálculo de alguns métodos. Indica os vértices
visitados na busca por caminhos entre eles.
         */
        bool *visited;

        /**
         * Atributo que efetua a contagem do caminho percorrido ao calcular
as ligações entre os vértices. Usado em alguns cálculos.
         */
        int lenghtOfPath;

        /**
         * Fila para controle de cálculo de diversos algoritmos
         */
        queue<int> edgeQueue;

        /**
         * Método auxiliar para limpar indicações de vertices visitados no
cálculo de buscas (Depth First Search – DFS).
         *
         */
        void resetVisited();

        int minKey(int key[], bool mstSet[]);

        int performDfs(const char &vertex, const bool printOutput = true);


    public:

        /**
         * Atributo estático que indica o número de arestas criadas pela
classe. Como mais de um grafo
         * pode ser criado, dentro do universo estudado, é interessante que
se saiba quantas arestas foram criadas
         * mesmo que não façam parte do mesmo grafo.
```

```
         *
         */
        static int totalEdges;
        static int getTotalEdges();
        Graph(int &numeroDeVertices);
        bool insert(const Edge &edge);
        bool remove(const Edge &edge);
        int getTotalVertex();
        bool edge(const Edge &edge) const;
        bool isComplete() const;
        void complete();
        void bfs(const char &vertex);
        void dfs(const char &vertex);
        void printAdjacencyMatrix() const;
        int getTotalConnections();
        void mst();
        void dijkstra(const char &initialVertex, const char &finalVertex);
        void travellingSalesman(const char &vertex);

        ~Graph();

};


#endif
```

Graph.cpp

```
#ifndef GRAPH_CPP
#define GRAPH_CPP

#include <iostream>
#include <queue>
#include <bits/stdc++.h>
#include "Graph.h"
#include "Edge.h"
#include "Vertex.h"
#include "CharUtil.h"

using namespace std;

/**
 * Classe que representa um grafo referente à tarefa:
 * "Crie uma classe para representar o conceito de
 * grafo (classe Graph), com vértices representados por números inteiros
(índices dos
 * vértices). A classe Graph utilizará uma representação interna por matriz
de adjacência."
 *
 * @author 2017089014 - Alex Souza <alexdcesouza@gmail.com>
 * @author 2017001320 - Raphael Ribeiro <raphaelribeiro@ufmg.br>
```

```
 *
 * @see Graph.h
 *
 * Obs: Os atributos estão documentados no arquivo .h, já os métodos,
 * no arquivo .cpp
 *
 */
int Graph::totalEdges = 0;


/**
 * Construtor do grafo. Recebe um inteiro que será usado para determinar o
tamanho da matriz de adjacência (quadrada).
 *
 * a) Um construtor, que receberá como parâmetro um inteiro indicando o
número de
 *    vértices do grafo;
 *
 * @param int &numeroDeVertices - O número de vértices do qual o grafo é
composto.
 */
Graph::Graph(int &nv) :
    numeroDeVertices(nv){
    visited = new bool[numeroDeVertices];
    resetVisited();

    adjacencyMatrix = new int*[numeroDeVertices];
    for(int i = 0 ; i < numeroDeVertices ; i++){
        adjacencyMatrix[i] = new int[numeroDeVertices];
    }
}

/**
 * Método que possibilita a inclusão de uma aresta entre dois vértices do
grafo. Preencherá a matriz de adjacëncia.
 *
 * 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.
 *
 * @param Edge &edge - Uma instância de Edge que encapsula as informações
de uma aresta a ser representada no grafo.
 */
bool Graph::insert(const Edge &e){
    if(e.getV1() >= numeroDeVertices
        || e.getV1() >= numeroDeVertices
        || e.getV1() == e.getV2()
        || edge(e)){
        return false;
    }
    adjacencyMatrix[e.getV1()][e.getV2()] = e.getWeight();
    adjacencyMatrix[e.getV2()][e.getV1()] = e.getWeight();
```

```
    Graph::totalEdges++;
    return true;
}

/**
 * Método que remove uma aresta definida anteriormente 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.
 *
 * @param Edge &edge - Uma instância de Edge que encapsula as informações
de uma aresta a ser removida do grafo.
 */
bool Graph::remove(const Edge &e){
    if(e.getV1() >= numeroDeVertices
        || e.getV1() >= numeroDeVertices
        || e.getV1() == e.getV2()
        || ! edge(e)){
        return false;
    }
    adjacencyMatrix[e.getV1()][e.getV2()] = 0;
    adjacencyMatrix[e.getV2()][e.getV1()] = 0;

    Graph::totalEdges--;
    return true;
}

/**
 * Método estático que retornará o número total de arestas criadas em todos
os grafos repersentados por esta classe (Graph).
 *
 * 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;
 *
 * @TODO: Revisar o método e o atributo não deveriam ser membros privados e
não estáticos.
 *
 */
int Graph::getTotalEdges(){
    return Graph::totalEdges;
};

/**
 * Método que fornece o número total de vértices do 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;
 *
 */
int Graph::getTotalVertex(){
    return numeroDeVertices;
};

/**
 * Método que indica se uma aresta existe no grafo.
 *
 * 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.
 *
 * @param Edge &edge - Uma instância de Edge que encapsula as informações
de uma aresta a ser removida do grafo.
 *
 */
bool Graph::edge(const Edge &edge) const{
    int *arr = adjacencyMatrix[edge.getV1()];
    return arr[edge.getV2()] != 0;
};

/**
 * Método que indica se o grafo é completo (todos os vértices possuiem
arestas entre si).
 *
 * g) Função booleana para verificar se o grafo desenhado é completo.
 *
 */
bool Graph::isComplete() const{
    for(int i = 0 ; i < numeroDeVertices ; i++){
        for(int j = 0 ; j < numeroDeVertices ; j++){
            if(i==j){
                continue;
            }
            if(!edge(Edge(i,j))){
                return false;
            }
        }
    }
    return true;
};

/**
 * Método que completa a matriz de adjacência adicionando arestas em todos
```

```
os vertices para todos os outros.
 *
 * h) Função para completar o grafo desenhado.
 *
 */
void Graph::complete(){
    for(int i = 0 ; i < numeroDeVertices ; i++){
        for(int j = 0 ; j < numeroDeVertices ; j++){
            if(i==j){
                continue;
            }
            if(!edge(Edge(i,j))){
                insert(Edge(i, j));
            }
        }
    }
};


/**
 * Efetua o cálculo da (Breath First Search – BFS) a partir de um vértice
para todos os outros. Imprime
 * todas as ligações do vértice fornecido como argumento.
 *
 * 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.
 *
 * @param char &vertex - o vértice a partir do qual se deseja efetuar a
busca
 *
 */
void Graph::bfs(const char &vi) {
    cout << endl << " *** Caminhamentos BFS de: " << vi << " *** " << endl
<< endl<< flush;
    bool first = true;
    int vertex = CharUtil::toInt(vi);

    bool *visited = new bool[numeroDeVertices];
    for(int i = 0; i < numeroDeVertices; i++){
        visited[i] = false;
    }

    queue<int> queue;

    visited[vertex] = true;
    queue.push(vertex);

    while(!queue.empty())
    {
```

```
        vertex = queue.front();
        if(first){
            first = false;
        }else{
            cout << "->" ;
        }

        cout << CharUtil::toLetter(vertex);
        queue.pop();

        for(int i = 0 ; i < numeroDeVertices ; i++)
        {
            if (!visited[i] && edge(Edge(i, vertex)))
            {
                visited[i] = true;
                queue.push(i);
                vertex = i;
            }
        }
    }
    cout << endl << endl;

};

/**
 * Efetua o cálculo da (Depth First Search – DFS) a partir de um vértice
para todos os outros.
 *
 * 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.
 *
 * @param char &vertex - o vértice a partir do qual se deseja efetuar a
busca
 *
 */
void Graph::dfs(const char &v){
    cout << endl <<" *** Caminhamentos DFS de: " << v << " *** " << endl <<
endl;
    int count = 0;
    for(int i = 0 ; i < numeroDeVertices ; i++){
        int vertex = CharUtil::toInt(v);
        visited[vertex] = true;
        if(edge(Edge(vertex, i))){
            cout << (++count) << ". ";
            cout << v;
            int count = performDfs(CharUtil::toLetter(i));
            resetVisited();
            cout << " (" << count << " passos)" << endl;
        }
```

```
    }
    cout << endl;
    resetVisited();
};

/**
 * Efetua o cálculo da (Depth First Search – DFS)
 *
 * 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.
 *    Função para retornar o número de componentes conectad
 *
 */
int Graph::performDfs(const char &v, const bool printOutput) {
    int count = 1;
    int vertex = CharUtil::toInt(v);
    if(visited[vertex]){
        return count;
    }
    if(printOutput){
        cout << "->" << CharUtil::toLetter(vertex);
    }
    count++;
    visited[vertex] = true;
    int *neighbours = adjacencyMatrix[vertex];
    for(int i = 0 ; i < numeroDeVertices ; i++){
        if(neighbours[i]==1){
            count += performDfs(CharUtil::toLetter(i), printOutput) -1 ;
        }
    }
    return count;
};

void Graph::resetVisited(){
    lenghtOfPath = 0;
    for(int i = 0 ; i < numeroDeVertices ; i++){
        visited[i] = false;
    }
}


/**
 * Método útil para impressão da matriz de adjacência em tela para
visualização dos valores
 *
 */
void Graph::printAdjacencyMatrix() const{

    cout << endl << endl;
```

```
    cout << "   ";
    for(int j = 0 ; j < numeroDeVertices ; j++){
        cout << CharUtil::toLetter(j) << " ";
    }

    cout << endl;
    for(int i = 0 ; i < numeroDeVertices ; i++){
        cout << endl;
        cout << CharUtil::toLetter(i) << "  ";
        for(int j = 0 ; j < numeroDeVertices ; j++){
            cout << adjacencyMatrix[i][j] << " ";
        }
    }

    cout << endl << endl;
    cout << "Numero de arestas : " << Graph::getTotalEdges() << endl;
    cout << "Numero de vertices: " << numeroDeVertices << endl;
    cout << "Matriz completa   : " << (isComplete()?"sim":"não") << endl;
    cout << endl;

}

/**
 * Método que fornece o número total de conexões presentes no 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.
 *
 */
int Graph::getTotalConnections(){
    int count = 0;
    for(int i = 0 ; i < numeroDeVertices ; i++){
        for(int j = 0 ; j < numeroDeVertices ; j++){
            visited[i] = true;
            if(edge(Edge(i, j))){
                count += performDfs(CharUtil::toLetter(j), false);
            }
        }
    }

    cout.clear();

    resetVisited();
    return count;
};

/**
 * Método auxiliar que fornece o valor mínimo para a geração da árvore
geradora mínima (MST)
 */
int Graph::minKey(int key[], bool mstSet[]){
    int min = INT_MAX, min_index;
```

```
    for (int i = 0; i < numeroDeVertices; i++){
        if (mstSet[i] == false && key[i] < min) {
            min = key[i];
            min_index = i;
        }
    }
    return min_index;
}

/**
 * Método que produz a árvore geradora mínima do grafo.
 *
 * n) Função para encontrar uma árvore geradora mínima de um grafo com peso
nas
 *    arestas.
 *
 */
void Graph::mst(){

    cout << endl << endl << " *** Arvore geradora minima *** " << endl <<
endl;

    int parent[numeroDeVertices];
    int key[numeroDeVertices];
    bool mstSet[numeroDeVertices];

    for (int i = 0; i < numeroDeVertices; i++){
        key[i] = INT_MAX, mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int i = 0; i < numeroDeVertices - 1; i++){

        int u = minKey(key, mstSet);
        mstSet[u] = true;

        for (int j = 0; j < numeroDeVertices; j++){
            if (edge(Edge(u, j)) && mstSet[j] == false &&
adjacencyMatrix[u][j] < key[j]){
                parent[j] = u;
                key[j] = adjacencyMatrix[u][j];
            }
        }

    }

    cout << "Aresta\t\tPeso\n";
    for (int i = 1; i < numeroDeVertices; i++){
        if(parent[i] < 0 || adjacencyMatrix[i][parent[i]] == 0){
            continue;
        }
        cout
```

```
            << "( "
            << CharUtil::toLetter(parent[i])
            << ", "
            << CharUtil::toLetter(i)
            << " )"
            << "\t"
            << adjacencyMatrix[i][parent[i]]
            << "\n";
    }
    cout << endl;

}

/**
 * Método que calcula o caminho pelo algoritmo de Dijkstra.
 *
 * 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.
 *
 * @param char &initialVertex - O vértice de partida para o cálculo.
 * @param char &finalVertex - O vértice de destino.
 *
 */
void Graph::dijkstra(const char &vi, const char &vf){

    cout << endl << endl << " *** Dijkstra de " << vi << " ate " << vf << "
*** " << endl << endl;

    if(vi == vf){
        cout << vi << endl << "Custo: 0" << endl << flush;
        return;
    }

    int vertex = CharUtil::toInt(vi);
    int finalVertex = CharUtil::toInt(vf);
    int dist[numeroDeVertices];
    bool sptSet[numeroDeVertices];

    for (int i = 0; i < numeroDeVertices; i++){
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }

    dist[vertex] = 0;

    for (int count = 0; count < numeroDeVertices-1; count++){

        int min = INT_MAX;
        int u;

        for (int v = 0; v < numeroDeVertices; v++){
```

```
            if (sptSet[v] == false && dist[v] <= min){
                min = dist[v];
                u = v;
            }
        }

       sptSet[u] = true;

        for (int v = 0; v < numeroDeVertices; v++){
            if ( ! sptSet[v]
                && adjacencyMatrix[u][v]
                && dist[u] != INT_MAX
                && dist[u] + adjacencyMatrix[u][v] < dist[v]){

                dist[v] = dist[u] + adjacencyMatrix[u][v];

            }

        }

    }

    int totalDist = 0;
    bool reachEnd = false;
    cout << vi << "(0)";
    for (int i = 0; i < numeroDeVertices; i++){
        if(i == vertex || dist[i] == INT_MAX ){
            continue;
        }
        cout << "->" << CharUtil::toLetter(i)<<"("<<dist[i] <<")";
        totalDist += dist[i];
        if(i == finalVertex){
            reachEnd = true;
            break;
        }
    }
    if( ! reachEnd){
        cout.clear();
        cout  << endl << "[INFO] Nao chega ao destino";
    }
    cout << endl << "Custo: " << totalDist << endl << flush;

}

/**
 * Método que resolve o problema do caixeiro viajante.
 *
 * m) Função para resolver o Problema do Caixeiro Viajante. Essa função
deverá completar o
 *    grafo caso o mesmo não seja completo.
 *
 * @param char &vertex - O vértice de partida.
 *
 */
```

```
void Graph::travellingSalesman(const char &vi){
    cout << endl << endl << " *** Caixeiro viajante de " << vi << " *** "
<< endl << endl;
    if( ! isComplete() ){
        cout << "[INFO] Completando a matriz de adjacência" << endl;
        complete();
    }
    cout << vi;

    int vertex = CharUtil::toInt(vi);
    vector<int> ver = vector<int>();

    for (int i = 0; i < numeroDeVertices; i++){
        if (i != vertex){
            ver.push_back(i);
        }
    }

    int minPath = INT_MAX;
    do {

        int currentPathweight = 0;

        int k = vertex;
        for (int i = 0; i < ver.size(); i++) {
            currentPathweight += adjacencyMatrix[k][ver[i]];
            k = ver[i];
        }
        currentPathweight += adjacencyMatrix[k][vertex];

        minPath = min(minPath, currentPathweight);

        if(!visited[k]){
            visited[k] = true;
        }

    } while (next_permutation(ver.begin(), ver.end()));

    for (int i = 0; i < numeroDeVertices; i++) {
        if(visited[i]){
            cout << "->" << CharUtil::toLetter(i) << flush;
        }
    }

    cout << endl << "[INFO] Custo do percurso: " << minPath << endl <<
flush;
    resetVisited();

}

/**
 * Destrutor da classe.
 *
 * b) Um destrutor, que se incumbirá de fazer a desalocação de memória
```

```
eventualmente
 *    utilizada na representação do grafo;
 *
 */
Graph::~Graph(){

};

#endif
```

I/O

Entradas e saídas do software.

vertex.txt

Arquivo que contém vértices a serem importados para uso no grafo. A primeira linha contém o tamanho do grafo (número de vértices). A partir da segunda linha, os valores das arestas são adicionados na orde: vértice 1, vértice 2 e peso.

```
10
0,4,1
2,4,2
3,1,3
0,1,1
2,3,1
```

Saída do Console

```
 >> Efetuando teste de grafo com arestas diversas <<

 *** Criando arestas para teste do grafo ***

- Aresta adicionada com sucesso (0, 3).
- Aresta adicionada com sucesso (3, 1).
- Aresta adicionada com sucesso (3, 4).
- Não foi possíve remover a aresta (2, 2).
- Aresta adicionada com sucesso (1, 4).
- Não foi possíve remover a aresta (4, 3).
- Não foi possíve remover a aresta (4, 3).
- -Aresta adicionada com sucesso (3, 4).
-Não foi possíve remover a aresta (4, 2).


   A B C D E

A  0 0 0 5 0
```

```
B  0 0 0 1 2
C  0 0 0 0 0
D  5 1 0 0 0
E  0 2 0 0 0

Numero de arestas : 3
Numero de vertices: 5
Matriz completa   : não


 *** Caminhamentos BFS de: A ***

A->D->B->E


 *** Caminhamentos BFS de: B ***

B->D->A


 *** Caminhamentos BFS de: C ***

C


 *** Caminhamentos BFS de: D ***

D->A


 *** Caminhamentos BFS de: E ***

E->B->D->A


 *** Caminhamentos DFS de: A ***

1. A->D->B (3 passos)


 *** Caminhamentos DFS de: B ***

1. B->D (2 passos)
2. B->E (2 passos)


 *** Caminhamentos DFS de: C ***



 *** Caminhamentos DFS de: D ***

1. D->A (2 passos)
2. D->B (2 passos)
```

```
 *** Caminhamentos DFS de: E ***

1. E->B->D (3 passos)



 *** Dijkstra de A ate A ***

A
Custo: 0


 *** Dijkstra de A ate B ***

A(0)->B(6)
Custo: 6


 *** Dijkstra de A ate C ***

A(0)->B(6)->D(5)->E(8)
[INFO] Nao chega ao destino
Custo: 19


 *** Dijkstra de A ate D ***

A(0)->B(6)->D(5)
Custo: 11


 *** Dijkstra de A ate E ***

A(0)->B(6)->D(5)->E(8)
Custo: 19


 *** Dijkstra de B ate A ***

B(0)->A(6)
Custo: 6


 *** Dijkstra de B ate B ***

B
Custo: 0


 *** Dijkstra de B ate C ***

B(0)->A(6)->D(1)->E(2)
```

```
[INFO] Nao chega ao destino
Custo: 9


 *** Dijkstra de B ate D ***

B(0)->A(6)->D(1)
Custo: 7


 *** Dijkstra de B ate E ***

B(0)->A(6)->D(1)->E(2)
Custo: 9


 *** Dijkstra de C ate A ***

C(0)
[INFO] Nao chega ao destino
Custo: 0


 *** Dijkstra de C ate B ***

C(0)
[INFO] Nao chega ao destino
Custo: 0


 *** Dijkstra de C ate C ***

C
Custo: 0


 *** Dijkstra de C ate D ***

C(0)
[INFO] Nao chega ao destino
Custo: 0


 *** Dijkstra de C ate E ***

C(0)
[INFO] Nao chega ao destino
Custo: 0


 *** Dijkstra de D ate A ***

D(0)->A(5)
Custo: 5
```

```
 *** Dijkstra de D ate B ***

D(0)->A(5)->B(1)
Custo: 6


 *** Dijkstra de D ate C ***

D(0)->A(5)->B(1)->E(3)
[INFO] Nao chega ao destino
Custo: 9


 *** Dijkstra de D ate D ***

D
Custo: 0


 *** Dijkstra de D ate E ***

D(0)->A(5)->B(1)->E(3)
Custo: 9


 *** Dijkstra de E ate A ***

E(0)->A(8)
Custo: 8


 *** Dijkstra de E ate B ***

E(0)->A(8)->B(2)
Custo: 10


 *** Dijkstra de E ate C ***

E(0)->A(8)->B(2)->D(3)
[INFO] Nao chega ao destino
Custo: 13


 *** Dijkstra de E ate D ***

E(0)->A(8)->B(2)->D(3)
Custo: 13


 *** Dijkstra de E ate E ***
```

```
E
Custo: 0


 *** Caixeiro viajante de A ***

[INFO] Completando a matriz de adjacência
A->B->C->D->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de B ***

B->A->C->D->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de C ***

C->A->B->D->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de D ***

D->A->B->C->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de E ***

E->A->B->C->D
[INFO] Custo do percurso: 5


 *** Arvore geradora minima ***

Aresta          Peso
( A, B )        1
( A, C )        1
( B, D )        1
( A, E )        1


Total de conexoes: 24

 >> Efetuando teste de grafo completo <<



   A B C D E

A  0 1 1 1 1
B  1 0 1 1 1
```

```
C  1 1 0 1 1
D  1 1 1 0 1
E  1 1 1 1 0

Numero de arestas : 20
Numero de vertices: 5
Matriz completa   : sim


 *** Caminhamentos BFS de: A ***

A->B->C->D->E


 *** Caminhamentos BFS de: B ***

B->A->C->D->E


 *** Caminhamentos BFS de: C ***

C->A->B->D->E


 *** Caminhamentos BFS de: D ***

D->A->B->C->E


 *** Caminhamentos BFS de: E ***

E->A->B->C->D


 *** Caminhamentos DFS de: A ***

1. A->B->C->D->E (5 passos)
2. A->C->B->D->E (5 passos)
3. A->D->B->C->E (5 passos)
4. A->E->B->C->D (5 passos)


 *** Caminhamentos DFS de: B ***

1. B->A->C->D->E (5 passos)
2. B->C->A->D->E (5 passos)
3. B->D->A->C->E (5 passos)
4. B->E->A->C->D (5 passos)


 *** Caminhamentos DFS de: C ***

1. C->A->B->D->E (5 passos)
2. C->B->A->D->E (5 passos)
```

```
3. C->D->A->B->E (5 passos)
4. C->E->A->B->D (5 passos)


 *** Caminhamentos DFS de: D ***

1. D->A->B->C->E (5 passos)
2. D->B->A->C->E (5 passos)
3. D->C->A->B->E (5 passos)
4. D->E->A->B->C (5 passos)


 *** Caminhamentos DFS de: E ***

1. E->A->B->C->D (5 passos)
2. E->B->A->C->D (5 passos)
3. E->C->A->B->D (5 passos)
4. E->D->A->B->C (5 passos)



 *** Dijkstra de A ate A ***

A
Custo: 0


 *** Dijkstra de A ate B ***

A(0)->B(1)
Custo: 1


 *** Dijkstra de A ate C ***

A(0)->B(1)->C(1)
Custo: 2


 *** Dijkstra de A ate D ***

A(0)->B(1)->C(1)->D(1)
Custo: 3


 *** Dijkstra de A ate E ***

A(0)->B(1)->C(1)->D(1)->E(1)
Custo: 4


 *** Dijkstra de B ate A ***

B(0)->A(1)
```

```
Custo: 1


 *** Dijkstra de B ate B ***

B
Custo: 0


 *** Dijkstra de B ate C ***

B(0)->A(1)->C(1)
Custo: 2


 *** Dijkstra de B ate D ***

B(0)->A(1)->C(1)->D(1)
Custo: 3


 *** Dijkstra de B ate E ***

B(0)->A(1)->C(1)->D(1)->E(1)
Custo: 4


 *** Dijkstra de C ate A ***

C(0)->A(1)
Custo: 1


 *** Dijkstra de C ate B ***

C(0)->A(1)->B(1)
Custo: 2


 *** Dijkstra de C ate C ***

C
Custo: 0


 *** Dijkstra de C ate D ***

C(0)->A(1)->B(1)->D(1)
Custo: 3


 *** Dijkstra de C ate E ***

C(0)->A(1)->B(1)->D(1)->E(1)
```

```
Custo: 4


 *** Dijkstra de D ate A ***

D(0)->A(1)
Custo: 1


 *** Dijkstra de D ate B ***

D(0)->A(1)->B(1)
Custo: 2


 *** Dijkstra de D ate C ***

D(0)->A(1)->B(1)->C(1)
Custo: 3


 *** Dijkstra de D ate D ***

D
Custo: 0


 *** Dijkstra de D ate E ***

D(0)->A(1)->B(1)->C(1)->E(1)
Custo: 4


 *** Dijkstra de E ate A ***

E(0)->A(1)
Custo: 1


 *** Dijkstra de E ate B ***

E(0)->A(1)->B(1)
Custo: 2


 *** Dijkstra de E ate C ***

E(0)->A(1)->B(1)->C(1)
Custo: 3


 *** Dijkstra de E ate D ***

E(0)->A(1)->B(1)->C(1)->D(1)
```

```
Custo: 4


 *** Dijkstra de E ate E ***

E
Custo: 0


 *** Caixeiro viajante de A ***

A->B->C->D->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de B ***

B->A->C->D->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de C ***

C->A->B->D->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de D ***

D->A->B->C->E
[INFO] Custo do percurso: 5


 *** Caixeiro viajante de E ***

E->A->B->C->D
[INFO] Custo do percurso: 5


 *** Arvore geradora minima ***

Aresta          Peso
( A, B )        1
( A, C )        1
( A, D )        1
( A, E )        1


Total de conexões: 24

 >> Efetuando teste de grafo com arestas carregadas de arquivo .csv <<

 *** Criando arestas para teste do grafo via arquivo data/vertex.csv ***
```

```
Criando arestas para teste do grafo via arquivo data/vertex.csv ***

Aresta adicionada com sucesso (0, 4).
Aresta adicionada com sucesso (2, 4).
Aresta adicionada com sucesso (3, 1).
Aresta adicionada com sucesso (0, 1).
Aresta adicionada com sucesso (2, 3).


   A B C D E F G H I J

A  0 1 0 0 1 0 0 0 0 0
B  1 0 0 3 0 0 0 0 0 0
C  0 0 0 1 2 0 0 0 0 0
D  0 3 1 0 0 0 0 0 0 0
E  1 0 2 0 0 0 0 0 0 0
F  0 0 0 0 0 0 0 0 0 0
G  0 0 0 0 0 0 0 0 0 0
H  0 0 0 0 0 0 0 0 0 0
I  0 0 0 0 0 0 0 0 0 0
J  0 0 0 0 0 0 0 0 0 0

Numero de arestas : 25
Numero de vertices: 10
Matriz completa   : não


 *** Caminhamentos BFS de: A ***

A->B->D->C->E


 *** Caminhamentos BFS de: B ***

B->A->E->C->D


 *** Caminhamentos BFS de: C ***

C->D->B->A->E


 *** Caminhamentos BFS de: D ***

D->B->A->E->C


 *** Caminhamentos BFS de: E ***

E->A->B->D->C


 *** Caminhamentos DFS de: A ***
```

```
1. A->B (2 passos)
2. A->E (2 passos)


 *** Caminhamentos DFS de: B ***

1. B->A->E (3 passos)
2. B->D->C (3 passos)


 *** Caminhamentos DFS de: C ***

1. C->D (2 passos)
2. C->E->A->B (4 passos)


 *** Caminhamentos DFS de: D ***

1. D->B->A->E (4 passos)
2. D->C (2 passos)


 *** Caminhamentos DFS de: E ***

1. E->A->B (3 passos)
2. E->C->D (3 passos)



 *** Dijkstra de A ate A ***

A
Custo: 0


 *** Dijkstra de A ate B ***

A(0)->B(1)
Custo: 1


 *** Dijkstra de A ate C ***

A(0)->B(1)->C(3)
Custo: 4


 *** Dijkstra de A ate D ***

A(0)->B(1)->C(3)->D(4)
Custo: 8


 *** Dijkstra de A ate E ***
```

```
A(0)->B(1)->C(3)->D(4)->E(1)
Custo: 9


 *** Dijkstra de B ate A ***

B(0)->A(1)
Custo: 1


 *** Dijkstra de B ate B ***

B
Custo: 0


 *** Dijkstra de B ate C ***

B(0)->A(1)->C(4)
Custo: 5


 *** Dijkstra de B ate D ***

B(0)->A(1)->C(4)->D(3)
Custo: 8


 *** Dijkstra de B ate E ***

B(0)->A(1)->C(4)->D(3)->E(2)
Custo: 10


 *** Dijkstra de C ate A ***

C(0)->A(3)
Custo: 3


 *** Dijkstra de C ate B ***

C(0)->A(3)->B(4)
Custo: 7


 *** Dijkstra de C ate C ***

C
Custo: 0


 *** Dijkstra de C ate D ***
```

```
C(0)->A(3)->B(4)->D(1)
Custo: 8


 *** Dijkstra de C ate E ***

C(0)->A(3)->B(4)->D(1)->E(2)
Custo: 10


 *** Dijkstra de D ate A ***

D(0)->A(4)
Custo: 4


 *** Dijkstra de D ate B ***

D(0)->A(4)->B(3)
Custo: 7


 *** Dijkstra de D ate C ***

D(0)->A(4)->B(3)->C(1)
Custo: 8


 *** Dijkstra de D ate D ***

D
Custo: 0


 *** Dijkstra de D ate E ***

D(0)->A(4)->B(3)->C(1)->E(3)
Custo: 11


 *** Dijkstra de E ate A ***

E(0)->A(1)
Custo: 1


 *** Dijkstra de E ate B ***

E(0)->A(1)->B(2)
Custo: 3


 *** Dijkstra de E ate C ***
```

```
E(0)->A(1)->B(2)->C(2)
Custo: 5


 *** Dijkstra de E ate D ***

E(0)->A(1)->B(2)->C(2)->D(3)
Custo: 8


 *** Dijkstra de E ate E ***

E
Custo: 0


 *** Caixeiro viajante de A ***

[INFO] Completando a matriz de adjacência
A->B->C->D->E->F->G->H->I->J
[INFO] Custo do percurso: 10


 *** Caixeiro viajante de B ***

B->A->C->D->E->F->G->H->I->J
[INFO] Custo do percurso: 10


 *** Caixeiro viajante de C ***

C->A->B->D->E->F->G->H->I->J
[INFO] Custo do percurso: 10


 *** Caixeiro viajante de D ***

D->A->B->C->E->F->G->H->I->J
[INFO] Custo do percurso: 10


 *** Caixeiro viajante de E ***

E->A->B->C->D->F->G->H->I->J
[INFO] Custo do percurso: 10


 *** Arvore geradora minima ***

Aresta          Peso
( A, B )        1
( A, C )        1
( A, D )        1
```

```
( A, E )        1
( A, F )        1
( A, G )        1
( A, H )        1
( A, I )        1
( A, J )        1
```

CharUtil.h

```
#ifndef CHAR_UTIL_H
#define CHAR_UTIL_H

#include <iostream>

/**
 * Classe utilitária que auxilia na conversão entre letras e números.
 *
 * @author 2017089014 - Alex Souza <alexdcesouza@gmail.com>
 * @author 2017001320 - Raphael Ribeiro <raphaelribeiro@ufmg.br>
 *
 * @see CharUtil.h
 *
 */
class CharUtil{
    public:
        /**
         * Função utilitária que converte um inteiro em uma letra,
maiúscula,
         * corespondente ao número, começando de A (0 = 'A', 1 = 'B', 2 =
'C', ...)
         */
        static char toLetter(const int &i);

        /**
         * Função utilitária que converte um char em um inteiro
considerando A como o primeiro inteiro.
         * (0 = 'A', 1 = 'B', 2 = 'C', ...)
         */
        static int toInt(const char &c);
};

#endif
```

CharUtil.cpp

```
#ifndef CHAR_UTIL_CPP
#define CHAR_UTIL_CPP

#include <iostream>
```

```
#include "CharUtil.h"

using namespace std;

char CharUtil::toLetter(const int &i){
    return ((char)(i+65));
};

int CharUtil::toInt(const char &c){
    return int(c - 65);
};

#endif
```

Referências

Info	Autor	Link
Graph Theory Playlist	Willian Fiset	Playlist no Youtube
Estruturas de dados para grafos	Paulo Feofiloff	Link
Depth First Search - DFS	Willian Fiset	Vídeo no Youtube
Breadth First Search or BFS for a Graph	GeeksForGeeks (site, autor não identificado)	Link
Dijkstra's shortest path algorithm - Greedy Algo-7	Ita_c, estenger (Nicknames, mais informações no link)	Link
Traveling Salesman Problem (TSP) Implementation	Nishant_Singh	Link
Prim's Minimum Spanning Tree (MST) - Greedy Algo-5	vt_m, AnkurKarmakar, udkumar249, GlitchFinder, rathbhupendra (Nicknames, mais informações no link)	Link